BRENDER

# Technical Reference Manual

## BRender



## Argonaut Technologies Limited

Capitol House
Capitol Way
Colindale
London NW9 0DZ
United Kingdom

Telephone: +44 (181) 358 2993
Facsimile: +44 (181) 358 2920
Internet: brender@argonaut.com

## Acknowledgements

| | | |
|---|---|---|
| **Programming** | Sam Littlewood | Dan Piponi |
| | Simon Everett | Philip Pratt |
| **Ports** | Simon Everett | Philip Pratt |
| | Patrick Buckland | Zareh Johannes |
| | Anthony Savva | Stainless Software Ltd |
| **Testing** | Philip Pratt | |
| **Project Management** | Stefano Zammattio | |
| **Technical Reference Manual** | Crosbie Fitch | Philip Pratt |
| **Tutorial & Installation Guides** | Robbie McQuaid | |
| **Marketing** | Paul Ayscough | Mitra Faulkner |
| **Sales** | Edwin Masterson | |
| **Artwork** | Michel Bowes | |
| **Technical Support** | Neela Dass | Tony Roberts |
| | John Gay | |
| **With Thanks to** | Pete Warnes | Mike Day |
| | Tim Gummer | Vinay Gupta |
| | Marcia Petterson | Jez San |

## Disclaimer



## Trademarks

# Contents

# Contents

# Contents

# Contents

# 1 Introduction

# Overview

BRender is an extremely powerful real-time 3D graphics library, with a comprehensive Application Programming Interface (API), facilitating rapid and intuitive 3D development.

The libraries have an elegant conceptual design throughout and are, as far as possible, platform independent. In brief, they provide functions and data structures relevant to the following areas:

- Hierarchical organisation of objects constituting a scene
- Lights, cameras and models
- Efficient handling of items in a 'world database'
- Texture and environment mapping
- Rendering, in a number of styles
- Scene picking and elementary collision detection
- Mathematical primitives, including: scalars, angles, quaternions, Euler angles, vectors and matrices.
- Transparently selectable, fixed or floating point numeric representation
- File systems, memory management and diagnostics

Typically, a BRender application will perform the following tasks:

- Initialisation
- Importing or generating data (models, materials and textures)
- Rendering scenes
- Modification of object positions and orientations
- User interaction

# What is BRender?

BRender is a rendering engine, the 'bit of magic in the middle' that turns a scene into an image. It is the TV camera of virtual reality, translating an abstract description of a set, with its actors, cameras and lights, into a picture on a screen.

BRender is a combination of the most recent research in 3D graphics techniques and algorithms, and painstaking efforts to translate those algorithms into streamlined, lean, mean and highly optimised C code. Efficiency notwithstanding, BRender has always been designed with cross-platform portability, a wide range of useful features, and general purpose application, firmly in mind.

BRender takes the form of a C library, a set of C headers and library files that you build into your application. This lets you concentrate solely upon the task of describing scenes, modelling them and presenting them to the user. BRender takes care of all the hard work involved in positioning items in a scene, lighting them, applying special effects, and from a given camera specification, working out what's in the image, and rendering it (clipping, hidden surface removal, transparency, reflection, etc.).

The BRender C Library not only provides access to the rendering engine, but also facilities at each end of the process: describing a scene and processing the resulting image.

## BRender turns a description of a scene into an image

Scenes in 3D worlds are described to BRender in terms of components called actors. Each actor represents a frame of reference in which geometric models (objects, shapes, polyhedra, etc.) and other actors can be positioned and oriented. This scene description often builds up to a fairly sizable hierarchical structure.

Actors may perform a variety of different functions. There are three primary types of actor: model actors, which define shapes and their surface properties; light actors, which provide light with which models can be seen; and camera actors, which determine the viewpoint from which a scene is rendered into a 2D image.

It is the application's task to describe the scene in terms of an actor hierarchy, and BRender's task to turn this description into a 2D image, i.e. rendering it. However, although BRender is responsible for performing such rendering as efficiently as possible, the application is just as responsible for describing the scene as efficiently as possible. Both application and renderer must be optimised if high performance is to be achieved. BRender cannot compensate for a poorly written application. So, let's work in harmony to help produce extreme, jaw dropping applications.

## Why is BRender so good?

The development team behind BRender includes experts with considerable experience in various fields including: 3D scene description and modelling, 3D algorithms, 3D graphics hardware, video games development, advanced mathematics, compiler and microprocessor design.

Intense research and development over the last few years has culminated in what we believe is the best portable, all-round 3D rendering system available.

# 2 Organisation of this Manual

## Objective

The BRender Technical Reference Manual is designed to provide a reference for the developer who has started working with BRender, and to answer as many questions about the technical details of the BRender API as possible.

## Audience

This manual is for all users of BRender that require detailed technical information. It is not intended as a 3D graphics tutorial or C programming tutorial. 3D application developers and programmers will need to be fully conversant with contemporary 3D graphics technology (ideally also having an appreciation of the lower level aspects) and considerably experienced at C programming.

## Coverage

This manual only covers the BRender API, it does not cover platform specific support libraries, hardware or performance issues, nor does it provide introductory or tutorial material. Although it may provide some background, or suggestions for use, the manual is primarily concerned with providing reference documentation.

## Organisation

Given that this manual is likely to be used from beginner as much as expert or intermediate, there is inevitably a compromise that has to be made in how the manual is organised. While the beginner might appreciate a step by step, procedural oriented manual (getting gradually more sophisticated through the pages), an expert might prefer everything arranged alphabetically as a pure reference. The compromise we have made is to have an initial section which describes the API in a structured way, with each aspect described in context, followed by a section with data structures in alphabetical order and the set of functions relevant to them. This is suitable for the intermediate to advanced developer, who has got up to speed with BRender, but doesn't yet know it inside out. Thus, a more structured organisation (than purely alphabetical) is required, but not so structured as to impede location. There is always going to be a steep learning curve for such a complex product as BRender, so introductory and tutorial material has been supplied separately.

Note that while the installation guides cover some technical material associated with particular platforms, the BRender API is covered in a single volume and includes all the inevitable platform variations. This is useful for giving an appreciation of the differences between function and performance across platforms, and should encourage developers to cater for such variation at the design stage.

## *The API Function*

Each BRender function is described in a structured way using the following sub-headings:

| | |
|---|---|
| **Description** | The function is briefly described |
| **Declaration** | The simple C declaration |
| **Arguments** | Each argument or parameter to the function is described |
| **Preconditions** | Any entry conditions are detailed |
| **Effects** | Any effects (apart from producing a possible result) the function has are described |
| **Result** | The function result is described |
| **Remarks** | Any remarks concerning caveats, possible uses, misunderstandings, etc. |
| **Example** | A brief source code extract placing the function usage in context |
| **See Also** | Any data structures or functions that may be of relevant interest |

## *The Data Structure*

Each BRender data structure is described in a structured way using the following sub-headings:

| | |
|---|---|
| **The Structure** | The data structure is a C `struct` data type |
| **The Integral Type** | The data structure is a basic type, based on a fundamental type such as `int` or `long` |
| **The Call-back Function** | The data structure describes a function pointer, usually for a call-back function |
| **Members** | Public members of structures are described in detail |
| **Specification** | The task of a call-back function is defined |
| **Operations** | Functions operating upon one or more instances of this structure or type |
| **Arithmetic** | Operations enabling this structure or type to be operated upon as though it were a simple type such as `int` |
| **Comparison** | Functions providing a means to quantitively compare two instances of this structure or type |
| **Conversion** | Functions converting this structure or type into another |
| **Copy/Assign** | Methods and functions to use to assign or duplicate this structure or type |
| **Access & Maintenance** | How the data structure or type is accessed and maintained by BRender and the application |
| **Referencing & Lifetime** | How the data structure can be referenced, and how long it should be maintained by BRender and the application |
| **Initialisation** | How the data structure or type can be initialised |
| **Construction & Destruction** | How the data structure should be created in memory and destroyed |
| **Supplementary** | Informational functions available for the data structure, not particularly concerned with its function |
| **Import & Export** | How the data structure may be created from, or written to, an external file |
| **Platform Specific** | Any platform specific differences relating to this data structure or its functions |

# Using this Manual

It is a good idea to skim through the entire manual first, just to familiarise yourself with the API. If you have a particular question concerning an aspect of the API, try to locate relevant documentation in the manual using the contents, function index, full index, or by leafing through appropriate sections. All functions and data structures have a subscript cross-reference to the page number on which they are described.

# Typographic conventions

Throughout this manual, syntax, arguments, definitions and example code are in a typewriter-like font, to distinguish them from normal text. For example:

```
void BrTransformToTransform(br_transform *dest, const br_transform *src);
```

or

```
BR_ACTOR_CAMERA
```

Most public BRender functions have names of the form **`BrFunctionName()`**, i.e. with a prefix '**`Br`**' and a case-separated name*. Most public BRender data structures have names of the form **`br_structure_name`**, i.e. with a prefix **`br_`** and an underscore separated name.

* Note though, that due to historical precedent, pixel map is considered a single word in function names, thus **`BrPixelmapCopy()`**.

# 3 Functions (Structured Reference)

# Library Initialisation & Termination

## Initialisation

The BRender library requires more than static initialisation, and needs an opportunity to perform various initialisations before most BRender functions are called. There is, as yet, no standard for library initialisation, so the function **`BrBegin()`**[10] is provided as a means of explicitly initialising the BRender library.

However, there are a few things that should precede **`BrBegin()`**[10], and in the following order:

1. Install a diagnostic handler using **`BrDiagHandlerSet()`**[119] (See Diagnostic Support, page 66)
2. Install a memory allocator using **`BrAllocatorSet()`**[103] (See Memory Management, page 54)
3. Install a filing system using **`BrFilesystemSet()`**[130] (See Filing System Support, page 58)

If any of these steps are not performed, **`BrBegin()`**[10] will perform the equivalent to a usable default (appropriate to the platform). Which steps you perform depends upon your application's requirements.

---

## `BrBegin()`

*Description:* Initialise BRender. This function must be called before most BRender functions are used.

*Declaration:* **`void BrBegin(void)`**

*Preconditions:* Static initialisation has completed (`main()` has been entered). The BRender library has not yet been initialised, or has been terminated.

*Effects:* Installs any missing handlers (diagnostic handler, memory allocator, filing system) to appropriate platform defaults.

Initialises filing system (See Filing System Support, page 58)

Initialises registry (See Pre-Rendering Pre-Processing, page 29)

Initialises resource classes (See Resource Management, page 49)

Creates various defaults (models, materials, etc.)

*Remarks:* Do not call **`BrBegin()`**[10] again without a preceding **`BrEnd()`**[11].

*See Also:* **`BrEnd()`**[11]

---

## Termination

The BRender library should be closed down when it is no longer required (having first ensured the application has explicitly released all use of BRender facilities). This typically causes BRender to release all memory allocated in resource classes. Nevertheless, it may still be restarted using **`BrBegin()`**[10] (though handlers do not necessarily need to be reset).

## BrEnd()

| | |
|---|---|
| *Description:* | End BRender, freeing its internal resources and memory. |
| *Declaration:* | **`void BrEnd(void)`** |
| *Preconditions:* | The BRender library has been initialised and has not already been terminated. |
| *Effects:* | Releases all memory allocated in resource classes. This is effectively all memory allocated by BRender for its own use and all memory allocated in the various BRender allocation functions, such as **`BrModelAllocate()`**243. |
| *Remarks:* | Your application should have gracefully released all dependence upon BRender data and functions before calling **`BrEnd()`**11. |
| *See Also:* | **`BrBegin()`**10 |

# Scene Modelling

## The Actor

*See* **br_actor**$_{76}$

Scenes are described in terms of actors*, typically used to place models, lights and cameras. Nothing can see or be seen in a scene without actors. Of course it is still possible to perform 2D operations, such as copying a backdrop into a screen pixel map, or plotting text, but BRender's 3D effects are all obtained via actors.

The actor is intrinsically, just a means of orienting and positioning something with respect to something else. Thus a scene is organised around a system of actors. Admittedly, most actors are used to place models (visible shapes), but they can be used for a variety of other purposes, such as placing lights and cameras, and defining common frames of reference.

The function **BrActorAllocate()**$_{92}$ is one of the many ways of creating an actor.

## Hierarchical Relationships

*See* **br_actor**$_{76}$*{next,prev,children,parent,depth}*

A scene is described in terms of a single actor, however, that actor can be augmented by any number of other actors, which can be similarly augmented in turn. This gives rise to the tree-like, hierarchy of actors describing a scene. It is often described as though it were a family tree, with terms such as parent, child, and sibling used.

The **br_actor**$_{76}$ has various members defining its hierarchical relationship. The `next` and `prev` members describe a linked list of siblings, `parent` and `children` are self-evident and `depth` is equivalent to the generation of the actor.

The actor hierarchy is built up using functions such as **BrActorAdd()**$_{82}$. Naturally, there will be some occasions when the structure must be modified, and functions **BrActorRemove()**$_{83}$ and **BrActorRelink()**$_{83}$ are provided for this purpose.

## Positional Relationship

*See* **br_actor**$_{76}$*{t},* **br_transform**$_{349}$

Descendants, in additional to their hierarchical relationship, also have an associated positional and orientational relationship with their parent. An actor's position and orientation is defined solely with respect to its parent, rather than being relative to some absolute co-ordinate space. For some, this has drawbacks, but most applications benefit from the ease with which complex systems of inter-related models can be positioned. If an absolute frame of reference is required then actors can all be made children of the 'universe' parent.

When rendering BRender applies a general affine transformation, starting at the tree's root, between each actor and each one of its children. As the tree is traversed, the transformations are accumulated. This means that a simple modification to one actor's transform will affect the position and orientation

* The reason for the term 'actor' rather than 'object', is due to the possible confusion that might arise if 'object' was used, especially in these days of object oriented technology. Try thinking of the scene as a stage or film set, with 'actor' as a natural, umbrella term for each of the individual elements making it up.

of all its descendants. Note that the root of the hierarchy has no parent, and consequently, its transform has no meaning.

The transformation matrix that can be used to transform co-ordinates in one actor's co-ordinate space into that of another's, can be obtained by using the function **BrActorToActorMatrix34()** 84.

# Functions

*See* **br_actor** 76*{type,type_data,model,material,render_style}*, **br_model** 232, **br_material** 154

While actors should be thought of as elements in a structure of relative frames of reference, actors can be set to perform specific functions. Usually this is to be a place holder for a model (scenery or props, say).

There are three primary types of actors: models, lights and cameras. Models are visible 3D shapes defined in terms of geometry (vertices and faces) and surface (colour and materials). Lights are invisible actors that cause lighting effects upon models' surfaces. Cameras are invisible actors that define a particular viewpoint and perspective from which to view a scene.

There are various other functions an actor can provide, typically for assisting BRender by reducing unnecessary rendering.

The smallest hierarchy that will produce an actual rendered scene consists of four actors, namely a root, a model, a camera and a light source. The root actor is patently at the root of the hierarchy, and the model, camera and light actors are its children.

# The Reference Actor

*See* **br_actor** 76*{type}*

This actor type specified by the symbol `BR_ACTOR_NONE`, is typically used to assist the layout and organisation of a hierarchy. Although every actor defines its own frame of reference (co-ordinate space), for this actor that is all it does.

In spite of its lack of specialisation it still serves a useful purpose. For instance it may be convenient to represent a flock of birds with each bird positioned relative to some notional position of the flock, and then simply move the flock as a whole*. The position of the flock could be defined by a reference actor, and each bird a child model actor.

In general this type of actor is very useful when direct and independent control is required of each stage in a complex transform that needs to be applied to a model or system of models. Rather than recalculate this transform each time, each stage (typically very few) can be represented by a separate actor. For instance a reference actor could represent a translation and rotation, while the child model actor could be solely concerned with scaling the model.

Invariably, the root of a hierarchy will be a reference actor.

* Forgiveness being begged from those conducting painstaking research into flocking behaviour.

# The Model Actor

*See* **br_actor**$_{76}$*{type,type_data,model,material,render_style}*, **br_model**$_{232}$, **br_material**$_{154}$

The model actor is what it's all about; the linchpin of all 3D rendering – no scene should be without one.

As stated earlier, models are visible 3D shapes defined in terms of geometry (vertices and faces) and surface (colour and materials). A model actor is primarily defined by its `model` member, but also has the `material` member to define a default material to be used for parts of the model's surface that don't specify a material. The `render_style` member can also affect things by rendering the model in different ways. Some of these ways can be useful for things such as selection highlights or simple, but rapid rendering, e.g. wireframe style.

A feature of model actors is that they can inherit properties such as model, material and rendering style from their ancestors. Thus a flock of birds could consist of the same model, but have varying materials. If the model was changed then all birds would change. Note that the `model`, `material` and `render_style` members are still effective for purposes of defining inheritable model actor properties in other actors (even of type `BR_ACTOR_NONE`).

Create a model actor using **BrActorAllocate(**`BR_ACTOR_MODEL,NULL`**)**$_{92}$, then assign a model to the actor's `model` member.

## *The Model*

*See* **br_model**$_{232}$*{vertices,faces}*, **br_vertex**$_{378}$, **br_face**$_{125}$

The model is defined by a list of vertices and faces. Each vertex defines a vector from the model origin to a corner of a face, typically shared by two or more faces. Each face, representing a part of the surface of a model over which a material is rendered, is defined in terms of a series of vertices. Thus a cube can consist of eight vertices and six pairs of co-planar triangular faces, with each face specifying three vertices.

The model's geometry can be continuously modified, enabling powerful effects such as deformation and morphing. Because BRender may optimise a model's geometry, if you need to make modifications, you will generally need to retain the vertex and face data as originally specified, there are flags to control this such as `BR_MODF_KEEP_ORIGINAL`.

BRender maintains some useful information about models, such as their bounding radii and bounding boxes (see `radius` and `bounds` of **br_model**$_{232}$).

Create a model using **BrModelAllocate()**$_{243}$, ensuring that the model is added to the registry (**BrModelAdd()**$_{240}$) before the actor hierarchy is rendered.

## *The Vertex*

*See* **br_vertex**$_{378}$*{p}*, **br_vector3**$_{367}$

The vertex defines a point within a model's co-ordinate space that may be referred to by faces. By changing this point, faces referring to it will also change. The number of vertices in a model can be specified at the time the model is allocated.

### *The Face*

*See* **br_face**$_{125}${vertices}, **br_model**$_{232}$, **br_vertex**$_{378}$

The face defines itself in terms of a polygon, typically a triangle, with vertices specified by indices into the `vertices` list within the model. The order in which they are specified is important; it specifies the perimeter of the visible side of a face, going anti-clockwise around it. It doesn't matter which vertex is specified first.

The face also defines the way in which its surface should be rendered, by specifying a material.

The number of vertices in a model can be specified at the time the model is allocated.

### *The Material*

*See* **br_material**$_{154}${flags,ka,kd,ks,colour}, **br_colour**$_{114}$

Materials define exactly how faces should be rendered. This can be anything from a single colour value to a lit environment map. It all depends upon how realistic the surface is required to be, how much it should be influenced by lighting conditions, and the complexity of the colouring.

Materials are so called, because it is expected that most models represent physical materials such as wood, granite, wall-paper, etc. Predictably, the more realistic the material is required to be, the greater the processing overhead is likely to be. Nearly all the parameters of BRender materials are concerned with making a compromise between quality and performance. The simplest material is defined simply in terms of a colour value, and is not subject to any lights in the scene; this gives cartoon-like materials. Lighting effects can be added by specifying that the material is lit and smooth shaded; this is suitable for materials such as plastic and painted walls. Textured materials such as wood and marble, can be represented using texture maps, and can also be affected by lighting.

Naturally, the more processing that can be done beforehand, the less that needs to be performed during rendering. For this reason BRender provides the ability to specify prelit materials. This feature can be utilised in situations such as outdoor scenes, where the sun moves relatively slowly and thus need not be involved in frame by frame lighting calculations. Of course, things frequently changing orientation in the scene will still need to be lit normally.

### Lighting

A model's material can be affected by lights in a scene. The material's colours will effectively appear dimmer or brighter according to how well they are lit. This will depend upon the surface's orientation with respect to the viewer and each light in the scene.

BRender uses the Phong lighting model. The following formula shows how the lighting $\lambda$ of a face depends upon $\theta$, the angle at the face between the light source and the face normal, and $\phi$, the angle at the face between the viewer and the reflected light ray.

$$\lambda = k_{ambient} + k_{diffuse}\cos\theta + k_{specular}\cos^{p}\phi$$

The ambient factor is the amount of light assumed to be reflected from other objects and lighting in general. Zero can produce a material whose illumination is highly dependent upon light sources, whereas higher values can give ever fluorescent or luminous effects. A typical sunny scene might have most materials with a significant ambient contribution, whereas a dusk scene might have a much lower one, and a moonlit one, probably zero.

The diffuse factor determines how much of the reflected light is made up of the component dependent upon the angle of the face to the direction of the light illuminating it. The closer the face comes to being perpendicular to the light source, the more light the face receives, and thus the more diffuse light that can be reflected. Zero can give a shiny surface, whereas higher values can give surfaces a more matt appearance.

The specular factor determines how much reflected light is made up of the component dependent upon the angle between the reflected light source and the direction of viewer (naturally, if the angle is zero, the component will be at its maximum). The greater the value, the more prominent highlights will be. There is also the power of the cosine, and the greater this is, the sharper any highlights will be.

## Colours

*See* **br_colour**$_{114}$

BRender has an integral type dedicated to the task of completely defining a particular colour. It is used directly when specifying true colours, which is taken to mean any non-indexed colour (one not utilising a palette). Colour can either be taken to mean the colour of a screen pixel, the colour of light a surface reflects, or the colour of a light source within a scene.

The colour structure is currently 32 bits, made up of three (or four if you include an alpha component) bytes. You can construct a colour using the `BR_COLOUR_RGB()` macro.

## The Shade Table

*See* **br_material**$_{154}$*{flags,index_base,index_range,index_table,colour_map}*, **br_pixelmap**$_{277}$

For performance reasons (with the added benefit of lower memory requirements) textures, rendering or both can be performed using colour indices instead of colour values. Each colour index is converted into a colour value using a colour look up table (CLUT), often called a palette. When textures are made up of indices there is no straightforward way of lighting them. The brute force way would be to look up each texture index in the colour table, apply lighting to it to see what shade it should be, and then search through the colour table for the index of the colour value most closely matching this shade (which may not be very close at all). Because there isn't any processing power to waste, BRender implements a scheme using a shade table which for a given colour index and proportion of light, will give a colour index corresponding to the shaded colour of the original index.

Being two dimensional and storing pixel values, the shade table is quite suitably represented in the form of a pixel map. It has as many columns as there are colour indices and as many rows as there are distinct shades (from unlit to fully lit).

A shade table is created as a pixel map, commonly of type `BR_PMT_INDEX_8`, when rendering in 256 colour modes, for indexed textures with 256 columns ($2^8$ for textures of type `BR_PMT_INDEX_8`) and 64 rows. Use **BrPixelmapAllocate(**`BR_PMT_INDEX_8,256,64,NULL`**)**$_{290}$, not forgetting to call **BrTableAdd()**$_{288}$ before it is used in rendering.

Tools are generally available to take the effort out of making shade tables.

## The Texture Map

*See* **br_material**$_{154}$*{flags,colour_map,map_transform}*, **br_pixelmap**$_{277}$, **br_matrix23**$_{174}$

For textured materials, there is an additional problem of how a model's surface should be covered. This is a bit like the reverse of how to lay a map of the world on a flat sheet of paper. Similar, but

while textures are effectively flat, models are hardly ever spherical (even ellipsoid). When it comes to wrapping textures around complex models, tools are essential.

When it comes to library facilities, BRender provides a general mapping transform (how the flat texture should be warped to cover each face) and texture co-ordinates at each vertex (defining the coverage of the infinitely tiled texture plane by each face). There is also an option as to whether textures should be perspectively correct or not (a compromise between performance and warping). A feature of texture maps in BRender is that the texture map transform can be continuously modified, thus providing animated surfaces for insignificant performance overhead.

A texture can even be produced by from a scene rendering. This can provide effects such as television screens and rear view mirrors. It can be extended further in combination with the environment map (a first reflection ray trace) to provide mirrored surface effects. Remember though, that fast rendering is not only a product of efficient rendering algorithms, but also upon the application programmer's skill at reducing its workload. For instance a wall mirror could have the rest of the room's reflection precomputed into a texture map to be used as an environment map. If passers-by should also be reflected, then all that's required is a rendering of just the passer-by over this map (from the mirror's point of view).

A reflective effect is often a good substitute for a reflection. Christmas tree baubles for instance can be environment mapped without really needing to reflect any movement by anything else.

A texture map is created as a pixel map, usually with dimensions being powers of two. If an index texture map is used that will be lit, then remember to supply an appropriate shade table as well. To create a texture map use **BrPixelmapAllocate()** 290 followed by **BrMapAdd()** 287 before it is used by a rendered material.

# The Light Actor

*See* **br_actor** 76 *{type, type_data}*, **br_light** 147, **br_material** 154

While models are the only visible elements of a scene, as in reality they require light in order to be seen. For this reason light actors may be used to position and orient lights within a scene. Lights are not visible even if looked at directly. Or course, a brightly prelit model can be made a child of the light – thus making a fairly realistic spotlight model. Note though, that models are transparent to lights (without computationally intensive shadow processing, anyway).

If light actors are not used, there are only two ways of making surfaces visible. These are pre-lit textures and ambient lighting. For prelit textures pre-computed lighting levels at each vertex are stored in the vertex data structure and the `BR_MATF_PRELIT` flag specified with the material data structure. This is fine for relatively stationary models (such as buildings) that are primarily lit by relatively stationary light sources (such as the sun). Ambient lighting only, is really only appropriate for models that are moderately and evenly lit from all directions, but where this lighting level can change, e.g. ceilings in well lit rooms with 'dimmer' light controls, or outdoor scenes with ambient lighting affected by cloud cover or nightfall.

Having all the control of the actor of which they are a part, light actors' position and orientation can be fully controlled. This greatly facilitates situations requiring moving lights such as: roving spot lights, sun-rises and sunsets, headlights moving with the car, etc.

Light actors can be created using **BrActorAllocate(**`BR_ACTOR_LIGHT, NULL`**)** 92. A light specification will automatically be allocated and pointed to by the `type_data` member of the actor. Of course, an instance of a **br_light** 147 structure can be conventionally created and supplied as the

second argument instead of NULL. Note that light actors will need to be enabled for their lighting to affect the scene (see **BrLightEnable()**88).

### *The Light Specification*

*See* **br_light**147{type,colour}, **br_colour**114

Lights come in various forms. The simplest light is a direct light, which has the effect of a light a long way off, such as the sun or moon, or in some cases a flood-light. It certainly won't be very good for effects such as a candle lit scene. That sort of thing is a job for the point light, which can also be used for things such as light bulbs, fires, street lights, etc. For more precise control of light, the spot light may be used. That can be used, predictably enough, for such things as spot lights, search lights, torches, head lights, light-houses (where the beam runs along a cliff face say), etc.

Point and spot lights can also be controlled in terms of how rapidly their light diminishes over distance (attenuates). This is useful for candles and head-lights, i.e. a candle doesn't light up much more than the immediate vicinity, and objects in the path of head lights get brighter as they get nearer.

The spread of spot lights can also be controlled in terms of an inner fully lit cone and an outer penumbral cone between which the light tails off. Note that spot lights are really conically limited point lights, rather than diverging (or converging) from a sized disc. This difference will need to be appreciated when it comes to implementing cylindrical lights such as lasers and search lights, the laser is probably better implemented using the **BrScenePick3D()**84 function, locating the face and point of illumination and then directly modifying the screen pixel (found using **BrActorToScreenMatrix4()**86).

Light specifications can be created automatically at the time the light actor is created, or they can be created conventionally.

## The Camera Actor

*See* **br_actor**76{type,type_data}, **br_camera**110

There is a third ingredient to a scene, given a model to see and light to see it by, and this is an eye to see it with. Given our familiarity with photography and television, it is more useful to think of this eye as a camera that relays its image to our computer's monitor or TV screen. Then we have no problem with the possibilities of having more than one camera in the scene at the same time. Each camera defines how a 2D image is produced. Each image is invariably stored on a pixel map of particular dimensions. These images are then either incorporated into the 3D scene or another 2D image. Eventually, a 2D image will be produced for display on all or part of the screen.

The camera actor's position and orientation can like any other actor be continuously positioned and oriented. This allows things such as a driver's view round a race track, a bird's eye view, a fly-on-the-wall view, and any other view you can think of. Cameras are also useful for creating reflections and mirror views, TV screens, and crude shadow and lensing effects.

Camera actors can be created using **BrActorAllocate(**BR_ACTOR_CAMERA,NULL**)**92. A camera specification will automatically be allocated and pointed to by the type_data member of the actor. Of course, an instance of a **br_camera**110 structure can be conventionally created and supplied as the second argument instead of NULL.

### *The Camera Specification*

*See* **br_camera**$_{110}${type,field_of_view,aspect}

There are two basic types of camera, the parallel camera and the normal perspective one. The parallel camera produces an isometric image, i.e. faces are not scaled in the image according their distance from the viewer. The perspective camera on the other hand, according to its field of view, can range from 'almost parallel' to a fish eye lens effect (180° field of view). Somewhere in between we get a conventional view. If you wished to consider the screen as a window onto a virtual world, the field of view would be the angle subtended at the eye by the top and bottom of the screen. Of course, it depends how far away you are, but it can typically be between 10° and 40°. Given frequent exposure to the cinema and television screen, most people are quite happy with the larger end of the scale.

Note that the aspect of the camera must be specified to ensure that what is square in a scene remains square when it is displayed on the screen. Calculating the aspect boils down to simply measuring the physical width of the output image and dividing it by the physical height. In the process of producing homogenous screen co-ordinates, which are mapped to the full width and height of the output pixel map, the original x axis is scaled down by the camera aspect (see **BrMatrix4Perspective()**$_{228}$).The formula to compute the aspect is thus:

$$Aspect = \frac{\text{Pixel Map Width } P_w}{\text{Pixel Map Height } P_h} \times \frac{\frac{\text{Screen Width } S_w}{\text{Horizontal Resolution } H_{res}}}{\frac{\text{Screen Height } S_h}{\text{Vertical Resolution } V_{res}}} = \frac{P_w}{P_h} \times \frac{S_w V_{res}}{S_h H_{res}}$$

So if you have a screen measuring 8" by 6", which has a horizontal resolution of 320 and vertical resolution of 240, and you are rendering to an image 120 pixels across by 100 pixels high, then the aspect is obtained as follows:

$$Aspect_{eg} = \frac{120}{100} \times \frac{8 \times 240}{6 \times 320} = 1.2 \times \frac{1920}{1920} = 1.2$$

You'll notice that if a screen has square pixels (as in the above example) that the camera aspect simply needs to be the width of the output image divided by its height. The reason why BRender doesn't perform this calculation itself and have the 'aspect' as the ratio between the sides of the physical pixel, is because the image pixel map is assumed to be the same shape as the screen – how many pixels are on each edge, in BRender's view, is simply a matter of resolution.

Most camera views need to be in perspective, but there are a few good uses for parallel views other than for viewing CAD models: projecting sun shadows onto walls, creating height fields in depth buffers (camera looking down), creating aerial maps, telescopic views, etc. Parallel views can be sized using the `width` and `height` members (not forgetting the influence of `aspect`).

Remember, the camera does not have to be reserved for the player's view.

Camera specifications can be created automatically at the time the camera actor is created, or they can be created conventionally.

# Other Actors

*See* **br_actor**$_{76}$*{type,type_data}*

There are three other functions actors may perform. These are primarily concerned with assisting BRender in reducing unnecessary processing. The bounds actors are ways of saving BRender from performing some of its usual 'on screen' or 'off screen' checks for models. The clip-plane actor is a way of selecting out model actors that although are in the view volume, are not required in the image.

These other actors are created in a similar fashion to other actors, e.g. using **BrActorAllocate(**BR_ACTOR_BOUNDS,NULL**)**$_{92}$. A bounds or clip-plane specification will automatically be allocated and pointed to by the type_data member of the actor. Of course, an instance of the specification structure can be conventionally created and supplied as the second argument instead of NULL. Note that clip-plane actors will need to be enabled before they affect the scene (see **BrClipPlaneEnable()**$_{89}$).

## *The Bounds & Bounds Correct Actors*

*See* **br_actor**$_{76}$*{type,type_data}*, **br_bounds**$_{108}$*{}*, **br_vector3**$_{367}$

Bounds actors are a way for the application to assist the renderer in removing the necessity for many on/off screen* checks. Bounds actors only affect rendering in terms of enabling or disabling rendering of descendant model actors, they are not a visible part of the scene†. It is up to you to calculate the co-ordinates of the bounding box, BRender does not do this automatically, although you could possibly exploit the model's bounding box that it does calculate (upon **BrModelUpdate()**$_{241}$). As long as you ensure that the co-ordinates of each model's bounding box are transformed into the bounds actor's co-ordinate space, they can all be accumulated as appropriate.

You should use bounds actors for complex systems of models that have relatively compact configurations, or systems of models with models that do not need to be drawn if a certain, primary model is not visible, e.g. selection controls. The difference between BR_ACTOR_BOUNDS and BR_ACTOR_BOUNDS_CORRECT is that the latter represents a guarantee to the renderer that there is no surface of a descendant model that projects outside the bounding box. If there is any chance that a descendant actor's model may protrude use BR_ACTOR_BOUNDS, as the undefined behaviour resulting from the use of BR_ACTOR_BOUNDS_CORRECT includes anything, not necessarily just corruption of the screen, or aborting.

Note that in the cases where, in spite of using bounding box actors, the renderer neither knows that a model is entirely off screen nor entirely on screen, that the model's bounding box is used to determine whether the model should be rendered (or its call-back called). If the model is partially or wholly on screen, each face is clipped against the viewing volume and enabled clip plane actors.

## *The Clip-Plane Actor*

*See* **br_actor**$_{76}$*{type,type_data}*, **br_vector4**$_{376}$

Clip-plane actors in single-pass rendering are really only good for rendering simple effects such as crude cross-sectioning and selective lighting. For an example of selective lighting consider a room with a single spot-light being the only form of illumination. It is sometimes quicker (with a lot of

---

* On screen is defined as within the viewing volume and in front of all enabled clip planes.

† Do not be confused by BR_RSTYLE_BOUNDING_POINTS, etc. The bounding box referred to there is the one calculated by BRender, referenced within the **br_model** data structure.

objects in the room) to define a conical region with three clip-planes than solely rely on the attenuation of the spot-light to select-out unlit faces. Thus only objects entering the spot-light pyramid (and the viewing volume) will be visible and lit by the spot-light.

Clip-planes really come in to their own when used in multi-pass rendering. Shadows (as opposed to silhouettes) can be implemented by rendering one sectioned part of a scene with a direct light, and the other unlit (using only the ambient component), e.g. similar to the spot-light example above, a window frame could be used to define a fully lit pyramid using four clip-planes. The fully lit rendering has the planes facing inward, the shade rendering has the planes facing outward. Another lighting effect is a sunset (setting over a flat horizon such as the sea) where the tops of trees, buildings and hills are fully lit, and one or more lower sections have deeper hued and darker lights. A more obscure trick would be to use a clip plane to define the surface of a pool and render the pool and its contents slightly differently, possibly even using a different camera position to produce a refraction-like effect (though this would require rendering the pool image to an intermediate texture map).

Clip planes are not recommended as a way of pruning large actor hierarchies*. Clip planes are intended for their clipping effects, not their pruning effects.

A clip plane is defined by a four-vector. This is made up of a three-vector, being the unit normal to the plane, and the offset of the plane from the origin (in the direction of the normal). The equation of the plane is given as the dot product of this four-vector and the homogenous co-ordinates of a point on the plane, being equal to zero. Thus:

$$\mathbf{n}_p \cdot P_p = 0$$

Where $\mathbf{n}_p$ is the four-vector defining the plane in terms of a unit normal and offset, and $P_p$ is a point on the plane.

Descendants' faces are clipped against the plane, with the side defined by the normal being 'in scene'.

Note that clip-plane actors will need to be enabled before they affect the scene (see **`BrClipPlaneEnable()`**89).

# Co-ordinate Spaces

While this manual requires understanding of 3D graphics, so much confusion arises out of the variety of co-ordinate spaces, that a discussion is worthwhile - even in this document. BRender itself, only really ever deals with three co-ordinate spaces, those of the model, the view, and the projected screen. Nevertheless, it is often useful for the 3D applications developer to have other co-ordinate spaces in mind.

During rendering, every model's co-ordinate space is transformed through accumulation of actor transforms into the view space, and thence to projected screen space: primitive vertex data, pixel and depth co-ordinates.

Of particular note to those beginning 3D graphics: BRender has no concept of a world co-ordinate space (an absolute frame of reference).

---

* Actor hierarchies should be organised according to visibility so that pruning can be performed directly, rather than by using a clip plane.

### *Model co-ordinate space*

A geometric model consisting of vertices, and faces between them, has its own local right-handed co-ordinate system. There is an implicit origin at (0,0,0)*. An untransformed model, as seen by an unrotated camera (translated along its positive z axis so it faces the model), will have its positive axes pointing as follows: x to the right, y upwards, and the z pointing toward the viewer. A unit of 1 in the model will remain a unit of 1 in the view space unless any intervening actor transform involves a scaling.

### *Actor co-ordinate space*

The actor co-ordinate space is shared by its model (if it has one), but is relative to its parent actor's co-ordinate space through the use of a transform. The actor transform is defined as the transform which must be applied to points in the actor's co-ordinate space (of its model's vertices, say) for them to represent the same points in its parent actor's co-ordinate space.

### *World co-ordinate space*

Whether there is any notion of an absolute frame of reference, or co-ordinate space, is entirely up to the application. It may be appealing to think of a root actor as defining a world co-ordinate space, but this is entirely arbitrary, as BRender treats the root actor like any other. The root actor is so called, because it represents the immediate parent of each actor supplied for rendering (e.g. using **`BrZbSceneRenderAdd()`**$_{37}$), and the ancestor of the camera used for rendering. Its co-ordinate space is certainly not special. Of course, the root actor's transform is redundant as the co-ordinate space of its parent (if it has one) is never used (for a given rendering).

### *Camera co-ordinate space*

The only actor whose co-ordinate space might be regarded as special is that of the camera actor used for a particular rendering. It is into this actor's co-ordinate space that every model is transformed† (during rendering). See **`br_model_custom_cbfn`**$_{251}$ for details of transformation matrices and functions that can be used to convert model co-ordinates into view space, or screen space. The use of the term 'view space' is preferred to 'camera space'.

### *View space*

View space is effectively the camera co-ordinate space, the co-ordinate system in which the view volume is defined. The view volume is the section of the pyramid‡ defined by the camera's field of view from `-hither_z` to `-yon_z`, and aspect ratio, `aspect`. It is further defined by the origin of the output pixel map, `origin`. Note that the sides of the view volume correspond with the sides of the pixel map irrespective of its own aspect, therefore the camera's aspect is the only means of ensuring a correct aspect ratio is maintained.

---

* Though **`br_model`** has a pivot point which enables translation of the model origin.

† Effectively, anyway. Whether a model's vertices actually exist in this co-ordinate space at any moment is implementation dependent.

‡ A rectangular prism if using a parallel camera.

### *Homogenous screen space*

An intermediate phase in the transformation between view space and projected screen space is that of homogenous screen space. This is the viewing volume transformed into a cube, still with a right handed co-ordinate system, defined between (left, bottom, near) (-1,-1,+1) and (+1,+1,-1).

### *Projected screen space*

Projected screen space is the homogenous screen space transformed into co-ordinates suitable for rendering to the screen. That is, the x and y limits will correspond to the bounds of the pixel map, and the z limits will be mapped to the range [-32,768,+~32,767.9]. There are various functions dealing with such values, e.g. **`BrZbScreenZToDepth()`**$_{31}$, **`BrOriginToScreenXYZO()`**$_{256}$.

The projected screen space is still a right handed co-ordinate space, but the positive axes now point as follows: x right, y down, and z away from the viewer.

When 'screen space' is referred to, i.e. without any qualifier, it should be assumed that 'projected screen space' is intended.

### *Physical screen space*

There isn't really a physical screen space, but it can be thought of as the projected screen co-ordinates converted into values used directly by the rendering engines, i.e. x & y converted to integer pixel map co-ordinates, and z converted to z buffer depth or z sort depth.

## Converting between Co-ordinate Spaces

It is often necessary to convert from 3D space to 2D screen co-ordinates and vice versa. There are various functions that can assist in this. **`BrMatrix4Perspective()`**$_{228}$ will produce the matrix transformation that transforms view space (of a notional camera actor) into homogenous screen space (assuming a perspective projection). It is often more convenient to have a transform between an actor's co-ordinate space and the homogenous screen space, and **`BrActorToScreenMatrix4()`**$_{86}$ is provided for this purpose.

There are more extensive screen oriented functions available for use within custom model rendering call-backs (see **`br_model_custom_cbfn`**$_{251}$). These are basically, functions to convert model co-ordinates into screen co-ordinates (**`BrPointToScreenXY()`**$_{255}$), and a function to determine whether a model is on screen (**`BrOnScreenCheck()`**$_{254}$).

The homogenous screen co-ordinate space is a cuboid defined between (left, bottom, near) (-1,-1,+1) and (+1,+1,-1). The projected screen co-ordinate space is defined such that the scalar 2D x and y co-ordinates range across the output pixel map, i.e. between (left top) (`-origin_x,-origin_y`) and (`width-1-origin_x,height-1-origin_y`) and the z ordinate lies in the range (near to far) [-32,768,+~32,767.9].

There is an inverse relationship between z values in the view co-ordinate space and projected screen space z values. The conversion from a view z ordinate $z_{view}$ to the corresponding projected screen z ordinate $Screen_z$ is given by:

$$Screen_z = -2^{15} \cdot \frac{2z_{yon}z_{hither} - z_{view}(z_{yon} + z_{hither})}{z_{view}(z_{yon} - z_{hither})}$$

This result is the expansion of applying the transform obtained from **BrMatrix4Perspective()** 228 to a z ordinate, and then multiplying the result by $-2^{15}$.

Note that z buffer and z sort depth values are not necessarily the same as projected screen space, z ordinates. Functions are available to convert between depths, projected screen space and camera co-ordinate space.

---

# BrScreenZToCamera()

*Description:* Convert screen z [-32,768,+~32,767.9] to view z [-hither_z,-yon_z].

*Declaration:*
```
br_scalar BrScreenZToCamera(const br_actor* camera,
br_scalar sz)
```

*Arguments:*

**const br_actor* camera**

Pointer to camera actor.

**br_scalar sz**

Screen z value, e.g. as returned by **BrOriginToScreenXYZO()** 256.

*Result:* **br_scalar**

Corresponding z value in the camera actor's co-ordinate space (view space).

---

# BrScreenXYZToCamera()

*Description:* Convert point in screen space to point in a camera actor's co-ordinate space (view space) (compare with **BrPointToScreenXYZO()** 257).

*Declaration:*
```
void BrScreenXYZToCamera(br_vector3* point,
const br_actor* camera, const br_pixelmap* screen_buffer,
br_int_16 x, br_int_16 y, br_scalar zs)
```

*Arguments:* **br_vector3 * point**

A non-NULL pointer to the vector to hold the converted point in camera space.

**const br_actor * camera**

A non-NULL pointer to the camera actor into whose co-ordinate space the point is to be converted.

**const br_pixelmap * screen_buffer**

A non-NULL pointer to the screen buffer to which the x & y co-ordinates apply.

**br_int_16 x**

X co-ordinate of pixel.

**br_int_16 y**

Y co-ordinate of pixel.

**br_scalar zs**

Screen z co-ordinate.

*Preconditions:* Between **BrBegin()** 10 & **BrEnd()** 11. Between **BrZbBegin()** 28 & **BrZbEnd()** 41.

*Effects:* Computes the x & y co-ordinates in screen space, and together with the z co-ordinate applies the inverse projection transform, and stores the resulting vector at point.

*Example:*

```
br_vector3 p;
br_uint_32 depth;
br_scalar sz;
depth=BrPixelmapGet(&my_depth_buffer,x,y);
sz=BrZbDepthToScreenZ(depth,&my_camera);
BrScreenXYZToCamera(&p,&my_camera,&my_screen_buffer,x,y,sz);
```

*See Also:* **BrOriginToScreenXYZO()** 256, **BrPointToScreenXYZO()** 257, **BrMatrix4Perspective()** 228.

# Scene Rendering

## The Rendering Engine

Having described a 3D scene in terms of actors (models, lights, cameras, etc.) the next step is to produce a 2D image*. This is where BRender flexes its muscles and races through the actor hierarchy, accumulating transforms, computing lighting, and rendering faces. The process is performed in three phases:

### Phase 1

- Traverse the actor hierarchy and produce a stream of models, materials, transforms, and rendering styles

### Phase 2

- Discard back faces
- Produce a stream of lit 2D face elements (typically triangles or quads)
- Possibly perform binary space partitioning (for hidden surface removal)

### Phase 3

- Possibly perform hidden surface removal (ordering, z-buffering, etc.)
- Render faces to destination image (pixel map)

Each phase produces data for the next phase to work on. It is possible in some implementations for a phase to commence before the previous phase has ended, possibly even processing in parallel.

Depending upon the platform, BRender will exploit dedicated hardware wherever possible to perform some of the phases.

In terms of how the process appears when using the BRender API, the following gives a simplified C skeleton of a possible application.

```
BrBegin(); /* The BRender library is Initialised */
...
/* The application is initialised */
...
BrZbBegin();/* The rendering engine is initialised */
...
/* Set up a scene in terms of an actor hierarchy */
...
/* Elements in the scene are pre-processed by being added to the
   registry (models, materials, textures, shade tables) */
...
do/* Main loop */
{do /* Perform rendering of the scene from each camera */
{ ...
/* Start supply of scene details to renderer */
```

* It is important to keep in mind that a rendering produces a 2D image on a pixel map – not necessarily the screen.

```
BrZbSceneRenderBegin(world,camera,colour_buffer,depth_buffer);
   ...
   for (...; ...; ...)/* Rendering one or more actor hierarchies */
      { ...
      /* Specify each branch of world hierarchy to include in
   rendering */
         BrZbSceneRenderAdd(...);
         ...
      }
      ...
      /* End supply of scene details to renderer */
      BrZbSceneRenderEnd();
      ...
   } while (...);
   ...
   /* Update the scene */
   ...
} while (...);
...
BrZbEnd();/* Close down the rendering engine */
...
BrEnd();/* Terminate the BRender library */
```

In order for the three rendering phases to take place the application must call BRender API functions to enable it to traverse an actor hierarchy. A 'Begin' function (**`BrZbSceneRenderBegin()`**[35] in the example above) let's BRender know that we're about to supply it with a selection of actors in a specific hierarchy, that should be viewed from a certain camera actor within it, and the resulting image to be placed in a supplied pixel map, with a certain method used to remove hidden surfaces. Each actor in our selection to be included in the rendering is then supplied by using an 'Add' function (**`BrZbSceneRenderAdd()`**[37] in the example). When all of the selection has been added an 'End' function is called (**`BrZbSceneRenderEnd()`**[38] in the example). The output from the renderer is not defined until the 'End' function returns. Therefore you may make no assumptions about the contents of any output image buffers between the 'Begin' and 'End' functions, i.e. custom model and render bounds call-backs should not access the buffers. Furthermore, no assumption should be made regarding how the actor hierarchy is traversed during rendering, e.g. whether branch by branch or generation by generation.

## Initialising the Renderer

Before rendering anything, the rendering engine must be initialised. This should be performed after library initialisation and before anything is added to the registry. Given that the registry is for performing pre-rendering preprocessing, it unsurprisingly needs to know beforehand which rendering engine is going to be used.

## BrZbBegin()

*Description:* Initialise the Z-buffer renderer. This is a rendering engine which utilises a depth buffer (a pixel map matching the colour buffer) containing z values for each pixel, which can be used to determine whether another pixel at the same position should be drawn over the existing one.

*Declaration:* **`void BrZbBegin(br_uint_8 colour_type, br_uint_8 depth_type)`**

*Arguments:* **`br_uint_8 colour_type`**

Pixel map type of buffer to render into.

**`br_uint_8 depth_type`**

Pixel map type of Z-buffer.

*Preconditions:* Between **`BrBegin()`**[10] & **`BrEnd()`**[11]. The registry is empty. No rendering engine is currently enabled.

*Effects:* Checks that the specified colour and depth types can be supported, initialises registry for this renderer.

*See Also:* **`BrZbEnd()`**[41], **`BrZsBegin()`**[28]

## BrZsBegin()

*Description:* Initialise the Z-sort renderer. This is a rendering engine which uses a bucket sort to determine the order in which primitives (faces, lines, points) should be rendered.

*Declaration:* **`void BrZsBegin(br_uint_8 colour_type, void* primitive, br_uint_32 size)`**

*Arguments:* **`br_uint_8 colour_type`**

Pixel map type of buffer to render into.

**`void * primitive`**

Non-`NULL` pointer to an allocated block of memory to be used as a heap to hold rendered* primitives and referenced vertices generated during rendering.

**`br_uint_32 size`**

Size of primitive heap. To ensure that everything is rendered completely, this should be large enough for the renderer to store the temporary data structures used to hold details of each rendered primitive and the transformed vertices indexed by them. A primitive can be anything from a point to a triangle (possibly a quad). For the purposes of estimation you should allow 26 bytes for each primitive and 64 bytes for each unique vertex. The number of unique vertices can be calculated

* i.e. not back faces

from the number of primitives, according to how vertices are shared by models' faces. At best there will be an average of one vertex per face primitive (a tetrahedron - four faces, four vertices), at worst there will be an average of three vertices per face primitive (a scene of independent triangles)*.

Note that clipping is likely to increase the number of vertices.

If you estimate that there is an upper limit of about 1,000 faces (of front facing surfaces) in a rendering sequence with an average of 2.5 vertices per triangular face (including increases due to clipping) then the value of this member should be 1,000×(26+2.5×64), i.e. 186,000 bytes, call it 200KB.

*Preconditions:* Between **BrBegin()** 10 & **BrEnd()** 11. The registry is empty. No rendering engine is currently enabled.

*Effects:* Checks that the specified colour types can be supported, initialises registry for this renderer.

*Remarks:* If `size` is insufficiently large, some primitives will be omitted from the rendering. In border line cases, these are likely to be faces of models toward the end of the traversal of the actor hierarchy (which may manifest itself as deterioration of models in a particular lineage).

*See Also:* **BrZsEnd()** 41, **BrZbBegin()** 28

---

# Pre-Rendering Pre-Processing

There are various optimisations that BRender can perform before rendering takes place. These apply to models, materials, textures and shade tables. The way that any of these are specified to BRender before rendering is by adding them to the registry. The registry can be thought of as a container for holding elements involved in a scene. It can also be used as a simple type of database, with fairly simple search facilities. The preprocessing takes place on each item as it is added to the registry, e.g. using **BrModelAdd()** 240. Any time an item is modified the preprocessing that BRender needs to be performed, will have to be redone. This is achieved by explicitly updating changed items using functions such as **BrMaterialUpdate()** 162. As one would hope, such updating only needs to be performed just before rendering takes place – rather than after each change. If no actor in a scene uses a registry item then that item doesn't need to be updated. Conversely, if a actor uses a model, material, texture or shade table, that item must be in the registry and if changed must be updated before the actor is supplied to the renderer (even if indirectly).

If an item is no longer involved in a scene, then it can be removed from the registry using a function such as **BrTableRemove()** 289.

Functions to search the registry for a particular item or to enumerate over all items of a certain type can be found in the supplementary sections of structures **br_model** 232, **br_material** 154, and **br_pixelmap** 277.

* This may be higher if primitives are quads – consult your installation guide.

# Hidden Surface Removal Schemes

There are various schemes for rendering faces such that overlapping or intersecting faces are drawn correctly, i.e. nearer faces are in front of, and hide, farther faces. Each scheme is a different compromise between, quality, processing overhead, and memory overhead.

### *The Z-Buffer*

The z buffer, or depth buffer, is a fast, high quality scheme, but uses a considerable amount of memory (something not likely to be found in abundance on games consoles).

How the depth buffer works is platform dependent, but it generally consists of a memory buffer corresponding to the display buffer, with a depth value for each pixel. The values it contains typically represent distances from the front - zero would thus represent the position of the front clip plane. Each time a face is rendered, the z value of each pixel is converted into a depth value, and only if it has a nearer (lesser) z value than the one in the depth buffer will it be plotted (the depth value in the buffer is then updated).

BRender supports 16 and 32 bit Z-Buffers, and a description of how the current implementation generally works now follows.

The depth buffer contains unsigned values representing distance from the front clip plane. A view z ordinate of `-hither_z` (as specified in the camera) would correspond to a depth of zero. A view z ordinate of `-yon_z` corresponds to a depth of $Depth_{max}$, which is an unsigned value of `0xFFFFFFFF` (in 32 bpp depth buffers, but `0xFFFF0000` in 16 bpp).

Note that as depth buffer values are treated by the rendering engine as unsigned words, before rendering the depth buffer should be filled to the maximum depth, i.e. `0xFFFFFFFF`. The following example call will suffice whatever the word size of the depth buffer:

```
BrPixelmapFill(view->depth_buffer,0xFFFFFFFF);
```

Thus when reading values from a depth buffer, the value will be in the range [0, $Depth_{max}$], with unwritten pixels having depth buffer values of whatever was used to Initialise them, e.g. `0xFFFFFFFF`.

The functions **`BrZbBegin()`** [28], **`BrZbSceneRender()`** [34], etc. are provided for applications requiring a Z-Buffer rendering engine.

The following functions are provided to convert between depths, screen and view co-ordinates.

---

## `BrZbDepthToScreenZ()`

*Description:* Convert z buffer depth [0,0xFFFFFFFF] to screen z [-32,768,+~32,767.9].

*Declaration:*
```
br_scalar BrZbDepthToScreenZ(br_uint_32 depth_z,
const br_camera* camera)
```

*Arguments:* **br_uint_32 depth_z**

A 32 bit value read from the z buffer pixel map.

**const br_camera* camera**

A non-NULL pointer to the camera being used for rendering, i.e. relevant to the depth values used in the z buffer.

*Result:* **br_scalar**

The corresponding screen z value.

*See Also:* **BrScreenZToCamera()** 24, **BrScreenXYZToCamera()** 25.

## BrZbScreenZToDepth()

*Description:* Convert screen z [-32,768,+~32,767.9] to z buffer depth [0,0xFFFFFFFF].

*Declaration:* **br_uint_32 BrZbScreenZToDepth(br_scalar sz, const br_camera* camera)**

*Arguments:* **br_scalar sz**

A screen z value as obtained from functions such as **BrOriginToScreenXYZO()** 256.

**const br_camera* camera**

A non-NULL pointer to the camera being used for rendering, i.e. relevant to the depth values used in the z buffer.

*Result:* **br_uint_32**

A 32 bit depth value suitable for writing to a z buffer pixel map.

*See Also:* **BrScreenZToCamera()** 24, **BrScreenXYZToCamera()** 25.

### *The Z-Sort*

See **br_order_table** 264 { }

The Z-Sort is a very fast, low quality scheme using relatively little memory (particularly for simple scenes). Before rendering, all primitives (faces or smaller elements, such as lines or points) are sorted according to the z value of one or more of their vertices. The primitives are then rendered farthest first.

Clipping of intersecting faces involves a severe performance hit and is only likely to be implemented on very capable platforms. Checking for cyclic overlaps reduces performance still further.

The particular method BRender uses to sort primitives is to use a bucket sort. This involves the use of one or more order tables. Each order table divides a stratum (depth range) of the observer's z-axis into a number of equal sub-strata (depths). The model geometry of each model actor (using a particular order table) will be converted into rendering primitives, and each primitive will be added to the bucket (a linked list) corresponding to the sub-strata appropriate to its z-values. It is called a bucket sort because the order table may be considered as an ordered row of buckets into which items

are thrown (in any order) according to their value (one bucket per range of values). Thus the order in which primitives of the same sub-strata are drawn is undefined.

Each order table effectively defines a separate layer of the rendered image, like an acetate. One should thus take care if order tables' depth ranges overlap, to avoid far faces being drawn over near faces. This is only likely to be a problem for intersecting models in separate order tables. Order tables are drawn in order of `sort_z` (typically the nearest end of an order table).

The application is responsible for ensuring model actors are using appropriate order tables. The rendering process can then traverse the actor hierarchy, processing each model into primitives and inserting these into the appropriate order tables. Actors can be assigned to order tables specifically, or by inheritance - there is a default, single bucket order table (spanning the entire view volume) which will be inherited if no ancestral actor defines one. Each new order table that is encountered during this process is inserted into an ordered linked list of order tables. When complete, the list of order tables is traversed, with the primitives rendered from each bucket in turn (from back to front).

Performance is relatively independent of the number of buckets you have, given the use of a radix sort to determine into which bucket a primitive should be placed. Performance does decrease however, the more order tables you have. This is due to the overhead, as each new order table is encountered, required to insert the order table at the correct position in the linked list of order tables. This is only likely to be significant with a large number of order tables.

It is difficult to give guidelines, as approaches to determining an appropriate set of order tables depends so much on the type of scene (sometimes, even from frame to frame, as models rotate and intersect). Approaches can vary from having a single order table with a large number of buckets, to a single bucket order table for each model.

The functions **`BrZsBegin()`** [28], **`BrZsSceneRender()`** [35], etc. are provided for applications requiring a Z-Sort rendering engine.

The following functions provide means of converting between z sort depth values (as required by order tables and supplied to primitive call-back functions). Note that the z sort depths are actually in a different range from view z values, i.e. depths are in the range `[-hither_z,+yon_z]` whereas view z values are in the range `[-hither_z,-yon_z]` (both ranges near to far).

---

# BrZsDepthToScreenZ()

*Description:* Convert z sort depth [`-hither_z,+yon_z`] to screen z [-32,768,+~32,767.9].

*Declaration:*
```
br_scalar BrZsDepthToScreenZ(br_scalar depth_z,
const br_camera* camera)
```

*Arguments:* **`br_scalar depth_z`**

A depth value read from an order table or obtained from a z sort primitive call-back.

**`const br_camera* camera`**

A non-NULL pointer to the camera being used for rendering, i.e. relevant to the depth values used in primitives and order tables.

*Result:* **`br_scalar`**

The corresponding screen z value.

*See Also:* **BrScreenZToCamera()** 24, **BrScreenXYZToCamera()** 25.

## BrZsScreenZToDepth()

*Description:* Convert screen z [-32,768,+~32,767.9] to z sort depth [-hither_z,+yon_z].

*Declaration:* **br_scalar BrZsScreenZToDepth(br_scalar sz, const br_camera* camera)**

*Arguments:* **br_scalar sz**

A screen z value as obtained from functions such as **BrOriginToScreenXYZO()** 256.

**const br_camera* camera**

A non-NULL pointer to the camera being used for rendering, i.e. relevant to the depth values used in primitives and order tables.

*Result:* **br_scalar**

A depth value suitable for writing to an order table or comparing with values of z obtained from a z sort primitive call-back.

*See Also:* **BrScreenZToCamera()** 24, **BrScreenXYZToCamera()** 25.

### *Other Schemes*

Other hidden surface removal schemes exist and sometimes are a result of the rendering method used. These include:

Binary Space Partitioning
Voxels
Ray Tracing
Scan Line Rendering

# Rendering Functions

There are two ways of rendering a scene. Either the scene is defined in a single hierarchy and rendered as a whole, or a scene is defined in terms of a selection of subtrees of a single hierarchy. In the former case, only one function needs to be called, **Br[Zb|Zs]SceneRender()** 34|35, this effectively wraps up the three functions required for the latter case, **Br[Zb|Zs]SceneRenderBegin()** 35|36, **Br[Zb|Zs]SceneRenderAdd()** 37|37, and **Br[Zb|Zs]SceneRenderEnd()** 38|38. Note that in both cases, the camera must be a descendant of the scene root actor, as should all the sub-trees. If an environment actor is currently specified then it must also be a descendant of the scene root.

Note that there are two key call-back functions that can get called during rendering. These are a custom model rendering call-back (see **br_renderbounds_cbfn** *315*) and a render bounds call-back (see **br_model_custom_cbfn** *251*). The custom model rendering call-back is specified within the **br_model** *232* structure defined for a model actor and enables an application to make a model's rendering dependent upon information only available at the time it's processed by the renderer. For similar reasons, the render bounds call-back is called for every model that affects the output image,

allowing the application to take note of information only available at the time the model's are processed by the renderer. The trivial use of the render bounds call-back is to keep track of modified rectangles of the output image. The render bounds call-back is specified before rendering commences, using **Br[Zb|Zs]RenderBoundsCallbackSet()**[39|39].

There is also a special call-back, **br_primitive_cbfn**[315], only used in the Z-Sort renderer. This allows customised insertion of primitives into order tables.

---

# BrZbSceneRender()

*Description:* All-in-one function to render a scene using the Z-Buffer renderer.

*Declaration:* **void BrZbSceneRender(br_actor* world, br_actor* camera, br_pixelmap* colour_buffer, br_pixelmap* depth_buffer)**

*Arguments:* **br_actor * world**

A non-NULL pointer to the root actor of a scene.

**br_actor * camera**

A non-NULL pointer to a camera actor that is a descendant of the root actor.

**br_pixelmap * colour_buffer**

A non-NULL pointer to the pixel map to render the scene into, whose type is colour_type as supplied to **BrZbBegin()**[28].

**br_pixelmap * depth_buffer**

A non-NULL pointer to the pixel map to be used as a depth buffer whose type is depth_type as supplied to **BrZbBegin()**[28]. It must have the same width and height as the colour buffer. See **BrPixelmapMatch()**[292].

*Preconditions:* Between **BrBegin()**[10] & **BrEnd()**[11]. Between **BrZbBegin()**[28] & **BrZbEnd()**[41]. Not currently rendering.

*Effects:* Equivalent to a call of **BrZbSceneRenderBegin(**world,camera,colour_buffer,depth_buffer**)**[35] followed by **BrZbSceneRenderAdd(**world**)**[37] and **BrZbSceneRenderEnd()**[38].

*Remarks:* The colour buffer and depth buffer should not be texture maps (or even shade tables), though they can of course be subsequently added as such once the rendering has completed.

*See Also:* **BrZbRenderBoundsCallbackSet()**[39], **BrZbModelRender()**[253].

---

# BrZsSceneRender()

*Description:* All-in-one function to render a scene using the Z-Sort renderer.

*Declaration:* **void BrZsSceneRender(br_actor* world, br_actor* camera, br_pixelmap* colour_buffer)**

*Arguments:* **br_actor * world**

A non-NULL pointer to the root actor of a scene.

**br_actor * camera**

A non-NULL pointer to a camera actor that is a descendant of the root actor.

**br_pixelmap * colour_buffer**

A non-NULL pointer to the pixel map to render the scene into, whose type is colour_type as supplied to **BrZsBegin()**[28].

*Preconditions:* Between **BrBegin()**[10] & **BrEnd()**[11]. Between **BrZsBegin()**[28] & **BrZsEnd()**[41]. Not currently rendering.

*Effects:* Equivalent to a call of **BrZsSceneRenderBegin(**world,camera,colour_buffer**)**[36] followed by **BrZsSceneRenderAdd(**world**)**[37] and **BrZsSceneRenderEnd()**[38].

*Remarks:* The colour buffer should not be a texture map (or even a shade table), though it can of course be subsequently added as such once the rendering has completed.

*See Also:* **BrZsRenderBoundsCallbackSet()**[39], **BrZsModelRender()**[254], **BrZsPrimitiveCallbackSet()**[40].

---

# BrZbSceneRenderBegin()

*Description:* Set up a new scene to be rendered using the Z-Buffer renderer, processing the camera, lights and environment.

*Declaration:* **void BrZbSceneRenderBegin(br_actor* world, br_actor* camera, br_pixelmap* colour_buffer, br_pixelmap* depth_buffer)**

*Arguments:* **br_actor * world**

A non-NULL pointer to the root actor of a scene.

**br_actor * camera**

A non-NULL pointer to a camera actor that is a descendant of the root actor.

**br_pixelmap * colour_buffer**

A non-NULL pointer to the pixel map to render the scene into, whose type is colour_type as supplied to **BrZbBegin()**[28].

**br_pixelmap * depth_buffer**

A non-NULL pointer to the pixel map to be used as a depth buffer whose type is depth_type as supplied to **BrZbBegin()**[28]. It must have the same width and height as the colour buffer. See **BrPixelmapMatch()**[292].

*Preconditions:* Between **BrBegin()**[10] & **BrEnd()**[11]. Between **BrZbBegin()**[28] & **BrZbEnd()**[41]. Not currently rendering.

*Effects:* Enter rendering state, prepare for destination buffers, preprocess view, screen and environment transforms, preprocess enabled lights, handle environment actor, preprocess enabled clip planes.

*Remarks:* The colour buffer and depth buffer should not be texture maps (or even shade tables), though they can of course be subsequently added as such once the rendering has completed.

*See Also:* **BrZbSceneRenderAdd()** 37, **BrZbSceneRenderEnd()** 38, **BrZbRenderBoundsCallbackSet()** 39, **BrZbModelRender()** 253.

---

# BrZsSceneRenderBegin()

*Description:* Set up a new scene to be rendered using the Z-Sort renderer, processing the camera, lights and environment.

*Declaration:* **void BrZsSceneRenderBegin(br_actor* world, br_actor* camera, br_pixelmap* colour_buffer)**

*Arguments:* **br_actor * world**

A non-NULL pointer to the root actor of a scene.

**br_actor * camera**

A non-NULL pointer to a camera actor that is a descendant of the root actor.

**br_pixelmap * colour_buffer**

A non-NULL pointer to the pixel map to render the scene into, whose type is colour_type as supplied to **BrZsBegin()** 28.

*Preconditions:* Between **BrBegin()** 10 & **BrEnd()** 11. Between **BrZsBegin()** 28 & **BrZsEnd()** 41. Not currently rendering.

*Effects:* Enter rendering state, prepare for destination buffer, preprocess view, screen and environment transforms, preprocess enabled lights, handle environment actor, preprocess enabled clip planes.

*Remarks:* The colour buffer should not be a texture map (or even a shade table), though it can of course be subsequently added as such once the rendering has completed.

*See Also:* **BrZsSceneRenderAdd()** 37, **BrZsSceneRenderEnd()** 38, **BrZsRenderBoundsCallbackSet()** 39, **BrZsPrimitiveCallbackSet()** 40, **BrZsModelRender()** 254.

---

# BrZbSceneRenderAdd()

*Description:* Include an actor (and its descendants) of the world in the current rendering.

*Declaration:* **void BrZbSceneRenderAdd(br_actor* tree)**

*Arguments:* **br_actor * tree**

A non-NULL pointer to an actor hierarchy, which must be a descendant of the world hierarchy supplied to **BrZbSceneRenderBegin()** 35.

*Preconditions:* Between **BrBegin()** 10 & **BrEnd()** 11. Between **BrZbBegin()** 28 & **BrZbEnd()** 41. Currently rendering, i.e. between **BrZbSceneRenderBegin()** 35 and **BrZbSceneRenderEnd()** 38. Not within a custom model render call-back or render bounds call-back

*Effects:* Add actor to list of actors to be rendered.

*Remarks:* Whether rendering takes place during this function or sometime before the return of **BrZbSceneRenderEnd()** 38 is undefined. When custom model render and render bounds call-back functions are called is similarly undefined.

*See Also:* **BrZbSceneRenderBegin()** 35, **BrZbSceneRenderEnd()** 38, **BrZbRenderBoundsCallbackSet()** 39, **BrZbModelRender()** 253.

---

## BrZsSceneRenderAdd()

*Description:* Include an actor (and its descendants) of the `world` in the current rendering.

*Declaration:* **void BrZsSceneRenderAdd(br_actor* tree)**

*Arguments:* **br_actor * tree**

A non-`NULL` pointer to an actor hierarchy, which must be a descendant of the `world` hierarchy supplied to **BrZsSceneRenderBegin()** 36.

*Preconditions:* Between **BrBegin()** 10 & **BrEnd()** 11. Between **BrZsBegin()** 28 & **BrZsEnd()** 41. Currently rendering, i.e. between **BrZsSceneRenderBegin()** 36 and **BrZsSceneRenderEnd()** 38. Not within a custom model render call-back, render bounds call-back or primitive call-back.

*Effects:* Add actor to list of actors to be rendered.

*Remarks:* Whether rendering takes place during this function or sometime before the return of **BrZsSceneRenderEnd()** 38 is undefined. When custom model render, render bounds and primitive call-back functions are called is similarly undefined.

*See Also:* **BrZsSceneRenderBegin()** 36, **BrZsSceneRenderEnd()** 38, **BrZsRenderBoundsCallbackSet()** 39, **BrZsPrimitiveCallbackSet()** 40, **BrZsModelRender()** 254.

---

## BrZbSceneRenderEnd()

*Description:* Complete the specification of actors to be rendered in a scene, and their rendering.

*Declaration:* **void BrZbSceneRenderEnd(void)**

*Preconditions:* Between **BrBegin()** 10 & **BrEnd()** 11. Between **BrZbBegin()** 28 & **BrZbEnd()** 41. Currently rendering, i.e. after **BrZbSceneRenderBegin()** 35. Not within a custom model render call-back or render bounds call-back

*Effects:* By the time this function ends, the scene as specified in terms of world root, camera and sub-trees, will have been rendered to the output buffers.

*Remarks:* Whether rendering takes place during this function or sometime before the return of **BrZbSceneRenderEnd()** 38 is undefined. When custom model render and render bounds call-back functions are called is similarly undefined.

*See Also:* **BrZbSceneRenderBegin()** 35, **BrZbSceneRenderAdd()** 37, **BrZbRenderBoundsCallbackSet()** 39, **BrZbModelRender()** 253.

## BrZsSceneRenderEnd()

*Description:* Complete the specification of actors to be rendered in a scene, and their rendering.

*Declaration:* **void BrZsSceneRenderEnd(void)**

*Preconditions:* Between **BrBegin()**[10] & **BrEnd()**[11]. Between **BrZsBegin()**[28] & **BrZsEnd()**[41]. Currently rendering, i.e. after **BrZsSceneRenderBegin()**[36]. Not within a custom model render call-back, render bounds call-back, or primitive call-back.

*Effects:* By the time this function ends, the scene as specified in terms of world root, camera and sub-trees, will have been rendered to the output buffer.

*Remarks:* Whether rendering takes place during this function or sometime before the return of **BrZsSceneRenderEnd()**[38] is undefined. When custom model render, render bounds, and primitive call-back functions are called is similarly undefined.

*See Also:* **BrZsSceneRenderBegin()**[36], **BrZsSceneRenderAdd()**[37], **BrZsRenderBoundsCallbackSet()**[39], **BrZsPrimitiveCallbackSet()**[40], **BrZsModelRender()**[254].

## BrZbRenderBoundsCallbackSet()

*Description:* Set the call-back function invoked for each rendered actor. For example, a call-back can be set up to log those rectangles in the colour buffer that have been written to (dirty rectangle flagging).

*Declaration:* **br_renderbounds_cbfn* BrZbRenderBoundsCallbackSet(br_renderbounds_cbfn* new_cbfn)**

*Arguments:* **br_renderbounds_cbfn * new_cbfn**

A pointer to the new call-back function. Specify the old call-back function when a function call-back is not required.

*Preconditions:* Between **BrBegin()**[10] & **BrEnd()**[11]. Between **BrZbBegin()**[28] & **BrZbEnd()**[41]. Not currently rendering.

*Effects:* Defines a function that will be called during rendering for each model that affects the output buffers.

*Result:* **br_renderbounds_cbfn ***

Returns a pointer to the old call-back function.

*Remarks:* Exactly when the call-back function gets called is undefined except that it will be sometime between **BrZbSceneRenderBegin()**[35] and **BrZbSceneRenderEnd()**[38]. It is also not necessarily associated with a particular point in the rendering process.

*See Also:* **br_renderbounds_cbfn**[315], **br_model_custom_cbfn**[251].

## BrZsRenderBoundsCallbackSet()

*Description:* Set the call-back function invoked for each rendered actor. For example, a call-back can be set up to log those rectangles in the colour buffer that have been written to (dirty rectangle flagging).

*Declaration:*
```
br_renderbounds_cbfn*
BrZsRenderBoundsCallbackSet(br_renderbounds_cbfn*  new_cbfn)
```

*Arguments:* **br_renderbounds_cbfn * new_cbfn**

A pointer to the new call-back function. Specify the old call-back function when a function call-back is not required.

*Preconditions:* Between **BrBegin()** [10] & **BrEnd()** [11]. Between **BrZsBegin()** [28] & **BrZsEnd()** [41]. Not currently rendering.

*Effects:* Defines a function that will be called during rendering for each model that affects the output buffer.

*Result:* **br_renderbounds_cbfn ***

Returns a pointer to the old call-back function.

*Remarks:* The actor order table will be supplied to the call-back function.

Exactly when the call-back function gets called is undefined except that it will be sometime between **BrZsSceneRenderBegin()** [36] and **BrZsSceneRenderEnd()** [38]. It is also not necessarily associated with a particular point in the rendering process.

*See Also:* **br_renderbounds_cbfn**[315], **br_model_custom_cbfn**[251].

## BrZsPrimitiveCallbackSet()

*Description:* Set the call-back function invoked for each primitive generated by the Z-Sort renderer. This call-back can be used to perform customised insertion of primitives into order tables.

*Declaration:*
```
br_primitive_cbfn*
BrZsPrimitiveCallbackSet(br_primitive_cbfn* new_cbfn)
```

*Arguments:* **br_renderbounds_cbfn * new_cbfn**

A pointer to the new call-back function. Specify the old call-back function when a function call-back is not required. `NULL` will indicate that the default call-back function should be used.

*Preconditions:* Between **BrBegin()** [10] & **BrEnd()** [11]. Between **BrZsBegin()** [28] & **BrZsEnd()** [41]. Not currently rendering.

*Effects:* Defines a function that will be called during rendering for each front facing primitive that is to be inserted into the order table.

*Result:* **br_renderbounds_cbfn ***

Returns a pointer to the old call-back function.

*Remarks:* Exactly when the call-back function gets called is undefined except that it will be sometime between **BrZsSceneRenderBegin()**[36] and **BrZsSceneRenderEnd()**[38]. However, it may be assumed that rendering to the output buffer has not yet commenced.

*See Also:* **br_primitive_cbfn**[315], **br_model_custom_cbfn**[251].

## Terminating the Renderer

Once rendering has completed, all items in the registry should be removed (not necessarily freed) and the rendering engine closed down. This should also happen before using a different rendering engine or rendering to a different type of pixel map or image, e.g. rendering to an 8 bit pixel map when the current rendering is to a 24 bit video buffer.

# BrZbEnd()

*Description:* Close down the Z-Buffer renderer.

*Declaration:* **void BrZbEnd(void)**

*Preconditions:* Between **BrBegin()**[10] & **BrEnd()**[11]. **BrZbBegin()**[28] has been called, and **BrZbEnd()**[41] has not yet been called since. The registry is empty. No other rendering engine is currently enabled.

*Effects:* Releases resources used by the Z-Buffer renderer.

*See Also:* **BrZbBegin()**[28]

# BrZsEnd()

*Description:* Close down the Z-Sort renderer.

*Declaration:* **void BrZsEnd(void)**

*Preconditions:* Between **BrBegin()**[10] & **BrEnd()**[11]. **BrZsBegin()**[28] has been called, and **BrZsEnd()**[41] has not yet been called since. The registry is empty. No other rendering engine is currently enabled.

*Effects:* Releases resources used by the Z-Sort renderer.

*See Also:* **BrZsBegin()**[28]

# Image Support

## Pixel Maps

*See* **br_pixelmap**$_{277}$

In BRender, all two dimensional images are described in terms of pixel maps. A pixel map may represent any device that can be described in terms of width and height, and can support primitive graphical operations such as plot a point, line or triangle, e.g. video memory, a printer, or an area of memory. As far as BRender is concerned there is very little difference between them, it'll quite happily produce an image on a dot matrix printer as on an accelerated graphics card. Of course, the performance may be quite different depending on what device the pixel map refers to. On some platforms there may be graphic acceleration hardware which may only be usable on video memory, which may be inaccessible to BRender, thus rendering to a screen pixel map may be a few times faster than rendering to a pixel map based in main memory. Sometimes it's the other way around.

### *Using Pixel Maps*

In general, you'll use a pixel map for the following:

- The output image
- A depth buffer (if you're using the Z-Buffer renderer)
- An off-screen image (if you're using double buffering)
- Any sub-images copied to the output image
- Any images to be involved in the scene on the surface of models
- Texture maps for models' surfaces
- Shade tables for indexed texture maps

### *Operations*

The operations available on pixel maps include:

- setting points, lines, and rectangles to specific colours
- copying rectangles from one pixel map to another
- writing text

### *Rendering To Pixel Maps*

**BrPixelmapDirtyRectangleFill()**$_{43}$ and **BrPixelmapDirtyRectangleCopy()**$_{43}$ are useful when using results from within a render bounds call back function (see **br_renderbounds_cbfn**$_{315}$).

In some cases a scene rendering may only affect a few small areas of the destination pixel map. It can provide some performance improvement if only these areas are cleared each frame, rather than clearing the entire pixel map. The procedure is to utilise the **br_renderbounds_cbfn**$_{315}$ call-back, setting it before rendering (using **BrZbRenderBoundsCallbackSet()**$_{39}$), collect a list (or overall bounding rectangle) of bounds supplied, and use this to clear the destination pixel map just before the next rendering. Each rectangle can be cleared or reset using **BrPixelmapDirtyRectangleFill()**$_{43}$ or **BrPixelmapDirtyRectangleCopy()**$_{43}$. These functions are used because they can be faster, though they may utilise a larger rectangle than that specified (for more efficient word aligned operation).

## BrPixelmapDirtyRectangleFill()

*Description:* Set an area of a pixel map that covers a specified rectangle to a given value.

This function is intended to be used in conjunction with a rendering call-back to reset those regions of a pixel map that have been rendered to.

*Declaration:* **void BrPixelmapDirtyRectangleFill(br_pixelmap* dst,br_int_16 x, br_int_16 y, br_uint_16 w, br_uint_16 h,br_uint_32 colour)**

*Arguments:* **br_pixelmap * dst**

A pointer to the destination pixel map.

**br_int_16 x,y**

Co-ordinates of the rectangle's top left corner.

**br_uint_16 w,h**

Rectangle width and height (in pixels).

**br_uint_32 colour**

Value to set each pixel to.

*Effects:* Sets an area of pixels in the destination pixel map to a value, such that each pixel in the specified rectangle is reset.

*Remarks:* The actual area affected depends upon the platform and implementation, but this function is intended to provide the fastest way of resetting a particular rectangle. It is possible that this could be as large as the entire pixel map. Hopefully, most redundant, repeated calls would be ignored.

*Example:*

```
br_uint_32 zfar=0xffffffff;
br_int_16 drx, dry;
br_uint_16 drw, drh;
br_pixelmap* zbuffer;
...
BrPixelmapDirtyRectangleFill(zbuffer,drx,dry,drw,drh,zfar);
```

*See Also:* **BrZbRenderBoundsCallbackSet()** 39

## BrPixelmapDirtyRectangleCopy()

*Description:* Copy a rectangular window of data from one pixel map to the same position in another pixel map.

This function is intended to be used in conjunction with a rendering call-back to copy those regions of a pixel map that have been rendered to.

*Declaration:* **void BrPixelmapDirtyRectangleCopy(br_pixelmap* dst,const br_pixelmap* src, br_int_16 x, br_int_16 y,br_uint_16 w, br_uint_16 h)**

*Arguments:* **br_pixelmap * dst**

A pointer to the destination pixel map.

**const br_pixelmap * src**

A pointer to the source pixel map.

**br_int_16 x,y**

Co-ordinates of the rectangle's top left corner.

**br_uint_16 w,h**

Rectangle width and height (in pixels).

*Preconditions:* The source and destination pixel maps must have the same type and dimensions.

*Effects:* Copies an area of pixels from the source to the destination pixel map, such that each pixel in the specified rectangle is copied.

*Remarks:* The actual area copied depends upon the platform and implementation, but this function is intended to provide the fastest way of copying a particular rectangle. It is possible that this could be as large as the entire pixel map. Hopefully, most redundant, repeated calls would be ignored.

*Example:*

```
br_int_16 drx, dry;
br_uint_16 drw, drh;
br_pixelmap* offscreen;
br_pixelmap* backdrop;
...
BrPixelmapDirtyRectangleCopy(offscreen,
backdrop,drx,dry,drw,drh);
```

*See Also:* **BrZbRenderBoundsCallbackSet()**[39]

---

## *Pixel Maps To Video*

There are three ways of rendering to video:

1. Render direct to a pixel map representing a video device
2. Render to a memory based pixel map and copy it to a video pixel map each frame
3. Render to a memory based pixel map and swap it with a video pixel map each frame

All three methods suffer from problems of tearing, the first also from visible image construction. That is, if modifications to video memory are not postponed until the start of the CRT fly-back.

Fortunately, BRender provides a function **BrPixelmapDoubleBuffer()**[45] which should be used to avoid the possible problems as described above. This will perform, transparently, either of the last two methods. If the video device supports page swapping then it will utilise that feature, otherwise it will copy a pixel map to video memory. However, it will automatically do this at the start of the CRT fly-back, wherever possible.

## BrPixelmapDoubleBuffer()

*Description:* If the destination pixel map relates to a device, for example a graphics hardware screen, then the source 'off-screen' pixel map is copied to the destination pixel map at a suitable moment. If the source is an off screen pixel map (created using **BrPixelmapMatch(…,**BR_PMMATCH_OFFSCREEN**)** 292) then it is swapped with the destination pixel map instead.

Otherwise, the function is equivalent to **BrPixelmapCopy()** 285 and the source pixel map is copied to the destination pixel map.

*Declaration:* **void BrPixelmapDoubleBuffer(br_pixelmap* dst, br_pixelmap* src)**

*Arguments:* **br_pixelmap * dst**

A pointer to the destination pixel map.

**br_pixelmap * src**

A pointer to the source pixel map.

*Effects:* If the destination is a device that supports a 'wait for vertical retrace' function, a copy or swap will be performed pending that event.

If the destination is a device that supports double buffering and the source pixel map is an off-screen secondary buffer, the destination and source will be switched to use each other's buffers.

*Remarks:* Returns immediately, but will cause further rendering or calls of this function to block until the copy or swap has been completed.

# Pick Functions

A common function required of a 3D API, by developers working on a 3D user interface, is one that will return the model at a particular pixel. This is typically used for picking functions, e.g. mouse selection operations by the user. For this purpose, BRender provides the function **BrScenePick2D()** 86 which will invoke a specified call-back function for each model actor whose bounds contain the pixel. It is up to the application to determine greater precision, such as which model, faces or texture co-ordinates appear at the pixel.

There is also a 3D version of the pick function, called **BrScenePick3D()** 84. This will perform a similar operation, but operates on a bounding box rather than a screen pixel. However, if the image is depth buffered, the depth information for a pixel could be used in conjunction with **BrActorToScreenMatrix4()** 86 to produce a bounding box, more closely corresponding to the pixel resulting from a model's face.

# Maths

One of the essential aspects of any 3D graphics API is a good set of maths data types and functions.

Fractional representation is almost unavoidable when transforming 3D scenes into 2D co-ordinates, but there is usually a performance penalty associated when using floating point types, so BRender also provides a fixed point representation. This is in the form of an alternate library. The same types are used, so the switch between fixed and floating point is performed at compile time. Naturally there are range and precision considerations, but at least you have the choice. In fact, on some platforms with dedicated hardware, the floating point library is faster than the fixed point one.

3D scenes in terms of models, positions and transformations, are conventionally described in terms of Cartesian co-ordinates, vectors, and matrices. BRender provides data types and structures to cater for this, and goes even further, providing a dedicated angle type, the Euler Angle Set, for simpler rotation transforms, and the Unit Quaternion for sophisticated rotational transforms.

## C Types

Various BRender data structures and functions still make occasional use of basic C types such as `int` and `unsigned`. However, such use tends to be for simple integer quantities and flags. The floating point types are rarely used except of course for arguments and results of conversion functions.

## Fundamental Types

BRender defines a selection of sized signed and unsigned integer types, for uses where a specific word size is critical (see **`br_int_8/16/32`**$_{146}$ and **`br_uint_8/16/32`**$_{358}$).

BRender defines a selection of fixed point types for later use by application oriented types (see **`br_fixed_[ls][su][f]`**$_{132}$).

## Angles

Although a scalar type could suffice for an angle, it tends to end up representing rotations given that a scalar is not necessarily restricted to a range corresponding to a single rotation. For the purpose of strictly representing an angle (as opposed to a rotation), the **`br_angle`**$_{105}$ type is provided. By its nature, this type can only represent angles, readily appreciated given that it is a two byte type, with 65536 corresponding to 360º, i.e. 0º.

## Scalars

When it comes to representing numbers in general, particularly co-ordinates, a number is needed that can take on a wide range of values. A floating point number is ideal, but as not all platforms can deal with these (quickly or at all), BRender's general value type is library dependant. Using one library **`br_scalar`**$_{342}$ compiles to a `float`, using the other it compiles to a fixed point representation (**`br_fixed_ls`**$_{132}$).

# Fractions

Where storage is critical and values are always less than unity in magnitude, it makes sense to only store the fractional components of values. For this reason BRender defines signed and unsigned fraction types (see **br_fraction**$_{141}$ and **br_ufraction**$_{356}$). These are not generally expected to be used by the application, but may come in handy when large numbers of fraction based data structures are required.

# Vectors

*See* **br_vector2**$_{359}$, **br_vector3**$_{367}$, **br_vector4**$_{376}$.

The vector plays a central part in 3D geometry and BRender implements 2, 3 and 4 element vectors to enable efficient representations. 2D vectors can be implemented either using 2 element vectors (any homogenous element being implicit), or as 3 element vectors, with an explicit homogenous element. 3D vectors can be implemented either using 3 element vectors (any homogenous element being implicit), or as 4 element vectors, with an explicit homogenous element.

# Matrices

*See* **br_matrix23**$_{174}$, **br_matrix34**$_{190}$, **br_matrix4**$_{217}$.

With 2D texture and 3D geometry transformations being crucial to 3D rendering, BRender implements various matrix data structures and arithmetic functions. **br_matrix23**$_{174}$ is effectively a 2D 3x3 transformation matrix (for use with homogenous 2D co-ordinates) with an implicit third column. Similarly, **br_matrix34**$_{190}$ is effectively a 3D 4x4 transformation matrix (for use with homogenous 3D co-ordinates) with an implicit fourth column. **br_matrix4**$_{217}$ is a fully defined 3D 4x4 transformation matrix (for use with homogenous 3D co-ordinates).

Note that BRender applies a matrix to a vector by pre-multiplying the matrix by the vector. Furthermore, a matrix **A** is multiplied by matrix **B** (written **AB**) by computing the dot product of each row of **A** with each column of **B** for each element. Further still, **ABC** is understood to mean (**AB**)**C** (equivalent to **A**(**BC**)). Post-Multiply **M** by **A** means **MA**, and Pre-Multiply **M** by **A** means **AM**.

# Euler Angle Sets

*See* **br_euler**$_{121}$.

The Euler* Angle Set is a set of three angles and an ordering that represents a rotational transformation about each of the orthogonal axes. It is an easier and simpler way of specifying such transforms, rather than composing them out of rotational matrices.

# Unit Quaternions

*See* **br_quat**$_{320}$.

The quaternions form an extension to the real numbers that can be used to represent rotations. Just as we form the complex numbers by starting with the real numbers and including an extra number **i**

* Pronounced “Oiler” as in “Boiler”.

with the property $\mathbf{i}^2$=-1, the quaternions are formed by adding in three extra numbers: **i**, **j**, and **k**, with $\mathbf{i}^2=\mathbf{j}^2=\mathbf{k}^2$=-1. In addition these new numbers obey the following rules:

$$\mathbf{ij} = \mathbf{k} = -\mathbf{ji}$$
$$\mathbf{jk} = \mathbf{i} = -\mathbf{kj}$$
$$\mathbf{ki} = \mathbf{j} = -\mathbf{ik}$$

For example:

$$(1+\mathbf{i})(2-2\mathbf{k}) = 2+2\mathbf{i}-2\mathbf{k}-2\mathbf{ik} = 2+2\mathbf{i}+2\mathbf{j}-2\mathbf{k}$$

Just as with complex numbers, the modulus of a quaternion is defined by:

$$|w+x\mathbf{i}+y\mathbf{j}+z\mathbf{k}| = \sqrt{w^2+x^2+y^2+z^2}$$

The quaternions with unit modulus are known as the unit quaternions.

The inverse of a unit quaternion* $\boldsymbol{q}=w+x\mathbf{i}+y\mathbf{j}+z\mathbf{k}$, is $\boldsymbol{q}^{-1}=w-x\mathbf{i}-y\mathbf{j}-z\mathbf{k}$. So we have $\boldsymbol{q}\boldsymbol{q}^{-1}=\boldsymbol{q}^{-1}\boldsymbol{q}=1$.

Rotations may be represented as unit quaternions as follows. Suppose we wish to rotate through an angle θ around the axis defined by the unit vector $(n_x,n_y,n_z)$. The corresponding quaternion $\boldsymbol{q}$ is given by:

$$\boldsymbol{q} = \cos\tfrac{1}{2}\theta + (n_x\mathbf{i}+n_y\mathbf{j}+n_z\mathbf{k})\sin\tfrac{1}{2}\theta$$

The value of quaternions lies in the fact that if we multiply the quaternions for two rotations together then the resulting quaternion represents the resulting rotation. These operations are considerably simpler than those needed to represent rotations using matrices.

More explicitly, given a vector $(\mathbf{v}_x,\mathbf{v}_y,\mathbf{v}_z)$, we apply the rotation via the quaternion $\boldsymbol{q}$ representing it using the following formula:

$$\mathbf{v}'_x\mathbf{i}+\mathbf{v}'_y\mathbf{j}+\mathbf{v}'_z\mathbf{k} = \boldsymbol{q}(\mathbf{v}_x\mathbf{i}+\mathbf{v}_y\mathbf{j}+\mathbf{v}_z\mathbf{k})\boldsymbol{q}^{-1}$$

There is a further property of quaternions requiring a little more discussion. A rotation through 360° is effectively the same as no rotation. However, if we substitute 0° and 360° into the earlier formula for $\boldsymbol{q}$ above, we find that these rotations are represented by +1 and -1 respectively. This means that both +1 and -1 represent the identity transform. This is not a problem though, it simply means that every rotation can be represented by two different quaternions. If we use the above formula for applying a quaternion to a vector, then whichever of the two equivalent quaternions we choose, we will still obtain the same result.

* The inverse of a non-unit quaternion $\boldsymbol{q}$ is $(w-x\mathbf{i}-y\mathbf{j}-z\mathbf{k})/|\boldsymbol{q}|^2$.

# Resource Management

BRender divides all memory allocation according to a set of classes, called memory classes, or resource classes. Each memory class is notionally separate and is useful in distinguishing between different types of memory usage (for allocation policies, debugging, performance analysis, releasing all memory in a class as a whole). The term resource class is used to indicate an advanced use of the memory class, in a structured fashion, where resource blocks (special blocks of memory allocated from the resource class) can be allocated as dependants of another resource block. Similar to C++, this enables destruction of a resource block to automatically delete all its dependent resource blocks automatically. This lightens the application programmer's burden of responsibility. BRender utilises this feature in a lot of the data structures it allocates – **br_actor**$_{76}$ is a good example in the way it may allocate default, type specific data.

Generally, you will not need to explicitly free resource blocks allocated by BRender, even if you overwrite pointers to them, as a separate record is maintained of their existence and what they are attached to. This ensures that when something is freed that any resource blocks attached to it are also freed. The only time attached resources need to be freed sooner is if memory is very limited and there is a disproportionately high turnover of allocations of a dependant compared to its parent's lifetime. In such cases **BrResFree()**$_{52}$ can be used (with caution) to free the dependent immediately, e.g. destroying unwanted models loaded using **BrFmtASCLoad()**$_{248}$.

For simple applications the simple memory allocation functions can be used (see **BrMemAllocate()**$_{56}$), and where structured allocation is required, resource blocks can be allocated from the `BR_MEMORY_APPLICATION` resource class. For more advanced applications requiring greater control, new resource classes can be created (see **br_resource_class**$_{334}$).

There are other features of resource blocks, including the ability to have a call-back function called for each resource block of a particular class.

You will find **BrResStrDup()**$_{50}$ quite useful for specifying identifiers of various data structures.

---

## BrResAllocate()

*Description:* Allocate a new resource block of a given class.

*Declaration:*
```
void* BrResAllocate(void* vparent, br_size_t size,
int res_class)
```

*Arguments:*

**void * vparent**

A pointer to a parental resource block, or `NULL` if it will have no parent.

**br_size_t size**

Size of block in bytes.

**int res_class**

Resource class (See Memory Management, page 54).

*Result:* **void ***

Returns a pointer to the allocated resource block (of at least `size` bytes).

## BrResAdd()

*Description:* Add a resource block as a dependant or child of another.

*Declaration:* **void* BrResAdd(void* vparent, void* vres)**

*Arguments:* **void * vparent**

A non-NULL pointer to the parent resource block.

**void * vres**

A non-NULL pointer to the child resource block (will be removed from any current parent).

*Result:* **void ***

Returns a pointer to the child resource block.

*Remarks:* Once a resource block has been added as a dependant, it can be freed due to its parent being freed.

## BrResStrDup()

*Description:* Duplicate a string.

*Declaration:* **char* BrResStrDup(void* vparent, const char* str)**

*Arguments:* **void * vparent**

A pointer to a parental resource block (or NULL if independent).

**const char * str**

A non-NULL pointer to the source string.

*Effects:* Allocates a resource block (from the BR_MEMORY_STRING resource class) and copies a string into it.

*Result:* **char ***

Returns a pointer to the allocated resource block.

*Remarks:* This is most useful for specifying identifiers for various BRender data structures.

## BrResClass()

*Description:* Determine the class of a given resource block.

*Declaration:* **br_uint_8 BrResClass(void* vres)**

*Arguments:* **void * vres**

A non-NULL pointer to a resource block.

*Result:* **br_uint_8**

Returns the resource class.

## BrResSize()

*Description:* Determine the size of a given resource block.

*Declaration:* **br_uint_32 BrResSize(void* vres)**

*Arguments:* **void * vres**

A non-NULL pointer to a resource block.

*Result:* **br_uint_32**

Returns the size of the resource block in bytes (may not necessarily be the same as that specified upon allocation, but will not be less).

## BrResChildEnum()

*Description:* Enumerate through all dependant resource blocks of a particular resource block.

*Declaration:* **br_uint_32 BrResChildEnum(void* vres, br_resenum_cbfn* callback, void* arg)**

*Arguments:* **void * vres**

A non-NULL pointer to a resource block.

**br_resenum_cbfn * callback**

A non-NULL pointer to a call-back function.

**void * arg**

An optional argument to pass to the call-back function.

*Effects:* Invoke a call-back function for each child of a resource block. The call-back is passed a pointer to each child, and its second argument is an optional pointer (arg) supplied by the user. The call-back itself returns a **br_uint_32**[358] value. The enumeration will halt at any stage if the return value is non-zero.

*Result:* **br_uint_32**

Returns the first non-zero call-back return value, or zero if all children are enumerated.

*Remarks:* This function does not recurse throughout all descendants. If you require such behaviour, you'll need to implement it yourself.

*Example:*

```
br_uint_32 example_callback(void *vres, void *arg)
{br_uint_32 value;
...
   return(value);
}
{  br_uint_32 ev;
   void *rblock;
...
   ev = BrResChildEnum(rblock,&example_callback,NULL);
}
```

## BrResRemove()

*Description:* Remove a resource block from a parent.

*Declaration:* **void* BrResRemove(void* vres)**

*Arguments:* **void * vres**

A non-NULL pointer to the resource block to be removed.

*Result:* **void ***

Returns a pointer to the removed resource block.

*Remarks:* The resource block will become independent, as though it had been allocated specifying NULL as its parent.

## BrResFree()

*Description:* Free a resource block.

*Declaration:* **void BrResFree(void* vres)**

*Arguments:* **void * vres**

A pointer to a resource block (previously allocated using **BrResAllocate()**[49] or **BrResStrDup()**[50]).

*Effects:* Frees the resource block, and any dependent resource blocks it has. If any resource classes have destructors (see **br_resource_class**[334]), they are invoked when appropriate.

*Example:*

```
void BR_CALLBACK example_destructor(void* res, br_uint_8
res_class, br_size_t size)
{
...
}

#define EXAMPLE_CLASS (BR_MEMORY_APPLICATION + 1)

static br_resource_class example={"My
Class",EXAMPLE_CLASS,example_destructor};

{  BrResClassAdd(&example);/* Create resource class */
   ...
   {  void *ptr;
      ptr = BrResAllocate(NULL,1024,EXAMPLE_CLASS);
      ...
      BrResFree(ptr);/* Destructor is invoked */
   }
}
```

*See Also:* **BrResClassAdd()** 335

# Memory Management

## Dynamic Memory

### *Simple Memory Services*

While C compilers typically provide dynamic memory allocation services via the standard C library, to assist platform independence, BRender indirects all its requirements via an allocator data structure, which defines allocation, deallocation, and inquiry functions. By default these will use the standard C library functions (`malloc` and `free`). Note that not all development systems provide any inquiry functions, so in some cases the corresponding, default BRender function will return zero. Given the vagaries of what is meant by 'available memory', the application programmer is expected to implement this function themselves if its behaviour is critical.

The following two functions are provided for efficient initialisation and copying of word aligned memory.

---

## BrBlockFill()

*Description:* Fast fill a block of memory.

*Declaration:* **`void BrBlockFill(void* dest_ptr, int value, int dwords)`**

*Arguments:* **`void * dest_ptr`**

A pointer to the block to be filled.

**`int value`**

Value to fill the block with.

**`int dwords`**

Size of block (number of 32-bit words).

*See Also:* **`BrBlockCopy()`** 54

---

## BrBlockCopy()

*Description:* Copy a block of memory. The source and destination blocks must not overlap.

*Declaration:* **`void BrBlockCopy(void* dest_ptr, const void* src_ptr, int dwords)`**

*Arguments:* **void * dest_ptr**

Destination pointer.

**const void * src_ptr**

Source pointer.

**int dwords**

Size of block to copy (number of 32-bit words).

*See Also:* **BrBlockFill()** 54

---

### *Memory Classes*

For more precise control over its memory usage, BRender distinguishes between its various uses of dynamic memory. Memory classes also map to resource classes, given that a resource class restricts itself to a single memory class. So identifier's of each are the same.

Memory classes are defined by name and number. The following symbols define the number of each BRender memory class:

```
BR_MEMORY_SCRATCH
BR_MEMORY_PIXELMAP
BR_MEMORY_PIXELS
BR_MEMORY_VERTICES
BR_MEMORY_FACES
BR_MEMORY_GROUPS
BR_MEMORY_MODEL
BR_MEMORY_MATERIAL
BR_MEMORY_MATERIAL_INDEX
BR_MEMORY_ACTOR
BR_MEMORY_PREPARED_VERTICES
BR_MEMORY_PREPARED_FACES
BR_MEMORY_LIGHT
BR_MEMORY_CAMERA
BR_MEMORY_BOUNDS
BR_MEMORY_CLIP_PLANE
BR_MEMORY_STRING
BR_MEMORY_REGISTRY
BR_MEMORY_TRANSFORM
BR_MEMORY_RESOURCE_CLASS
BR_MEMORY_FILE
BR_MEMORY_ANCHOR
BR_MEMORY_POOL
BR_MEMORY_RENDER_MATERIAL
BR_MEMORY_DATAFILE
```

The class names are the same bar the `BR_MEMORY` prefix, thus `BR_MEMORY_LIGHT` has an identifier of "`LIGHT`".

The applications programmer may also indirectly request memory using classes between `BR_MEMORY_APPLICATION` and `BR_MEMORY_MAX-1` (inclusive). `BR_MEMORY_APPLICATION`

should be used for general heap requirements (like `malloc()`). Classes `BR_MEMORY_APPLICATION+1` and onwards are reserved for user defined classes (see **`br_resource_class`**[334] and **`BrResClassAdd()`**[335]).

### *Dynamic Memory Services*

When allocating memory or making an inquiry, the memory class needs to be specified. For general purposes use `BR_MEMORY_APPLICATION`.

---

## BrMemInquire()

*Description:* Find the amount of memory available of a given type.

*Declaration:* **`br_size_t BrMemInquire(br_uint_8 type)`**

*Arguments:* **`br_uint_8 type`**

Memory type.

*Result:* **`br_size_t`**

Returns memory available in bytes.

---

## BrMemAllocate()

*Description:* Allocate memory.

*Declaration:* **`void* BrMemAllocate(br_size_t size, br_uint_8 type)`**

*Arguments:* **`br_size_t size`**

Size of memory block to be allocated.

**`br_uint_8 type`**

Memory type.

*Result:* **`void *`**

Returns a pointer to the allocated memory, or NULL is unsuccessful.

---

## BrMemCalloc()

*Description:* Allocate and clear memory.

*Declaration:* **`void* BrMemCalloc(int nelems, br_size_t size, br_uint_8 type)`**

*Arguments:* **`int nelems`**

Number of elements to allocate.

**`br_size_t size`**

Size of each element.

**br_uint_8 type**

Memory type.

*Result:* **void ***

Returns a pointer to the allocated memory, or NULL is unsuccessful.

## BrMemStrDup()

*Description:* Duplicate a string.

*Declaration:* **char* BrMemStrDup(const char* str)**

*Arguments:* **const char * str**

A pointer to the source string.

*Result:* **char ***

Returns a pointer to the new copy of the string.

## BrMemFree()

*Description:* Deallocate memory.

*Declaration:* **void BrMemFree(void* block)**

*Arguments:* **void * block**

A pointer to the block of memory to deallocate.

### *Memory Allocation Handler*

BRender provides the facility for the application to specify their own memory allocation functions, through which all BRender's memory allocation is performed. A static **br_allocator**[102] structure, containing pointers to an allocator, deallocator and inquiry function, is defined and engaged using **BrAllocatorSet()**[103].

The application will typically need to do this for platforms that don't have the standard C library functions (`malloc()` etc.), and in cases where simple memory allocation is not appropriate due to limited memory space or performance.

### *Memory Pools*

Pools are a flexible way of allocating memory from a class, whereas resource classes permit structured allocation. See **br_pool**[306] for more information.

# Filing System Support

## Standard Filing System Services

BRender provides a set of filing system services very similar to those provided in the standard C library. All filing should be performed using these functions, so that on platforms where there is no standard C library or where the filing system is based on an unusual media such as cartridge, calls to load various items will still succeed. In some cases it may be necessary to have some read operations accessing a CD-ROM and some read/write operations accessing a memory card (not that one expects to store much in a memory card).

Note that BRender will recognise an environment variable (in some systems) called BRENDER_PATH, which will be used, if defined, to extend the search for unqualified file names beyond the current directory.

For background information on the following functions, read any standard C library documentation that you may have.

---

## BrFileAttributes()

*Description:* Determine capabilities of the filing system.

*Declaration:* **`br_uint_32 BrFileAttributes(void)`**

*Preconditions:* Filing system handler dependent. BRender's default filing system requires BRender to have completed initialisation.

*Effects:* Interrogates filing system.

*Result:* **`br_uint_32`**

The attributes of the filing system; its capabilities as defined by a combination of the following flag values:

| Flag | Attribute |
|---|---|
| BR_FS_ATTR_READABLE | Filing system can read files |
| BR_FS_ATTR_WRITEABLE | Filing system can write files |
| BR_FS_ATTR_HAS_TEXT | Filing system can interpret ASCII text files |
| BR_FS_ATTR_HAS_BINARY | Filing system can support binary files, i.e. maintains integrity of streams of any combination of bytes (8 bit). |
| BR_FS_ATTR_HAS_ADVANCE | Filing system can directly skip bytes |

---

## BrFileOpenWrite()

*Description:* Open a file for writing, over-writing any existing file of the same name.

*Declaration:* **void* BrFileOpenWrite(const char* name, int mode)**

*Arguments:* **const char * name**

Name to open file as.

**int mode**

Mode in which to open file (`BR_FS_MODE_TEXT` or `BR_FS_MODE_BINARY`). In the default implementation of the filing system (using the standard C library), this is effectively turned into a `"w"` or `"wb"` write mode parameter to `fopen()`.

*Preconditions:* Filing system handler dependent. BRender's default filing system requires BRender to have completed initialisation.

*Effects:* Overwrite or create file using specified `name` and `mode`.

*Result:* **void ***

Return a file handle or `NULL` if the file could not be opened.

*Remarks:* In the default implementation, when a file is opened for writing in text mode, 'end of file' (EOF, Ctrl-Z, 0x1A) characters may have special significance, and 'line feed' (LF, Ctrl-J, 0x0A, '\n') characters may be translated to 'carriage return, line feed' combinations. Consult your Standard C library documentation.

---

# BrFileWrite()

*Description:* Write a block of data to a file.

*Declaration:* **int BrFileWrite(const void* buf, int size, int n, void* f)**

*Arguments:* **const void * buf**

Buffer containing block to be written.

**int size**

Size of each element in block.

**int n**

Maximum number of elements to write.

**void * f**

Valid file handle - as returned by **BrFileOpenWrite()**[58].

*Preconditions:* Filing system handler dependent. BRender's default filing system requires BRender to have completed Initialisation.

*Effects:* Write up to `n` elements from `buf` to the file `f`.

*Result:* **int**

Return the number of complete elements written, which may be less than `n` if an error occurs (such as running out of file space).

*Remarks:* The data written to the file may be affected by the current 'write mode' of the file.

## BrFilePrintf()

*Description:* Write a formatted string to a file.

*Declaration:* **`int BrFilePrintf(void* f, const char* fmt, ...)`**

*Arguments:* **`void * f`**

Valid file handle - as returned by **`BrFileOpenWrite()`**[58].

**`const char * fmt`**

Format string as supplied to `printf()`.

`...`

Any further necessary parameters as would be required in `printf()`.

*Preconditions:* Filing system handler dependent. BRender's default filing system requires BRender to have completed initialisation.

*Effects:* Write the string to the file (effectively using `vsprintf()`).

*Result:* **`int`**

Returns the number of characters written, or a negative value if an error occurs.

*Remarks:* The data written to the file may be affected by the current 'write mode' of the file.

*See Also:* Consult your Standard C library documentation.

## BrFilePutLine()

*Description:* Write a line of text to a file, followed by writing the new-line character (`'\n'`).

*Declaration:* **`void BrFilePutLine(const char* buf, void* f)`**

*Arguments:* **`const char * buf`**

Pointer to zero terminated string containing line of text to be written.

**`void * f`**

Valid file handle - as returned by **`BrFileOpenWrite()`**[58].

*Preconditions:* Filing system handler dependent. BRender's default filing system requires BRender to have completed initialisation.

*Effects:* Write the string to the file. Write the new-line character to the file (`'\n'`).

*Remarks:* The data written to the file may be affected by the current 'write mode' of the file.

*See Also:* Consult your Standard C library documentation.

## BrFilePutChar()

*Description:* Write a single character to a file.

*Declaration:* **`void BrFilePutChar(int c, void* f)`**

*Arguments:* **int c**

Character to write.

**void * f**

Valid file handle - as returned by **BrFileOpenWrite()**[58].

*Preconditions:* Filing system handler dependent. BRender's default filing system requires BRender to have completed initialisation.

*Effects:* Write the character to the file.

*Remarks:* The data written to the file may be affected by the current 'write mode' of the file.

*See Also:* Consult your Standard C library documentation.

---

# BrFileOpenRead()

*Description:* Open a file for read access.

*Declaration:* **void* BrFileOpenRead(const char* name, br_size_t n_magics, br_mode_test_cbfn* mode_test, int* mode_result)**

*Arguments:* **const char * name**

Name of file.

**br_size_t n_magics**

Number of characters required for `mode_test` to determine file type (less than or equal to `BR_MAX_FILE_MAGICS`).

**br_mode_test_cbfn * mode_test**

Call-back function that can be used to determine file type given the first `n_magics` characters of a file. Will not be used if `NULL`.

**int * mode_result**

If this argument is non-`NULL`, the file type (if it could be determined) will be stored at the address pointed to.

*Preconditions:* Filing system handler dependent. BRender's default filing system requires BRender to have completed Initialisation.

*Effects:* Searches for a file called `name`, if no path is specified with the file, looks in the current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined). Having found the file, use `mode_test` (if supplied) to find out if the file is text, binary or unknown. Store the result through `mode_result` (if non-`NULL`). Obtain a handle to the file.

*Result:* **void ***

Return a file handle, or `NULL` if the file could not be opened.

*See Also:* Consult your Standard C library documentation.

## BrFileRead()

*Description:* Read a block of data from a file.

*Declaration:* **int BrFileRead(void* buf, int size, int n, void* f)**

*Arguments:* **void * buf**

Buffer to receive block.

**int size**

Size of each element in block.

**int n**

Maximum number of elements to read.

**void * f**

Valid file handle - as returned by **BrFileOpenRead()**[61].

*Preconditions:* Filing system handler dependent. BRender's default filing system requires BRender to have completed initialisation.

*Effects:* Read up to `n` elements of `size` bytes from the file `f` and store them in `buf`.

*Result:* **br_size_t**

Return the number of complete elements read, which may be less than `n` if the end of file is encountered before all the elements could be read.

*Remarks:* The data read from the file may be affected by the 'write mode' used to write the file.

*See Also:* Consult your Standard C library documentation.

## BrFileGetLine()

*Description:* Read a line of text (excluding terminators) from a file.

*Declaration:* **int BrFileGetLine(char* buf, br_size_t buf_len, void* f)**

*Arguments:* **char * buf**

Buffer to hold text read.

**br_size_t buf_len**

Length of buffer (maximum number of characters to store - including `'\0'`).

**void * f**

Valid file handle - as returned by **BrFileOpenRead()**[61].

*Preconditions:* Filing system handler dependent. BRender's default filing system requires BRender to have completed Initialisation.

*Effects:* Read characters into supplied buffer until `buf_len-1` characters have been read, end of line has been read, or end of file has been reached. If the last character read was `'\n'` it is removed from the buffer.

*Result:* **int**

The number of characters stored in the buffer is returned. If at the end of file upon entry, zero will be returned.

*Remarks:* The data read from the file may be affected by the 'write mode' used to write the file.

*See Also:* Consult your Standard C library documentation.

---

# BrFileGetChar()

*Description:* Read a character from a file.

*Declaration:* **int BrFileGetChar(void* f)**

*Arguments:* **void * f**

Valid file handle - as returned by **BrFileOpenRead()**[61].

*Preconditions:* Filing system handler dependent. BRender's default filing system requires BRender to have completed initialisation.

*Effects:* If the file position is at the end of the file, the file enters the end-of-file state, otherwise a character is read and the file position is advanced.

*Result:* **int**

The character read from the file is returned (as though the character had been cast as `(int)(unsigned char)`). If a character could not be read because the file position was at the end of the file, `BR_EOF` is returned.

*Remarks:* The data read from the file may be affected by the 'write mode' used to write the file.

*See Also:* Consult your Standard C library documentation.

---

# BrFileAdvance()

*Description:* Advance the file pointer a number of bytes through a binary stream.

*Declaration:* **void BrFileAdvance(long int count, void* f)**

*Arguments:* **long int count**

Number of bytes to advance.

**void * f**

Valid file handle - as returned by **BrFileOpenRead()**[61] and **BrFileOpenWrite()**[58].

*Preconditions:* Filing system handler dependent. BRender's default filing system requires BRender to have completed initialisation.

*Effects:* Advance file position by `count` bytes.

*Remarks:* This function may be affected by the 'write mode' used to write the file. If the 'write mode' is `BR_FS_MODE_TEXT`, the accuracy of the file pointer is not guaranteed (unless count is zero).

*See Also:* Consult your Standard C library documentation.

## BrFileEOF()

*Description:* Test a file pointer for end of file.

*Declaration:* **`int BrFileEof(const void* f)`**

*Arguments:* **`const void * f`**

Valid file handle - as returned by **`BrFileOpenRead()`**[61] and **`BrFileOpenWrite()`**[58].

*Preconditions:* Filing system handler dependent. BRender's default filing system requires BRender to have completed initialisation.

*Effects:* None.

*Result:* **`int`**

Returns a non-zero value after the first read operation that attempts to read past the end of the file. It returns 0 if the current position is not end of file.

*Remarks:* This function may be affected by the 'write mode' used to write the file. If the 'write mode' is `BR_FS_MODE_TEXT`, the result may indicate end-of-file upon reaching an EOF character.

*See Also:* Consult your Standard C library documentation.

## BrFileClose()

*Description:* Close a previously opened file.

*Declaration:* **`void BrFileClose(void* f)`**

*Arguments:* **`void * f`**

Valid file handle - as returned by **`BrFileOpenRead()`**[61] and **`BrFileOpenWrite()`**[58].

*Preconditions:* Filing system handler dependent. BRender's default filing system requires BRender to have completed initialisation.

*Effects:* Close file.

*See Also:* Consult your Standard C library documentation.

# Special Filing System Services

BRender allows the application to specify how the various export functions should write data to files. Text mode will write data in ASCII, using text labels and expanding numbers into strings. Text mode

files are primarily used for debugging but can be useful to allow hand editing of input data. Binary mode will cause exported data to be written in binary, i.e. a more compact, unreadable form.

## BrWriteModeSet()

*Description:* Instruct BRender to output data files in text or binary for subsequent 'Save' operations.

*Declaration:* **`int BrWriteModeSet(int mode)`**

*Arguments:* **`int mode`**

Write mode. Either `BR_FS_MODE_TEXT` or `BR_FS_MODE_BINARY`.

*Effects:* Records the mode with which BRender should subsequently open files for writing and how data should be written to them. This will affect export functions such as **`BrModelSave()`** 249.

*Result:* **`int`**

Returns the old write mode.

*Remarks:* The purpose of this function is primarily to indicate to BRender how it should output its data files, whether in compact binary form or in vaguely human readable, ASCII form.

If the application produces binary and text files then it will specify the write mode upon opening a file for writing.

When opening a file for read, it is opened in the same mode as it was written.

# Diagnostic Support

Very, very, very few programs are perfect. The compiler may generate incorrect code*, there may be bugs in the standard C libraries, BRender or the application may be faulty, even the user may upset things through copying the wrong versions of files, say. Certain aspects of the development platform, operating platform, and user configuration may make certain features unavailable. When it comes down to it, you have to expect the unexpected. Fortunately, BRender provides some facilities to assist in dealing with this problem.

The first thing to do is to understand the different situations in which the unexpected can happen (the umbrella term of 'Failure' is used). This is covered in the following 'Classifying Failure' section. What to do when you spot failure is covered by 'Handling Failure' and various relevant techniques are covered in subsequent sections.

## Classifying Failure

### *Build Errors*

Build errors are those caught in the build process by the preprocessor, compiler and linker. Syntax, semantic and symbol definition errors are the sort of errors that come under this description.

BRender does not issue any of these errors.

### *Logic Errors*

These are errors in the logic of the program, when it just doesn't quite behave in the desired way. Some errors are caught by the compiler, others are hopefully caught by observation (from programmer to user testing).

### *Subtle Errors*

These are errors caused by limitations of the program environment, e.g. loss of precision, rounding errors, numeric overflow, invalid algorithms, invalid data, etc. The best way of catching these is to use liberal 'asserts' throughout the program.

### *Unexpected Failure*

The program can go catastrophically wrong, typically because of memory corruption. Examples of causes that might lead to such failure include: invalid arguments, incorrect bounds, incorrect casting, invalid pointers. The only way of catching these is either to double check everything using 'asserts' or to use third party tools that can monitor invalid memory access and dynamic memory use.

---

* Only consider this as the very last resort, when you have explored every possible mode of failure in your own program.

### *Expected Failure*

There are known limitations within most computer environments. Time, resources, features, and user correctness are all things that are limited and sometimes the limit is reached. If a program is going to make any effort to avoid crashing it should at least cater for those occasions when failure can be expected.

BRender provides various reporting facilities for handling failure.

## Handling Failure

The situations in which failure can be caught are detailed below. Unexpected failure is sometimes caught, but usually for a consequential expected failure. Some systems may be able to detect invalid memory access and in some circumstances, it may be appropriate for the application to handle such exceptions.

### *Unexpected Program Failure*

Errors that should not happen, i.e. assertion failures, should cause an abort in the debug build of an application and a fatal error message, but should be ignored in the release. The point about the use of 'assert' is that it states a fact about the state of the program at a particular point. If failure is expected, an assert should not be used, but another handler used instead. Predictably, if an assert would fail in the release, but is properly ignored, it is quite likely that the program will proceed to crash. It is therefore very important to be confident that no assertion can be expected to fail.

### *Probable, Expected Failure*

Those cases where failure is probable and catered for do not warrant any special diagnostic treatment. Examples include: a user specified file not being found, a missing joystick, an unimplemented feature, etc. These are all very much expected failures and are either remedied by the user taking corrective action, or by the program using an alternative. Obviously, the program will need to keep the user informed, but such a failure is not considered an error.

### *Minor, Expected Failure*

Those cases where failure does not significantly affect the program, but is improbable, and possibly surprising, can be considered minor failures. A warning should be issued in the debug build. In the release there will still be some informational evidence of failure, e.g. a greyed-out menu option.

### *Serious, Expected Failure*

An expected failure, but only with a severe remedy available, can be considered a serious failure. An error message can be issued in the debug build, and an information response to the user in the release.

### *Catastrophic, Expected Failure*

Unexpected failure, or expected failure with no remedy available, can be considered catastrophic failure. A fatal error message should be issued in the debug build, followed by abort. In the release

build, there should be an informational response. The likely remedy will be exit to the OS/Menu or restarting the application.

## Debug & Release Builds

BRender supports the convention of only defining 'assert' macros when the `DEBUG` preprocessor symbol is defined. This together with likely differences in compiler options for debugging support, distinguishes the 'debug' build from the 'release' build of a BRender application.

For conditional compilation depending upon debug or release builds, always use the defined state of the `DEBUG` symbol to determine the current build type.

With `DEBUG` defined `BR_ASSERT()` and `BR_VERIFY()` will abort and report failure if their single argument evaluates to zero or `NULL`, without `DEBUG` defined, `BR_ASSERT()` is defined as void, and `BR_VERIFY()` is defined to evaluate its argument. The 'verify' version is typically used to assert the value of an expression, that has a required side effect.

The programmer is expected to define `ASSERT` and `VERIFY` symbols as equivalent to `BR_ASSERT` and `BR_VERIFY`, rather than use `ASSERT` and `VERIFY` directly. This can also be achieved by explicitly using `#include "brassert.h"` in each source file in which this is required.

## Debugging

For debugging an application it is sometimes useful to write progress messages to `stderr` so that when errors are reported there is some contextual information preceding them. Such messages can also display the values of certain variables. The macro `BR_TRACE()` is defined for such a purpose, together with `BR_TRACE0()` to `BR_TRACE6()` which accept `printf()` style arguments for formatted output.

## Reporting Diagnostics

Warnings, Errors and Fatal Errors are reported using the macros `BR_WARNING()`, `BR_FAILURE()`, and `BR_FATAL()`. Each takes a single string argument, and this is output to `stderr`. This should be considered a diagnostic level reporting facility and should only be used for testing purposes. The finished application should take a more polished approach and should tailor its response to each type of failure.

There are also formatted versions of these macros taking multiple arguments. The macro has a single digit suffix indicating the number of arguments (from 0 to 6) in addition to the format string – the arguments are the same as for `printf()`.

Use `BR_WARNING()` to generate a warning or error message, but continue with the program afterwards. Use `BR_FAILURE()` to generate an error message, and not return, but perform a recovery (exit). Use `BR_FATAL()` for severe cases where source level information (source file name and line number) should also be reported.

## Errors Reported by BRender

BRender may utilise the diagnostics itself in some circumstances, e.g. a missing file, insufficient memory, etc.

# Specifying a Diagnostic Handler

A diagnostic handler may be set using **`BrDiagHandlerSet()`**[119]. This may be to report errors more appropriately, e.g. in a dialog box (in a GUI), or to a remote debugging terminal. Or, it could simply be to disable the messages.

The error handler lists pointers to functions handling `BR_TRACE()`, `BR_WARNING()`, `BR_FAILURE()`, and `BR_FATAL()`.

# Miscellaneous

## Input Device Support

BRender has little support for 3D specific input devices, however, the following function may be useful in simulating a 3D input device using a 2D input device such as a mouse.

---

## BrMatrix34RollingBall()

*Description:* This function generates a matrix which provides an intuitive way of controlling the rotation of 3D objects with a mouse or other 2D pointing device. The mouse may be thought of as controlling a horizontal flat surface resting on top of a fixed sphere. As the surface is moved in any direction (but not rotated), so the sphere will rotate.

*Declaration:* **`void BrMatrix34RollingBall(br_matrix34* mat, int dx, int dy, int radius)`**

*Arguments:* **`br_matrix34 * mat`**

A pointer to the destination matrix.

**`int dx,dy`**

The amount the 'top surface' has moved in each direction.

**`int radius`**

The radius of the imaginary ball.

*Effects:* This function calculates the tangent vector (`dx`,`dy`) to the sphere of radius `radius`. in 3D, and uses this to determine the axis of rotation normal to this tangent at the centre of the sphere, and the angle subtended by the tangent vector. From this, a transform matrix is created, describing the rotation.

*Remarks:* The function is expected to be used with frequent samples of movements made with the 2D manipulator, i.e. a movement of 10cm in one go will produce a smaller rotation (no greater than 180°) whereas the same movement sampled at 100 intervals will cumulatively produce a larger rotation (possibly several revolutions). This is unlikely to be a problem in practice.

Note that using the 2D manipulator to describe small circles can rotate the 3D object about its vertical axis.

*Example:*

```
int mouse_x,mouse_y;
br_matrix34 mat;
...
BrMatrix34RollingBall(&mat,-mouse_x,mouse_y,500);
```

*See Also:* The book *Graphics Gems III*, edited by David Kirk, ISBN 0-12-409670-0, Ch.2, Pt.3, 'The Rolling Ball', Andrew J. Hanson, p51.

---

# Platform Specific Support

## All Platforms

Note that the installation guide will have more detailed documentation for each platform.

### *Diagnostic Support*

See **br_diaghandler**[118] for details of providing a platform specific, diagnostic handler.

### *Dynamic Memory Support*

See **br_allocator**[102] for details of providing a platform specific, memory allocation handler.

### *Filing System Support*

See **br_filesystem**[128] for details of providing a platform specific, filing system handler.

### *Machine Word Ordering*

The network order of an n byte word is defined as each byte sorted in order of significance, with the most significant byte at offset 0, and thus the least significant byte at offset n-1. The host order of a word is undefined and varies across platforms. No assumptions should be made about host ordering, and any time anything is done which is order dependent, the word must be first converted to network order. BRender defines the following macros to convert between host and network ordering.

`BrNtoHF(n)` Converts a `float` argument from network order to host order

`BrNtoHL(n)` Converts a `long` argument from network order to host order

`BrNtoHS(n)` Converts a `short` argument from network order to host order

`BrHtoNF(n)` Converts a `float` argument from host order to network order

`BrHtoNL(n)` Converts a `long` argument from host order to network order

`BrHtoNS(n)` Converts a `short` argument from host order to network order

## Games consoles

### *Sony PSX/Sega Saturn*

No documentation is available at the time of writing, except that these platforms are supported.

# DOS

### *MS DOS/PC DOS/DR DOS*

BRender is supported on this platform in conjunction with a 32 bit DOS extender (refer to the DOS installation guide). There is a DOS support API (described in the DOS platform installation guide) covering the following areas:
Setting graphics modes and obtaining a video buffer
Setting the video palette
Obtaining mouse input
Reading the system clock
Keyboard handling
Divide overflow exception handling
Keyboard and Mouse event handling

### *Specific Graphics Hardware*

No documentation on support for specific graphics hardware was available at the time of writing, except that VESA and VGA compliant graphics cards are supported.

# Windows (16 bit)

### *Windows 3.1 & 3.11 (Win32s API)*

Note that BRender is only able to operate effectively on these platforms by using the Win32s API.

BRender is able to render to a device independent pixel map. Displaying the pixel map requires use of a Windows or WinG API function such as BitBlt, StretchDIBits, WinGBitBlt, WinGStretchBlt, etc.

### *Specific Graphics Hardware*

No documentation on support for specific graphics hardware was available at the time of writing.

# Windows (32 bit)

### *Windows 95 & NT (Win32 API)*

BRender is able to render to a device independent pixel map. Displaying the pixel map requires use of a Windows or WinG API function such as BitBlt, StretchDIBits, WinGBitBlt, WinGStretchBlt, etc.

### *Direct Draw, Direct 3D*

No documentation is available at the time of writing, except that BRender will exploit these APIs.

### *Intel/Alpha/MIPS/PowerPC*

No documentation concerning any differences in BRender API or operation across these MPUs is available at the time of writing.

### *Specific Graphics Hardware*

No documentation on support for specific graphics hardware was available at the time of writing.

## Macintosh

### *68k*[*]*, Power Mac*

No platform specific documentation is available at the time of writing except that this platform is supported.

## Other

Other platforms may be supported, but no documentation is available at the time of writing.

* 68020 or better

# 4 Data Structures (Alphabetical Reference)

# br_actor

## The Structure

The basic unit of scene construction. The **br_actor**[76] object is designed to facilitate hierarchical relationships between elements of a scene, particularly in terms of position and orientation.

See The Actor, page 12, in the structured description for further details.

### *The* ***typedef***

(See *actor.h* for precise declaration and ordering)

#### Hierarchical relationships

| | | |
|---|---|---|
| **br_actor *** | **parent** | ***Parent of this Actor*** |
| **br_actor *** | **next** | ***Next sibling of this Actor*** |
| **br_actor **** | **prev** | ***Previous sibling of this Actor*** |
| **br_actor *** | **children** | ***Children of this Actor*** |
| **br_uint_16** | **depth** | ***Depth of this Actor from root of hierarchy*** |

#### Positional relationship

| | | |
|---|---|---|
| **br_transform** | **t** | ***Transform to convert to parent co-ordinates*** |

#### Actor function

| | | |
|---|---|---|
| **br_uint_8** | **type** | ***Actor type*** |
| **void *** | **type_data** | ***Extra, type specific data for this Actor*** |
| **br_model *** | **model** | ***Model data for model Actors*** |
| **br_material *** | **material** | ***Material data for model Actors*** |
| **br_uint_8** | **render_style** | ***Rendering style for model Actors*** |

#### Supplementary

| | | |
|---|---|---|
| **char *** | **identifier** | ***String to identify actor*** |
| **void *** | **user** | ***User data (application dependent)*** |

### *Related Functions*

#### Scene Rendering

See **BrZbSceneRenderBegin()**[34], **BrZbSceneRenderAdd()**[36], **BrZbSceneRender()**[34], **Br[Zb|Zs]ModelRender()**[253|254]

# Members

### *Hierarchical Relationships*

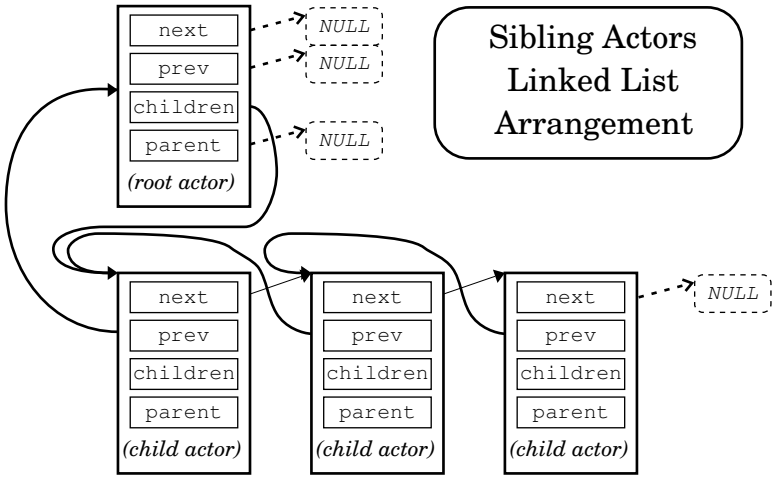


*Diagram showing linked list of sibling actors for three children of a root.*

## br_actor * parent

A pointer to the actor's parent actor in an actor hierarchy. If the actor has no parent, i.e. is the root actor of its hierarchy, `parent` is `NULL`.

## br_actor * next

A pointer to the next sibling actor in the linked list of sibling actors (parent's children). If the actor has no siblings or is the last sibling actor, `next` is `NULL`.

## br_actor ** prev

A pointer to the previous sibling actor's `next` member in the linked list of sibling actors (parent's children). If the first sibling, `prev` will point to its parent actor's `children` member. If the root actor, `prev` is `NULL`. Given that the `next` member is the first member of **`br_actor`**$_{76}$, `prev` can be cast to a pointer to the previous actor (where `prev` points to a `next` member).

## br_actor * children

A pointer to the first actor in a linked list of child actors. If the actor has no children, `children` is `NULL`.

## br_uint_16 depth

The depth of the actor from the root of the hierarchy. `depth` is zero for the root actor and increases by one each generation (indirection through `children`).

### Common Features of `next`, `prev`, `children`, `depth`

These members are maintained by BRender, and only ever changed during hierarchy construction (See **`BrActorAdd()`**[82], **`BrActorRemove()`**[83], **`BrActorRelink()`**[83]). Although they should never be directly modified, direct use for traversal purposes is permitted.

### *Positional Relationship*

## br_transform t

This defines the transform to apply to co-ordinates of this actor (e.g. of models), to convert them to co-ordinates in its parent's co-ordinate system. This is the primary way of defining relative model positions and orientations. Any valid transform may be used.

This member is read when rendering and computing actor/actor transforms. It can be modified at any time, although it should be appreciated that this may have unpredictable effects if done during render call-back functions.

See **`BrActorToActorMatrix34()`**[84] for information on obtaining transforms between actors other than parent and child.

### Example

Remember that each actor can be considered to have its own (right handed) co-ordinate system. Consider a point (1,2,3), of a model vertex say. Let's suppose the actor has a transform that just translates by a vector of (1,0,0). In applying this transform we obtain the point (2,2,3), and these are now co-ordinates in the parent's co-ordinate's system. Now let's suppose the actor has a transform that also rotates by 90° about the x-axis (+y topples toward us, +z) and is then followed by the translation of (1,0,0). If we applied this transform we'd obtain the point (2,-3,2).

If you wished to perform this using BRender functions, you might first use **`BrTransformToMatrix34()`**[353], and then **`BrMatrix34ApplyP()`**[194].

### *Actor Function*

## br_uint_8 type

This member defines the type of function required of this actor. The actor's `type` should contain a value defined by one of the symbols described in the following table.

| Actor Type Symbol | Behaviour |
|---|---|
| BR_ACTOR_NONE | Reference actor – no special behaviour. May be used to define frames of reference, actor groups, intermediate transforms, inheritable properties, and temporarily disable an alternative behaviour. |
| BR_ACTOR_MODEL | Renderable model – render the specified model in the specified style, possibly using the default material |
| BR_ACTOR_LIGHT | Light source – affects rendering of models using materials that are lit. |
| BR_ACTOR_CAMERA | Camera – used to define view point and perspective for rendering output image. |
| BR_ACTOR_BOUNDS | A cuboid bounding box, application defined within the actor's co-ordinate space.<br>Box completely off screen: rendering of all descendants is disabled<br>Box partially on/off screen: descendants subject to 'on screen' check<br>Box completely on screen: descendants subject to 'on screen' check |
| BR_ACTOR_BOUNDS_CORRECT | A cuboid bounding box, application defined within the actor's co-ordinate space.<br>Box completely off screen: rendering of all descendants is disabled<br>Box partially on/off screen: descendants subject to 'on screen' check<br>Box completely on screen: descendants not subject to 'on screen' check[a] |
| BR_ACTOR_CLIP_PLANE | The four vector pointed to by type_data defines a three vector unit normal to a plane whose distance from the origin is represented by the fourth element.<br>Descendants are clipped against the plane, with the side defined by the normal being 'in scene'. |

a. Rendering of a model that is off screen is undefined, and should be considered a fatal error

Although set at initialisation, this member can be changed outside rendering at any time (as long as the other members are set appropriately, and, if a Light or Clip Plane, it has first been disabled (See **BrLightDisable()**$_{88}$ and **BrClipPlaneDisable()**$_{89}$)). During rendering (within call-back functions) change is not recommended – Actors should not be changed to or from Lights or Clip Planes, and the particular Camera supplied for the rendering should not have its type changed.

## void * type_data

This member is used to refer to additional, type specific data. It should point to data according to the contents of type as shown in the following table:

| Actor type | Contents of type_data |
|---|---|
| BR_ACTOR_NONE | NULL (or application defined) |
| BR_ACTOR_MODEL | NULL (or application defined) |
| BR_ACTOR_LIGHT | Pointer to an instance of **br_light**$_{147}$ |
| BR_ACTOR_CAMERA | Pointer to an instance of **br_camera**$_{110}$ |
| BR_ACTOR_BOUNDS | Pointer to an instance of **br_bounds**$_{108}$ |
| BR_ACTOR_BOUNDS_CORRECT | Pointer to an instance of **br_bounds**$_{108}$ |
| BR_ACTOR_CLIP_PLANE | Pointer to an instance of **br_vector4**$_{376}$ |

The type_data member should not be NULL except for BR_ACTOR_NONE or BR_ACTOR_MODEL.

This member can be changed at any time, though not recommended during rendering.

Experienced C programmers will be familiar with the technique of supplying pointers to structures that are actually embodied within others, for the purpose of accessing attached application specific data*. This technique is still a workable way of extending the amount of data within BRender data structures, particularly here†. For example, it may be desired to introduce rotating lights. The corresponding details could be appended to the **br_light**147 structure as demonstrated in this example:

```
typedef struct      /* My light, which can be used instead of
   br_light */
{  br_light light;  /* The standard BRender light specification */
   unsigned type;   /* My extended light type */
   br_scalar speed; /* Various properties */
   br_vector3 axis;
} my_light;
```

## br_model * model

The model defines the geometry and any specific face characteristics to be rendered in this actor's co-ordinate space.

This member points to model information, which is used by this actor if it is a model actor, and descendent model actors that inherit it. If NULL, model information is obtained (when required by a model actor) from the previous ancestor (parent) that supplied it. If no ancestor supplies it, a default **br_model**232 data structure is used which defines a cube (this is for diagnostic purposes only).

Note that an actor does not have to be a model actor in order for it to define a model to be inherited by a descendent model actor.

## br_material * material

The material is used for a model's faces that don't specify a material.

This member points to material information, which is used by this actor if it is a model actor, and descendent model actors that inherit it. If NULL, material information is obtained (when required by a model actor) from the previous ancestor (parent) that supplied it. If no ancestor supplies it, a default **br_material**154 data structure is used which defines a flat-shaded grey (this is for diagnostic purposes only). Out of interest, it is currently defined as follows:‡

---

* For C++ programmers, this is embraced by the language – any derived class may be supplied where a base class is expected.

† Though BRender will of course, not be aware of any attached data (so will not save it in **BrActorSave()**, say).

‡ Yet subject to change

```
{  "default",
   BR_COLOUR_RGB(255,255,255), /* colour */
   255,                        /* opacity */
   BR_UFRACTION(0.10),         /* Indexed ka */
   BR_UFRACTION(0.70),         /* kd */
   BR_UFRACTION(0.0),          /* ks */
   BR_SCALAR(20),              /* power */
   BR_MATF_LIGHT,              /* flags */
   {{    BR_VECTOR2(1,0),      /* map transform */
         BR_VECTOR2(0,1),
         BR_VECTOR2(0,0),
   }},
   0,63,                       /* index base/range*/
};
```

Note that an actor does not have to be a model actor in order for it to define a material to be inherited by a descendent model actor.

## br_uint_8 render_style

This member determines the style of rendering, which is used by this actor if it is a model actor. If the NULL style (BR_RSTYLE_DEFAULT) is specified, the rendering style is obtained (when required by a model actor) from the first ancestor (from the root) that defines a non-NULL style. If no ancestor defines one, the model is rendered as though BR_RSTYLE_FACES had been specified.

This is subtly different from normal inheritance, in that non-default styles set by intervening actors have no effect. It is designed this way to facilitate the use of an attribute primarily intended for highlighting (obviously not one concerned with realism).

The render_style member should be set to a value defined by one of the following (enum) symbols:

| Rendering Style Symbol | Rendering Effect |
|---|---|
| BR_RSTYLE_DEFAULT | Uses first non-default rendering style defined in this branch of the hierarchy (renders faces if no non-default style defined). |
| BR_RSTYLE_NONE | Does not render this actor or any of its descendants. |
| BR_RSTYLE_POINTS | Renders only the points at each vertex of a face using the model's material. |
| BR_RSTYLE_EDGES | Renders only the points along each edge of a face using the model's material.(see **br_face**125, flags) |
| BR_RSTYLE_FACES | Renders all the points across rendered faces using the model's material. |
| BR_RSTYLE_BOUNDING_POINTS | Renders only the points at each vertex of the model's bounding box using the actor's material |
| BR_RSTYLE_BOUNDING_EDGES | Renders only the points at along edge of the model's bounding box using the actor's material |
| BR_RSTYLE_BOUNDING_FACES | Renders the faces of the model's bounding box using the actor's material |

br_actor

*Supplementary*

### char * identifier

Pointer to unique, zero terminated, character string (or NULL if not required). Can be used as a handle to retrieve a pointer to the actor, given only an ancestral actor. Not intended for intensive use. Typically used to collect pointers to actors loaded using **BrActorLoad()** 98 and **BrActorLoadMany()** 100. Also ideal for diagnostic purposes.

A non-unique string can be supplied, but which of a set of actors having the same string will be matched by search functions (See **BrActorSearch()** 94), is undefined. Also in consideration of searching, it is not recommended that non-alphabetic characters are used, especially Slash ('/'), Asterisk ('*'), and Query ('?'), which are used for pattern matching.

This member can be modified by the programmer at any time.

If identifier is set by **BrActorLoad()** 98 or **BrActorLoadMany()** 100 it will have been constructed using **BrResStrDup()** 49.

### void* user

This member may be used by the application for its own purposes. It is initialised to NULL upon allocation, and not accessed by BRender thereafter.

## Operations

*Hierarchical Relationships*

---

# BrActorAdd()

*Description:* Add an actor hierarchy as a child of a given parent.

*Declaration:* **br_actor* BrActorAdd(br_actor* parent, br_actor* a)**

*Arguments:* **br_actor * parent**

A non-NULL pointer to the parental actor.

**br_actor * a**

A non-NULL pointer to its new child.

*Preconditions:* Between **BrBegin()** 10 & **BrEnd()** 11. New child must be root of its hierarchy, i.e. must not still be child of another parent.

*Effects:* Parent's children member may be modified. Added hierarchy will be linked into linked list of parent's children. Members next and prev of some of parent's children and of added hierarchy will be modified. The parent member of the added hierarchy will be modified. All depth members of added hierarchy will be updated.

*Result:* **br_actor ***

The new child pointer `a` is returned as supplied (for convenience).

*Remarks:* Do not attempt to add an actor at more than one place in a hierarchy at a time – BRender currently only supports simple hierarchical actor structures, not directed acyclic (or cyclic) graph structures. Note that in spite of this restriction, models on the other hand, may be simultaneously referred to by any number of actors.

*See Also:* **BrActorRemove()** 83, **BrActorRelink()** 83

---

# BrActorRemove()

*Description:* Remove an actor hierarchy from its parent.

*Declaration:* **br_actor* BrActorRemove(br_actor* a)**

*Arguments:* **br_actor * a**

A non-`NULL` pointer to the hierarchy to remove.

*Preconditions:* Between **BrBegin()** 10 & **BrEnd()** 11. Actor to remove must be a child.

*Effects:* Parent's `children` member may be modified. Removed hierarchy will be unlinked from linked list of parent's children. Members `next` and `prev` of some of parent's children and of removed hierarchy will be modified. The `parent` member of the removed hierarchy will be modified. All `depth` members of removed hierarchy will be updated.

*Result:* **br_actor ***

The pointer to the removed hierarchy is returned as supplied (for convenience).

*Remarks:* Note that a Light or Clip plane actor must be disabled before it is removed.

*See Also:* **BrActorAdd()** 82, **BrActorRelink()** 83

---

# BrActorRelink()

*Description:* Move an actor in a hierarchy, but preserve its apparent world transformation by manipulating its own transformation as necessary.

*Declaration:* **void BrActorRelink(br_actor* parent, br_actor* a)**

*Arguments:* **br_actor * parent**

A non-`NULL` pointer to the new parent.

**br_actor * a**

A non-`NULL` pointer to the actor to move.

*Preconditions:* Between **BrBegin()** 10 & **BrEnd()** 11. Both supplied actors are in the same hierarchy.

*Effects:* Equivalent to a call of **BrActorRemove(**`a`**)** 83 followed by **BrActorAdd(**`parent,a`**)** 82, except that the moved actor's transform is modified to reproduce the effects of the original transform.

*Remarsk:* Note that a light or clip plane actor must be disabled before it is relinked.

*See Also:* **BrActorAdd()**[82], **BrActorRemove()**[83]

---

*Positional Relationship*

---

# BrActorToActorMatrix34()

*Description:* Accumulate the transformations between one actor and another, representing the result as a matrix.

*Declaration:*
```
br_uint_8 BrActorToActorMatrix34(br_matrix34* m,
const br_actor* a, const br_actor* b)
```

*Arguments:* **br_matrix34 * m**

A non-NULL pointer to the destination matrix, into which will be placed the transform.

**const br_actor * a**

A non-NULL pointer to the actor from whose co-ordinate space co-ordinates are to be transformed.

**const br_actor * b**

A non-NULL pointer to the actor into whose co-ordinate space co-ordinates are to be transformed.

*Preconditions:* Between **BrBegin()**[10] & **BrEnd()**[11]. Both actors must be in the same hierarchy.

*Effects:* Will calculate the transform required to transform co-ordinates in the co-ordinate space of a, into the co-ordinate space of b.

*Result:* **br_uint_8**

The type of the accumulated transformation, e.g. BR_TRANSFORM_MATRIX34 (see **br_transform**[349]).

---

# BrActorToBounds()

*Description:* Compute an actor hierarchy's bounding box, encompassing the bounding box of all models (including descendants).

*Declaration:*
```
br_bounds* BrActorToBounds(br_bounds* b, const br_actor* ap)
```

*Arguments:* **br_bounds * b**

Non-NULL pointer to resulting bounding box.

**const br_actor * ap**

Non-NULL pointer to actor whose bounding box is required.

*Preconditions:* Between **BrBegin()**[10] & **BrEnd()**[11].

*Effects:* Obtains smallest bounding box (in this actor's co-ordinate space) that will contain any model or descendant model.

*Result:* **br_bounds ***

The bounding box result pointer `b` is returned as supplied (for convenience).

*Remarks:* The default model is used in cases where a model actor would inherit from an ancestor of the supplied actor.

*See Also:* **BrBoundsToMatrix34()** 109, **BrScenePick3D()** 83, **br_bounds** 108, **br_model** 232.

---

# BrScenePick3D()

*Description:* Traverse an actor hierarchy and invoke a call-back function for each model actor whose bounds intersect a given bounds in a given reference actor's space. If the call-back returns a non-zero value, traversal halts.

*Declaration:*
```
int BrScenePick3D(br_actor* world,
const br_actor* reference, const br_bounds* bounds,
br_pick3d_cbfn* callback, void* arg)
```

*Arguments:* **br_actor * world**

A pointer to the root of a world hierarchy.

**const br_actor * reference**

A pointer to the reference actor.

**const br_bounds * bounds**

A pointer to a bounds structure.

**br_pick3d_cbfn * callback**

A pointer to a pick-3D call-back function.

**void * arg**

An optional argument to pass to the call-back function.

*Result:* **int**

If the call-back returns a non-zero value and traversal halts, that value is returned.

Otherwise, zero is returned.

*See Also:* **BrScenePick2D()** 86

---

# BrActorToScreenMatrix4()

*Description:* Accumulate the transformations between an actor and the screen, representing the result as a matrix.

*Declaration:*
```
void BrActorToScreenMatrix4(br_matrix4* m,
const br_actor* a, const br_actor* camera)
```

*Arguments:* **br_matrix4 * m**

A non-NULL pointer to the destination matrix to receive the transform between homogenous co-ordinates in the actor's co-ordinate space into the homogenous screen space.

**const br_actor * a**

A non-NULL pointer to an actor.

**const br_actor * camera**

A non-NULL pointer to a camera actor.

*Remarks:* The function **BrMatrix4ApplyP()** [219] is typically used with this function to convert co-ordinates in an actor's co-ordinate space into homogenous screen space (assuming a centred projection). Note though that the resultant 4-vector is a set of homogenous co-ordinates and thus the x, y and z values will need to be divided by the w component.

See Homogenous screen space, page 23.

*See Also:* **BrMatrix4Perspective()** [228].

---

# BrScenePick2D()

*Description:* Traverse a world hierarchy, picking model actors from the scene by casting a ray through a given viewport pixel attached to a given camera. A call-back is invoked for each actor whose bounds intersect the ray. If the call-back returns a non-zero value, traversal halts.

*Declaration:*
```
int BrScenePick2D(br_actor* world, const br_actor* camera,
const br_pixelmap* viewport, int pick_x, int pick_y,
br_pick2d_cbfn* callback, void* arg)
```

*Arguments:* **br_actor * world**

A pointer to the root of a world hierarchy.

**const br_actor * camera**

A pointer to a camera actor.

**const br_pixelmap * viewport**

A pointer to the viewport through which the pick ray passes.

**int pick_x, pick_y**

Co-ordinates of viewport pixel through which the pick ray passes.

**br_pick2d_cbfn * callback**

A pointer to a pick-2D call-back function.

**void * arg**

An optional argument to pass to the call-back function.

*Result:* **int**

If the call-back returns a non-zero value and traversal halts, that value is returned. Otherwise, zero is returned.

*See Also:* **BrScenePick3D()** 84, **BrModelPick2D()** 235

---

*Actor Function*

## BrEnvironmentSet()

*Description:* This function is relevant to model actors using environment mapping (set in their materials' `flags` member by using `BR_MATF_ENVIRONMENT_I` or `BR_MATF_ENVIRONMENT_L`).

It set a new environment anchor for such model actors. By default, the reflective effect produced by an environment map on the surface of a model appears to rotate with the model itself. If this is unsatisfactory, the map can be anchored to an actor. If the actor is the root of a world hierarchy, then reflection effects will appear more realistic.

*Declaration:* **br_actor* BrEnvironmentSet(br_actor* a)**

*Arguments:* **br_actor * a**

A pointer to an actor to which all environment maps should be anchored. If `NULL`, environment maps are not anchored, but rotate with models.

*Result:* **br_actor ***

Returns a pointer to the old environment anchor.

*Remarks:* Environment actors must be part of the rendered scene (a descendant of the effective root). If rendering a scene to produce a reflection environment map in another scene, then (unless the reflected scene also involves environment maps) the function should be called with `NULL` before the reflected scene is rendered, and then called with an appropriate anchor actor (that is part of the latter scene, in which the environment map is used).

---

## BrLightEnable()

*Description:* Enable a light actor's effect as a light source within the scene. Light actors only affect the lighting of a scene if they are in the enabled state before rendering.

*Declaration:* **void BrLightEnable(br_actor* l)**

*Arguments:* **br_actor * l**

A non-`NULL` pointer to a light actor.

*Remarks:* By default, light actors are disabled.

Light actors must be in the disabled state before they are removed from, relinked within, or added to a hierarchy. They must also be disabled before they are freed (or a parent is).

For optimum performance, disable all lights known to have insignificant effect upon the scene. Some platforms are limited in the number simultaneously enabled lights that they can support.

*See Also:* **BrLightDisable()** 88

---

# BrLightDisable()

*Description:* Disable a light actor's effect as a light source within the scene. Light actors only affect the lighting of a scene if they are in the enabled state before rendering.

*Declaration:* **void BrLightDisable(br_actor* l)**

*Arguments:* **br_actor * l**

A non-NULL pointer to a light actor.

*Remarks:* By default, light actors are disabled.

Light actors must be in the disabled state before they are removed from, relinked within, or added to a hierarchy. They must also be disabled before they are freed (or a parent is).

For optimum performance, disable all lights known to have insignificant effect upon the scene. Some platforms are limited in the number of simultaneously enabled lights that they can support.

*See Also:* **BrLightEnable()** 88

---

# BrClipPlaneEnable()

*Description:* Enable a clip plane. Such clip plane actors affect the contents of a scene (as long as they are enabled before rendering).

*Declaration:* **void BrClipPlaneEnable(br_actor* cp)**

*Arguments:* **br_actor * cp**

A non-NULL pointer to a clip plane actor.

*Remarks:* By default, clip planes are disabled.

Clip planes must be in the disabled state before they are removed from, relinked within, or added to a hierarchy. They must also be disabled before they are freed (or a parent is).

## BrClipPlaneDisable()

*Description:* Disable a clip plane.

*Declaration:* **void BrClipPlaneDisable(br_actor* cp)**

*Arguments:* **br_actor * cp**

A non-NULL pointer to a clip plane actor.

*Remarks:* By default, clip planes are disabled.

Clip planes must be in the disabled state before they are removed from, relinked within, or added to a hierarchy. They must also be disabled before they are freed (or a parent is).

## Copy/Assign

While it may be sensible to copy some members, it is generally not sensible to copy the actor as a whole.

## Access & Maintenance

Given that the order in which a hierarchy is rendered is not defined, the benefits of modifying members during rendering are doubtful. The structure of the actor hierarchy should not be modified during rendering.

An actor may be indirectly accessed by BRender library functions if they are called while the actor is in a hierarchy that is thus modified or traversed (through rendering or otherwise). Consequently, particular care must be taken if modifying the actor while such indirect accesses may be taking place (especially within rendering call-back functions).

The only functions that indirectly modify an actor's public members are **BrActorAdd()** 82, **BrActorRemove()** 83, **BrActorRelink()** 83 and **BrActorFree()** 92, and these only affect some or all of the parent, next, prev, children and depth members.

If using the Z-Sort renderer it is likely that order tables will be need to be assigned to model actors on an individual basis. The following two functions are provided to specify and access an actor's order table.

No maintenance is required, except to ensure that members are valid. Hierarchical changes (Additions, Removals) are effected immediately (given that these are only performed by BRender functions). All ramifications of other changes will be effected each rendering.

See **br_model** 232 and **br_material** 154 for their maintenance requirements.

## BrZsActorOrderTableSet()

*Description:* Given a model actor, set the order table into which its primitives will be sorted.

*Declaration:* **br_order_table* BrZsActorOrderTableSet(br_actor* actor, br_order_table* order_table)**

*Arguments:* **br_actor * actor**

A non-NULL pointer to a model actor.

**br_order_table * order_table**

A pointer to an order table. NULL if the actor is to inherit an order table (the default condition).

*Effects:* Assigns the specified order table to the actor, replacing the current one if any. Be very careful, if assigning an order table to more than one model actor (whether explicitly or by inheritance). In such cases judicious use of the order table flags, such as BR_ORDER_TABLE_NEW_BOUNDS, may be required within a custom model call-back function.

*Result:* **br_order_table ***

Returns order_table as supplied (for convenience).

*Remarks:* Only useful to model actors involved during rendering by the Z-Sort renderer.

*See Also:* **BrZsActorOrderTableGet()** 90

---

# BrZsActorOrderTableGet()

*Description:* Given a model actor, obtain a pointer to the currently set order table (into which its primitives will be sorted).

*Declaration:* **br_order_table* BrZsActorOrderTableGet(const br_actor* actor)**

*Arguments:* **const br_actor * actor**

A non-NULL pointer to a model actor.

*Result:* **br_order_table ***

Returns the order table currently assigned to the specified actor. NULL is returned if no order table is explicitly assigned, i.e. an order table is inherited.

*Remarks:* Only useful to model actors involved during rendering by the Z-Sort renderer.

*See Also:* **BrZsActorOrderTableSet()** 90

---

## Referencing & Lifetime

References are maintained to enabled Light or Clip Plane actors even if they are subsequently detached from a hierarchy that is then rendered. Thus it is important to disable such actors before removing them.

Actors must be maintained while they are part of an actor hierarchy, especially ones that will be rendered.

If the actor was allocated using **BrActorAllocate()** 92 it will be freed by **BrEnd()** 11 (if it hadn't been freed previously – as it should have been).

## Initialisation

The actor is automatically initialised by **BrActorAllocate()** 92, however, if you allocate it yourself, you should either use `calloc(,sizeof(br_actor))` or `memset(,0,sizeof(br_actor))`, and ensure the following is done:
Set `parent`, `next`, `prev`, `children` to `NULL`
Set `depth` to zero
Set `t` to the identity transform
Set `type` to any valid actor type
Set `model` to `NULL`
Set `material` to `NULL`
Set `render_style` be `BR_RSTYLE_DEFAULT`
Set `type_data` to point to a valid data structure according to the type.
Set `identifier` to `NULL`

Refer to the respective description of each member for further details.

## Construction & Destruction

Actors should be constructed using **BrActorAllocate()** 92 and destroyed using **BrActorFree()** 92. Although actors can be constructed any other way (as long as they observe the access & lifetime requirements), they should be detached (See **BrActorRemove()** 83) from a hierarchy before any ancestor (parent) is freed using **BrActorFree()** 92. This is because **BrActorFree()** 92 will attempt to release storage for all of an actor's descendants – whether or not they were allocated by **BrActorAllocate()** 92. In general, actors should be destroyed strictly using the complementary method of construction.

Note that **BrActorLoad()** 98 and **BrActorLoadMany()** 100 will effectively call **BrActorAllocate()** 92 for each actor they import.

---

# BrActorAllocate()

*Description:* Allocate a new actor.

*Declaration:*
```
br_actor* BrActorAllocate(br_uint_8 actor_type,
void* type_data)
```

*Arguments:* **br_uint_8 actor_type**

Defines the type member of the actor allocated (See the type member of **br_actor**$_{76}$ for a list of acceptable types). It is also used to determine suitable type specific data if NULL is supplied for the type_data argument.

**void * type_data**

A pointer to optional, additional, type specific data used to determine the type_data member of the actor allocated. (See the type_data member of **br_actor**$_{76}$ to determine the type of data structure that should be referenced). NULL may be supplied in all cases.

*Preconditions:* Between **BrBegin()**$_{10}$ & **BrEnd()**$_{11}$.

Memory can be allocated.

*Effects:* A **br_actor**$_{76}$ data structure is allocated and initialised (See **br_actor**$_{76}$, Initialisation). Its type member is initialised with the value of actor_type. Where the actor requires type specific data, and NULL has been supplied, the type_data member is initialised with a pointer to an appropriate, freshly allocated data structure containing default values.

*Result:* **br_actor ***

A pointer to the new **br_actor**$_{76}$ data structure.

*Remarks:* If this function allocates type specific data, it is allocated and attached to the actor using **BrResAllocate()**$_{48}$, so will be freed with the actor even if the actor's type_data member is subsequently changed. Until the actor is freed, the data may be freely accessed.

*Example:*

```
br_actor * t_pActor=BrActorAllocate(BR_ACTOR_MODEL,NULL);
```

*See Also:* **BrActorFree()**$_{92}$

---

# BrActorFree()

*Description:* Free an actor and all its descendants (children) if any.

*Declaration:* **void BrActorFree(br_actor* a)**

*Arguments:* **br_actor * a**

A pointer to an actor previously allocated using **BrActorAllocate()**$_{92}$. NULL is not acceptable. If a Light, or Clip Plane actor, it should have first been disabled. The actor should have first been detached from any hierarchy by using **BrActorRemove()**$_{83}$.

*Preconditions:* Between **BrBegin()**$_{10}$ & **BrEnd()**$_{11}$.

*Effects:* Effectively applies **BrActorRemove()**$_{83}$ and **BrActorFree()**$_{92}$ to each child of the actor. Calls **BrResFree()**$_{51}$ to release storage for the actor and any attached data (e.g. type specific data created by default).

*Remarks:* Not recommended to be applied to an actor during rendering of the hierarchy that it is part of.

Ensure that any references to the actor or its children will not be used subsequent to this call. Remember that a light or clip plane should be disabled before it is removed and freed.

*Example:*

```
br_actor * t_pActor=BrActorAllocate(BR_ACTOR_MODEL,NULL);
BrActorFree(t_pActor);
```

*See Also:* **BrActorAllocate()** 92

**BrResFree()** 51

---

# Supplementary

---

## BrActorEnum()

*Description:* Enumerates an actor's children, calling a user supplied call-back function for each child actor.

*Declaration:* **br_uint_32 BrActorEnum(br_actor* parent, br_actor_enum_cbfn* callback, void* arg)**

*Arguments:* **br_actor * parent**

A pointer to the actor whose children are to be enumerated.

**br_actor_enum_cbfn * callback**

A pointer to the call-back function to be called for each child actor.

**void * arg**

The argument to pass to the call-back function (use `NULL` if not required).

*Preconditions:* Between **BrBegin()** 10 & **BrEnd()** 11.

*Effects:* For each child of `parent`, `callback` is invoked supplied with a pointer to the child and the `arg` pointer. If `callback` returns a non-zero result, the **BrActorEnum()** 93 function will immediately return with the same result, otherwise the enumeration continues until all children have been enumerated.

*Result:* **br_uint_32**

The result is zero, or the first and only non-zero result returned by `callback`.

*Remarks:* An entire actor hierarchy can be enumerated simply by having the call-back function invoke **BrActorEnum()** 93 itself.

*Example:*

```
br_uint_32 BR_CALLBACK CountDescendantsCB(br_actor* a,void* n)
{  ++*(int*)n;
   return BrActorEnum(a,CountDescendantsCB,n);
```

```
}

int CountDescendants(br_actor* a)
{  int N=0;
   BrActorEnum(a,CountDescendantsCB,&N);
   return N;
}
```

*See Also:* **br_actor_enum_cbfn**[101]

---

# BrActorSearch()

*Description:* Traverse an actor's descendants, searching for an actor of a particular generation with a lineage and identifier matching a specified wild carded string. To find actors anywhere within a hierarchy will require the additional use of **BrActorEnum()**[93].

*Declaration:* **br_actor* BrActorSearch(br_actor* root, const char* pattern)**

*Arguments:* **br_actor * root**

A pointer to an actor.

**const char * pattern**

Zero terminated character string containing the search pattern. This specifies the generation of actor required by separating descendants with slash characters ('/'), e.g. "/<Child>/<Grand-child>/<Great grand-child>/etc.". Identifiers of actors along the lineage may be matched exactly or in combination with the wild-card characters '*' (match any number of any characters) and '?' (match any single character). The pattern is fully permuted, e.g. "b*an?s" will match "bananas", "banks", and "bandstands".

*Preconditions:* Between **BrBegin()**[10] & **BrEnd()**[11].

*Effects:* Compares the actor's descendants' identifiers with the respective component of the search pattern. Terminates as soon as the complete lineage matches, or all possible matching lineages have been explored.

*Result:* **br_actor ***

A pointer to the first actor that was found whose lineage matched the search pattern, or NULL if there was no match.

*Remarks:* Note that the search pattern's first generation will be compared with the identifiers of the children of root – the identifier of root is not involved in the pattern matching process.

Where more than one match is possible, it is not recommended that any reliance be placed on which match is found first. No assumption should be made regarding how the actor hierarchy is traversed, e.g. whether branch by branch or generation by generation.

To find an actor irrespective of its lineage, use this function in combination with **BrActorEnum()**[93].

*Example:* Given a root actor with identifier "Root", children "Child1" and "Child2", and child "Child1" with grand-children "Grand-son1" and "Grand-daughter1", and child "Child2" with grand-children "Grand-son2" and "Grand-daughter2":

"Child?/Grand*1" would match either "Grand-son1" or "Grand-daughter1".

"Child1" would match "Child1".

"*/*son?" would match either "Grand-son1" or "Grand-son2".

*See Also:* **BrActorSearchMany()** 95

**BrActorEnum()** 93

---

# BrActorSearchMany()

*Description:* Traverse an actor's descendants, searching for actors of a particular generation with a lineage and identifier matching a specified wild carded string.

*Declaration:*
```
br_uint_32 BrActorSearchMany(br_actor* root,
const char* pattern, br_actor** actors, int max)
```

*Arguments:* **br_actor * root**

A pointer to an actor.

**const char * pattern**

Zero terminated character string containing the search pattern. This specifies the generation of actor required by separating descendants with slash characters ('/'), e.g. "/<Child>/<Grand-child>/<Great grand-child>/etc.". Identifiers of actors along the lineage may be matched exactly or in combination with the wild-card characters '*' (match any number of any characters) and '?' (match any single character). The pattern is fully permuted, e.g. "b*an?s" will match "bananas", "banks", and "bandstands".

**br_actor ** actors**

A pointer to a series of actor pointers. Storage must have been allocated (though not necessarily initialised) for at least max actor pointers.

**int max**

The maximum number of matched actors to store at actors.

*Preconditions:* Between **BrBegin()** 10 & **BrEnd()** 11.

*Effects:* Compares the actor's descendants' identifiers with the respective component of the search pattern. Terminates as soon as the number of complete lineage matches has reached max, or all possible matching lineages have been explored.

*Result:* **br_uint_32**

The number of matched actors stored at actors. This will be between zero and max (inclusive).

*Remarks:* The same as **BrActorSearch()** 94 except that it allows matches of more than one actor.

Note that the search pattern's first generation will be compared with the identifiers of the children of `root` – the identifier of `root` is not involved in the pattern matching process.

To find actors irrespective of their lineage, use this function in combination with **BrActorEnum()** 93.

*Example:*

```
br_actor* t_aActorWheels[10];
int t_n;

t_n=BrActorSearchMany(pActorLorry,"Chassis/*Axle/
   Wheel?",t_aActorWheels,10);

ASSERT(!(t_n%2));/* Even number of wheels */
```

*See Also:* **BrActorSearch()** 94, **BrActorEnum()** 93

## Import & Export

Actor hierarchies can be saved and loaded in their entirety. This can be useful for setting up scenes, or importing arbitrary, complex systems.

# BrActorFileCount()

*Description:* Locate a given file and count the number of actor hierarchies stored in it.

*Declaration:* **br_uint_32 BrActorFileCount(const char* filename, br_uint_16* num)**

*Arguments:* **const char * filename**

Name of the file previously created by **BrActorSave()** 97 or **BrActorSaveMany()** 99 (See Filing System Support, page 57).

**br_uint_16 * num**

Pointer to destination into which the number of actor hierarchies will be stored. If `NULL`, the file will still be located and the success result returned, but no count will be made.

*Preconditions:* Between **BrBegin()** 10 & **BrEnd()** 11. Filing system available.

*Effects:* Searches for `filename`, if no path specified with file looks in current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined).

Counts the number of actor hierarchies stored in the file. If the file is not an actor file, the function is still successful, but would obviously produce a zero count of actors.

*Result:* **br_uint_32**

Returns zero if successful (the file was found), non-zero otherwise.

*See Also:* **BrActorLoad()** 98, **BrActorLoadMany()** 100

## BrActorSave()

*Description:* Saves an actor and its descendants as a hierarchy.

*Declaration:*
```
br_uint_32 BrActorSave(const char* filename,
const br_actor* actor)
```

*Arguments:* **const char * filename**

The name of the file under which the hierarchy should be saved (See Filing System Support, page 57).

**const br_actor * actor**

A pointer to the effective root actor of the hierarchy to be saved.

*Preconditions:* Between **BrBegin()** 10 & **BrEnd()** 11. Available filing system. Success depends on sufficient file space.

*Effects:* If successful, the actor is written to a file* along with the `identifier` of its material (if non-`NULL`), the `identifier` of its model (if non-`NULL`), and any type specific data. This same process is then applied to each child in turn.

*Result:* **br_uint_32**

The result is one if the hierarchy was saved successfully, and zero if not.

*Remarks:* If the function fails (returns zero), the filing system can be considered returned to the state it was in just before the call, e.g. no file created.

The hierarchical relationship between actors is recorded implicitly, no use is made of the actor's `parent`, `next`, `prev`, `children`, or `identifier` members.

Note, that model and material data is not saved with actors, therefore this should be saved separately if required. It will need to be available to **BrModelFind()** 244 and **BrMaterialFind()** 165 (in the registry) before the actor hierarchy is loaded.

*Example:*
```
...
return BrActorSave("MyScene",pActorScene);
```

*See Also:* **BrActorLoad()** 98, **BrActorSaveMany()** 99, **BrWriteModeSet()** 64

## BrActorLoad()

*Description:* Load a hierarchy of actors from a file.

* Any existing file of the same name is overwritten.

*Declaration:* **br_actor* BrActorLoad(const char* filename)**

*Arguments:* **const char * filename**

Name of the file previously created by **BrActorSave()** 97 or **BrActorSaveMany()** 99 (loads the first hierarchy) (See Filing System Support, page 57).

*Preconditions:* Between **BrBegin()** 10 & **BrEnd()** 11. Filing system available. Memory can be allocated.

*Effects:* Searches for `filename`, if no path specified with file looks in current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined).

Allocates a root actor, reads members (`type`, `identifier`, `render_style`, `t`) in the order they were saved in, attempts to set `model` with **BrModelFind()** 244 using the model identifier read, attempts to set `material` with **BrMaterialFind()** 165 using the material identifier read, allocates and sets the type specific actor data. Any child actors are read using this same process (thus recursively), and then added to this actor using **BrActorAdd()** 82. If a model or material was not found, the member is left at `NULL`.

*Result:* **br_actor ***

A pointer to the root actor of the loaded hierarchy, or `NULL` if unsuccessful.

*Remarks:* Ensure that necessary models and materials are loaded and added into the registry before calling this function.

*Example:*

```
br_actor* t_pMyScene=BrActorLoad("MyScene");
```

*See Also:* **BrActorSave()** 97, **BrActorLoadMany()** 100

---

# BrActorSaveMany()

*Description:* Saves a series of actors and their descendants as hierarchies.

*Declaration:* **br_uint_32 BrActorSaveMany(const char* filename, const br_actor* const * actors, br_uint_16 num)**

*Arguments:* **char * filename**

The name of the file under which the hierarchies should be saved (See Filing System Support, page 57).

**const br_actor * const * actors**

A pointer to a series of `num` pointers to effective root actors of the hierarchies to be saved. The actors do not necessarily need to be from unrelated hierarchies.

**br_uint_16 num**

Number of hierarchies to save.

*Preconditions:* Between **BrBegin()** 10 & **BrEnd()** 11. Available filing system. Success depends on sufficient file space.

*Effects:* If successful for every one of the actor hierarchies, each is saved as described in **BrActorSave()** 97, except that all are written to the same file*.

*Result:* **br_uint_32**

The function returns the number of complete actor hierarchies that were written to the file. This will be between zero and `num` (inclusive). Success is indicated by the return value being equal to `num`.

*Remarks:* If the function fails (returns a value less than `num`), the filing system can be considered returned to the state it was in just before the call, e.g. no file created.

Note, that model and material data is not saved with actors, therefore this should be saved separately if required. It will need to be available to **BrModelFind()** 244 and **BrMaterialFind()** 165 (in the registry) before any of the actor hierarchies are loaded.

*Example:*

```
br_actor* apActorsInMyScene[N];
...
if (BrActorSaveMany("MyScene",apActorsInMyScene,(br_uint_16)N)<N)
...
```

*See Also:* **BrActorLoadMany()** 100, **BrActorSave()** 97, **BrWriteModeSet()** 64

---

# BrActorLoadMany()

*Description:* Load one or more hierarchies of actors from a file.

*Declaration:* **br_uint_32 BrActorLoadMany(const char* filename, br_actor** actors, br_uint_16 num)**

*Arguments:* **const char * filename**

Name of the file previously created by **BrActorSaveMany()** 99 or **BrActorSave()** 97. See Filing System Support for details of file naming.

**br_actor ** actors**

A pointer to a series of actor pointers. Storage must have been allocated (though not necessarily initialised) for at least `num` actor pointers. See **BrActorFileCount()** 97 for a way of determining this value from a file.

**br_uint_16 num**

The maximum number of matched actors to store at `actors`.

*Preconditions:* Between **BrBegin()** 10 & **BrEnd()** 11. Filing system available. Memory can be allocated.

---

* Any existing file of the same name is overwritten.

*Effects:* Searches for `filename`, if no path specified with file looks in current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined).

Stores a pointer to the root actor of each loaded hierarchy into the respective element of `actors`. See **`BrActorLoad()`** 98 for details of how each hierarchy is loaded.

*Result:* **`br_uint_32`**

The number of actor hierarchies loaded successfully. Will be between zero and `num` (inclusive).

*Remarks:* Ensure that necessary models and materials are loaded and added into the registry before calling this function.

*Example:*

```
br_actor* t_aScenes[5];
br_uint_32 t_u32;

t_u32=BrActorLoadMany("MyScenes",t_aScenes,5);
```

*See Also:* **`BrActorSaveMany()`** 99, **`BrActorLoad()`** 98

# br_actor_enum_cbfn

## The Call-Back Function

This type defines a function, supplied to **BrActorEnum()**[93], and to be called by it for an actor's children.

### *The* ***typedef***

(See *fwproto.h* for a precise declaration)

**br_uint_32 br_actor_enum_cbfn(br_actor*, void*)***Enumerator*

## Specification

---

## CBFnActorEnum()

| | |
|---|---|
| *Description:* | An application defined call-back function accepting an actor and an application supplied argument (as supplied to **BrActorEnum()**[93]). |
| *Declaration:* | **br_uint_32 BR_CALLBACK CBFnActorEnum(br_actor* a, void* arg)** |
| *Arguments:* | **br_actor * a**<br>One of the child actors enumerated by **BrActorEnum()**[93].<br>**void * arg**<br>The argument supplied to **BrActorEnum()**[93]. |
| *Preconditions:* | BRender has completed initialisation. |
| *Effects:* | Application defined. Avoid adding or removing children to the parent actor within this function. |
| *Result:* | **br_uint_32**<br>Any non-zero value will terminate the enumeration and be returned by **BrActorEnum()**[93]. Return zero to continue the enumeration. |
| *See Also:* | **BrActorEnum()**[93], **BrActorSearch()**[94]. |

---

# br_allocator

## The Structure

This structure represents the definition of a memory allocation system or memory handler. All BRender's memory allocation is provided by just three functions, which can be specified by the programmer. This is essential in cases where the standard C library functions, which BRender's handler uses by default, are not available. Sometimes, more sophisticated behaviour is desired, or diagnostic features are needed.

### *The* ***typedef***

(See *brmem.h* for precise declaration and ordering)

Handling Functions

| | | |
|---|---|---|
| **brmem_allocate_cbfn *** | **allocate** | ***Memory allocation function*** |
| **brmem_free_cbfn *** | **free** | ***Memory deallocation function*** |
| **brmem_inquire_cbfn *** | **inquire** | ***Memory inquiry function*** |

Supplementary

| | | |
|---|---|---|
| **char *** | **identifier** | ***Allocator name*** |

## Members

### *Handling Functions*

#### brmem_allocate_cbfn * allocate

This is a pointer to the memory allocation function (See **brmem_allocate_cbfn**394). This is called by **BrMemAllocate()**55.

#### brmem_free_cbfn * free

This is a pointer to the memory deallocation function (See **brmem_free_cbfn**396). This is called by **BrMemFree()**56.

#### brmem_inquire_cbfn * inquire

This is a pointer to the memory inquiry function (See **brmem_inquire_cbfn**397). This is called by **BrMemInquire()**55.

*Supplementary*

`char * identifier`

Pointer to unique, zero terminated, character string (or `NULL` if not required). A string constant is recommended.

## Operations

---

### BrAllocatorSet()

*Description:* Install a new set of memory allocation/deallocation functions.

*Declaration:* **`const br_allocator* BrAllocatorSet(const br_allocator* newal )`**

*Arguments:* **`const br_allocator * newal`**

A pointer to an instance of a **`br_allocator`**[102] structure.

*Result:* **`const br_allocator *`**

Returns a pointer to the old **`br_allocator`**[102] structure.

---

## Copy/Assign

Beware of copying the structure if `identifier` has been allocated from the heap.

## Access & Maintenance

While the structure is the current allocator (most recently passed to **`BrAllocatorSet()`**[103]) the members should not be changed. No maintenance required.

## Referencing & Lifetime

The structure must remain valid until a new allocator is passed to **`BrAllocatorSet()`**[103].

## Initialisation

The members should be set before the structure is passed to **`BrAllocatorSet()`**[103].

## Construction & Destruction

The structure should ideally be statically constructed.

If constructed by 'malloc' ensure it is destroyed by 'free', i.e. the memory allocation handler that provided the storage should be the one to reclaim it. Be careful if macros have been used to redefine malloc to use BRender's allocator. Generally, be aware of any constraints one handler may place on memory allocated by another handler.

## Supplementary

The identifier may be used to determine the current handler in use.

# br_angle

## The Integral Type

This is BRender's own angle representation. One complete revolution corresponds to the entire range of this data type, and therefore repeated rotations will 'wrap around'correctly. Use whenever an angle needs to be represented.

Also see **br_scalar**$_{342}$ and **br_euler**$_{121}$.

### *The* ***typedef***

(See *angles.h* for a precise declaration)

**br_fixed_luf** **br_angle** ***Angle type***

### *Related Functions*

#### Degree/Angle conversion

**br_scalar BrDegreeToRadian(br_scalar s)**

Converts `s` from angular units of degrees into radians.

**br_scalar BrRadianToDegree(br_scalar s)**

Converts `s` from angular units of radians into degrees.

## Arithmetic

An angle may be negated.

Two angles may be added and subtracted.

An angle may be multiplied or divided by a standard integral type.

The following trigonometric macros may be used:

`BR_SIN(a)` Return Sine of **br_angle**$_{105}$ `a` as a **br_scalar**$_{342}$.

`BR_COS(a)` Return Cosine of **br_angle**$_{105}$ `a` as a **br_scalar**$_{342}$.

`BR_ASIN(s)` Return angle whose Sine is **br_scalar**$_{342}$ `s`, as a **br_angle**$_{105}$.

`BR_ACOS(s)` Return angle whose Cosine is **br_scalar**$_{342}$ `s`, as a **br_angle**$_{105}$.

`BR_ATAN2(y,x)` Return angle whose Tangent is **br_scalar**$_{342}$ `y/x`, as a **br_angle**$_{105}$.

`BR_ATAN2FAST(y,x)` Faster, lower precision version of `BR_ATAN2(y,x)`.

See **br_fixed_ls**$_{132}$ for details of fixed point functions used by these macros.

## Comparison

All standard comparison operators may be applied between two angles. An angle may be compared with zero without conversion.

## Conversion

### *From Numeric Constants*

**BR_ANGLE_DEG(x)**

Converts `x` from a number of degrees to **br_angle**[105].

**BR_ANGLE_RAD(x)**

Converts `x` from a number of radians to **br_angle**[105].

**PI**

If not already defined, `PI` is defined as `3.14159265358979323846`.

### *From Integral Types*

To convert from integral types, use **br_scalar**[342] as an intermediary.

### *From **br_scalar**[342]*

**br_angle** *BrScalarToAngle*(**br_scalar s**)

Converts `s` from a fractional revolution (0..1).

**br_angle** *BrDegreeToAngle*(**br_scalar d**)

Converts `d` from an angle in degrees.

**br_angle** *BrRadianToAngle*(**br_scalar r**)

Converts `r` from an angle in radians.

### *To Integral Types*

To convert to integral types, use **br_scalar**[342] as an intermediary.

### *To **br_scalar**[342]*

**br_scalar** BrAngleToScalar(**br_angle a**)

Converts `a` to a fractional revolution (0..1).

**br_scalar** BrAngleToDegree(**br_angle a**)

Converts `a` to an angle in degrees.

**br_scalar** BrAngleToRadian(**br_angle a**)

Converts `a` to an angle in radians.

## Copy/Assign

Only assign zero or angles. Use conversions in all other cases.

# br_boolean

## The Integral Type

BRender's boolean type. Use this type where only a boolean value is intended.

### *The* ***typedef***

(See *compiler.h* for precise declaration)

| | | |
|---|---|---|
| **int** | **br_boolean** | ***Normal type of boolean C expression*** |

## Arithmetic

All standard C arithmetic operators are valid as with the standard C boolean type, i.e `int`.

## Comparison

All standard C comparison operators are valid as with the standard C boolean type, i.e `int`.

## Conversion

### *From True*

`BR_TRUE`

Use as a value representing boolean true.

### *From False*

`BR_FALSE`

Use as a value representing boolean true.

### *From any integral type*

`BR_BOOLEAN(exp)`

Converts `exp` from any integral or pointer value into a boolean value.

## Copy/Assign

Freely assign (**br_boolean** to **br_boolean**, that is). Use wherever a boolean value is intended.

# br_bounds

## The Structure

A data structure describing an axis-aligned bounding box for a model or hierarchy of actors.

### *The* ***typedef***

(See *vector.h* for precise declaration and ordering)

| | | |
|---|---|---|
| **br_vector3** | **min** | ***Minimal corner*** |
| **br_vector3** | **max** | ***Maximal corner*** |

### *Related Functions*

Scene Modelling

See **BrScenePick3D()** 84.

Image Support

See **BrOnScreenCheck()** 254.

### *Related Structures*

Note that this structure is not used by the render bounds call-back function (it represents its bounds in pixel co-ordinates).

Scene Modelling

See **br_actor** 76, **br_model** 232.

## Members

### br_vector3 min

Co-ordinates of the minimal corner of the bounding box.

### br_vector3 max

Co-ordinates of the maximal corner of the bounding box.

## Operations

### BrBoundsToMatrix34()

*Description:* Find the transform that maps a 2 unit cube centred at the origin to the given bounding box.

*Declaration:* **br_matrix34* BrBoundsToMatrix34(br_matrix34* mat, const br_bounds* bounds)**

*Arguments:* **br_matrix34 * mat**

A non-NULL pointer to the destination matrix, into which will be placed the transform.

**const br_bounds * bounds**

A non-NULL pointer to the bounding box.

*Effects:* Will calculate the transform required to transform a 2 unit cube centred at the origin to the given bounding box.

*Result:* **br_matrix34 ***

The pointer to the destination matrix, mat, returned for convenience.

*Remarks:* Typically used to facilitate 3D cursors, e.g. a child model actor with a cursor model whose co-ordinates are based upon a 2 unit cube, can use this function to define its transform.

*See Also:* **br_actor**[76]

## Copy/Assign

Copy by assignment.

## Initialisation

Use vector initialisers, e.g.

```
br_bounds my_bounds={BR_VECTOR3(-1.,-2.,-3.),
   BR_VECTOR3(1.,2.,3.)};
```

# br_camera

## The Structure

BRender's camera data structure. See Camera Actors.

### *The* ***typedef***

(See *camera.h* for precise declaration and ordering)

#### Behaviour

| | | |
|---|---|---|
| **br_uint_8** | **type** | ***Camera type*** |

#### Parameters

| | | |
|---|---|---|
| **br_angle** | **field_of_view** | ***Field of view*** |
| **br_scalar** | **hither_z** | ***Front of view volume*** |
| **br_scalar** | **yon_z** | ***Back of view volume*** |
| **br_scalar** | **aspect** | ***Aspect ratio (x/y)*** |
| **br_scalar** | **width** | ***Width of projection surface*** |
| **br_scalar** | **height** | ***Height of projection surface*** |

#### Supplementary

| | | |
|---|---|---|
| **char *** | **identifier** | ***Camera name*** |
| **void *** | **user** | ***User data (application dependent)*** |

### *Related Functions*

#### Image Support

See also **BrActorToScreenMatrix4()** 86, **BrMatrix4Perspective()** 228.

### *Related Structures*

#### Scene Modelling

See **br_actor** 76.

# Members

*Behaviour*

## br_uint_8 type

Type of camera. Which can be one of the following:

| Camera type | Behaviour |
|---|---|
| BR_CAMERA_PARALLEL | A parallel camera. Object size is independent of its distance from the camera. |
| BR_CAMERA_PERSPECTIVE | A standard perspective camera. |

*Parameters*

## br_angle field_of_view

Field of view, i.e. the angle subtended at the camera between the top and bottom of the view volume (pyramid). Applies only to perspective cameras. The value should be greater than zero and less than 180°. Note than this is the full angle, i.e. not the half angle, between the view z axis and top (or bottom) of view volume.

## br_scalar hither_z

Distance of front of view volume from camera along negative z axis, i.e. -hither_z in view co-ordinates. The value should be greater than zero.

## br_scalar yon_z

Distance of back of view volume from camera along negative z axis, i.e. -yon_z in view co-ordinates. The value should be greater than hither_z.

## br_scalar aspect

Scaling factor for width of viewing volume. For perspective cameras, ±1 in y view ordinates is mapped to the height of the output pixel map, and ±aspect in x view ordinates is mapped to the width of the output pixel map. For parallel cameras, the height of the view volume is mapped to the height of the output pixel map, whereas width×aspect is mapped to the width of the output pixel map.

| Correct Aspect |
| --- |
| If you want to maintain correctly proportioned images on a screen, then the ratio between the sides of the physical image should be the same as that of the view volume. Therefore, for perspective cameras, `aspect` should be calculated as the ratio between the physical dimensions of the output pixel map, i.e. its physical width divided by its physical height (both of which may be computed directly in terms of pixel map resolution if pixels are square). For parallel cameras, `aspect` would have to be defined such that `width×aspect÷height` is the same as the ratio between the physical dimensions of the output pixel map – this is effectively modifying the width of the view volume to neatly fit the pixel map. In this case, `aspect` is relatively redundant as it might as well be left at unity with just `width` needing to be changed. |

### br_scalar width

Width of view volume (rectangular prism) in world co-ordinates (before `aspect` is applied) – the actual width is `width×aspect`. Applies only to parallel cameras.

### br_scalar height

Height of view volume (rectangular prism) in world co-ordinates. Applies only to parallel cameras.

*Supplementary*

### char * identifier

Pointer to unique, zero terminated, character string (or `NULL` if not required).

If `identifier` is set by **`BrActorLoad()`**[98] or **`BrActorLoadMany()`**[100] it will have been constructed using **`BrResStrDup()`**[49].

### void * user

This member may be used by the application for its own purposes. It is initialised to `NULL` upon allocation (if allocated by BRender), and not accessed by BRender thereafter.

## Copy/Assign

The camera should copied using structure assignment.

## Access & Maintenance

Modification is not recommended during rendering, especially if applied to the viewing camera. The members should always accord with the camera type. The aspect may need to be updated in line with changing image dimensions.

## Referencing & Lifetime

If not constructed by **BrActorAllocate()**[92] it may be multiply referenced (given a sufficient lifetime). The camera structure should be maintained at least as long as it is referenced by any actor.

## Initialisation

Only members applicable to the camera type need be initialised.

E.g. `BrActorAllocate(BR_ACTOR_CAMERA,NULL)` currently initialises a **br_camera**[110] structure as follows (after performing `memset(,0,sizeof(br_camera))`):

```
br_camera* camera;
...
camera->type=BR_CAMERA_PERSPECTIVE;
camera->field_of_view=BR_ANGLE_DEG(45);
camera->hither_z=BR_SCALAR(0.1);
camera->yon_z=BR_SCALAR(10.0);
camera->aspect=BR_SCALAR(1.0);
```

## Construction & Destruction

The camera structure can either be constructed/destroyed by the application, or by supplying `NULL` to **BrActorAllocate()**[92].

## Supplementary

Cameras created by **BrActorAllocate()**[92] are allocated from the `BR_MEMORY_CAMERA ("CAMERA")` resource class, and hence can benefit from the resource class supplementary functions.

## Import & Export

Cameras are imported and exported with actors that specify them. See **BrActorLoad()**[98] and **BrActorSave()**[97].

# br_colour

## The Integral Type

BRender's native 32 bit colour representation. Red, Green and Blue each have 8 bits, leaving a further 8 bits for an optional alpha value (typically representing opacity).

Blue occupies bits 0 to 7, Green occupies bits 8 to 15, Red occupies bits 16 to 23, and Alpha occupies bits 24 to 31.

Note that in some situations, a colour of zero is considered transparent (See **br_material**[154] and **br_pixelmap**[277]).

### *The **typedef***

(See *colour.h* for precise declaration and ordering)

**unsigned long int** **br_colour** ***Packed RGBA values***

## Arithmetic

Binary operations (And &, Or |, Not ~, XOr ^) may sometimes be useful. However, all operands should still be colours, and if necessary, constructed using BR_COLOUR_RGB().

## Comparison

Only equality comparisons are valid between colours.

## Conversion

### *From individual component values*

**BR_COLOUR_RGB(r,g,b)**

Returns a **br_colour**[114] given three 8-bit colour components.

**BR_COLOUR_RGBA(r,g,b,a)**

Returns a **br_colour**[114] given three 8-bit colour components and an 8-bit alpha component.

### *To component values*

**BR_RED(c)**

Returns the Red component of a colour (0..255).

**BR_GRN(c)**

Returns the Green component of a colour (0..255).

**BR_BLU(c)**

Returns the Blue component of a colour (0..255).

**BR_ALPHA(c)**

Returns the Alpha component of a colour (0..255).

## Copy/Assign

Only assign colours.

# br_diag_failure_cbfn

## The Call-Back Function

This type defines a call-back function, primarily intended for the `failure` member of the **br_diaghandler**[118] structure.

### *The* ***typedef***

(See *brdiag.h* for a precise declaration)

```
void                    br_diag_failure_cbfn(const char *)
```
*Handle failure and message*

## Specification

---

## CBFnDiagFailure()

| | |
|---|---|
| *Description:* | An application defined call-back function that is called upon a serious, unexpected error or failure, with a descriptive message. Remember, a *release* version of your software should never fail. |
| *Declaration:* | **void BR_CALLBACK CBFnDiagFailure(const char* message)** |
| *Arguments:* | **const char * message** |
| | Pointer to zero terminated character string describing the failure. |
| *Preconditions:* | The diagnostic handler has been setup (**BrDiagHandlerSet()**[119]). BRender has not necessarily completed initialisation. In an unknown state. A failure has occurred. BRender is not expecting the function to return. |
| *Effects:* | Behaviour is up to the application, but the following procedure can be taken as a suggestion. |
| | Set a failure condition flag (to detect escalation). Optionally, immediately inform user of failure. Re-establish a known state. Perform diagnostics of hardware and software, inform user of any diagnosed faults (optionally, also of original failure). End failure condition, and resume (do not return). |
| *Remarks:* | Avoid allocating memory in your failure call-back function, or you may need special care in handling out-of-memory conditions. BRender's default failure call-back function does not allocate memory. |
| *Example:* | See *stddiag.c* for examples of diagnostic handler functions. |
| *See Also:* | ***CBFnDiagWarning()***[117]. |

---

# br_diag_warning_cbfn

## The Call-Back Function

This type defines a call-back function, primarily intended for the `warning` member of the **br_diaghandler**[118] structure.

### *The* ***typedef***

(See *brdiag.h* for a precise declaration)

```
void                    br_diag_warning_cbfn(const char *)
```
*Handle warning and message*

## Specification

---

## CBFnDiagWarning()

| | |
|---|---|
| *Description:* | An application defined call-back function that is called upon a non-serious, unexpected failure, with a message describing the warning. |
| *Declaration:* | **void BR_CALLBACK CBFnDiagWarning(const char* message)** |
| *Arguments:* | **const char * message**<br>Pointer to zero terminated character string describing the warning. |
| *Preconditions:* | The diagnostic handler has been setup (**BrDiagHandlerSet()**[119]). BRender has not necessarily completed initialisation. In a recoverable state. |
| *Effects:* | Optionally inform user of warning. Return. |
| *Remarks:* | This function is primarily intended for debugging and testing purposes. BRender does not currently generate warnings. |
| *Example:* | See *stddiag.c* for examples of diagnostic handler functions. |
| *See Also:* | ***CBFnDiagFailure()***[116]. |

---

# br_diaghandler

## The Structure

This structure represents the definition of a diagnostic handler. All BRender's diagnostics are handled by just two functions, which can be specified by the programmer. This is essential in cases where `stdout` and `stderr` (as used by the standard C library functions) are not available (or not suitable).

### *The* ***typedef***

(See *`brdiag.h`* for precise declaration and ordering)

Handling Functions

| | | |
|---|---|---|
| **`br_diag_warning_cbfn *`** | **`warning`** | ***Warning delivery function*** |
| **`br_diag_failure_cbfn *`** | **`failure`** | ***Failure delivery function*** |

Supplementary

| | | |
|---|---|---|
| **`char *`** | **`identifier`** | ***Diagnostic handler name*** |

## Members

### *Handling Functions*

#### br_diag_warning_cbfn * warning

This is a pointer to the warning delivery function (See **`br_diag_warning_cbfn`**$_{117}$). This is invoked by the warning macros (See Diagnostic Support, page 65).

#### br_diag_failure_cbfn * failure

This is a pointer to the failure delivery function (See **`br_diag_failure_cbfn`**$_{116}$). This is invoked by the failure, fatal and assertion macros (See Diagnostic Support).

### *Supplementary*

#### char * identifier

Pointer to unique, zero terminated, character string (or `NULL` if not required). A string constant is recommended.

## Operations

---

# BrDiagHandlerSet()

*Description:* Install a new diagnostic handler.

*Declaration:* **br_diaghandler* BrDiagHandlerSet(br_diaghandler* newdh)**

*Arguments:* **br_diaghandler * newdh**

A pointer to an instance of a **br_diaghandler**[118] structure (should really be static). If NULL the default diagnostic handler will be used (uses the standard C I/O library).

*Result:* **br_diaghandler ***

Returns a pointer to the old diagnostic handler (possibly NULL). This may be used to pass diagnostics on, if desired.

*Remarks:* The diagnostic handler may be specified at any suitable time. **BrBegin()**[10] will specify a default diagnostic handler if no handler is currently defined.

Note that diagnostics generated without a diagnostic handler present will only be caught in the debug build.

If not **BrBegin()**[10], This should be the first BRender function called.

You should cater for the circumstance arising, if in the process of handling diagnostics, your diagnostic handler may call functions that may themselves generate diagnostics.

---

## Copy/Assign

Beware of copying the structure if identifier has been allocated from the heap.

## Access & Maintenance

While the structure is the current handler (most recently passed to **BrDiagHandlerSet()**[119]) the members should not be changed. No maintenance required.

## Referencing & Lifetime

The structure must remain valid at least until a new handler is passed to **BrDiagHandlerSet()**[119], and may need to remain valid longer if a newer diagnostic handler uses it to pass diagnostics on.

## Initialisation

The members should be set before the structure is passed to **BrDiagHandlerSet()**[119].

`br_diaghandler`

## Construction & Destruction

The structure should ideally be statically constructed.

## Supplementary

The `identifier` may be used to determine the current handler in use.

# br_euler

## The Structure

Euler* angles can be used to represent the orientation of an object. Three separate rotations are applied in turn, in a specific order. Euler angles can be either static or relative. With relative Euler angles, the rotations are performed around axes relative to the rotations already performed. With static Euler angles, the rotations are performed around static axes.

### *The* ***typedef***

(See *angles.h* for precise declaration and ordering)

| | | |
|---|---|---|
| **br_angle** | **a** | ***First angle*** |
| **br_angle** | **b** | ***Second angle*** |
| **br_angle** | **c** | ***Third angle*** |
| **br_uint_8** | **order** | ***Rotation order (static/relative)*** |

### *Related Functions*

Maths

See **BrMatrix34ToEuler()** 207, **BrMatrix4ToEuler()** 224, **BrQuatToEuler()** 323.

### *Related Structures*

Maths

See **br_transform**349.

## Members

### br_angle a

First angle of rotation.

### br_angle b

Second angle of rotation.

### br_angle c

Third angle of rotation.

---

* Pronounced 'oiler' as in *boiler*.

br_euler

## br_uint_8 order

The order in which the angles of rotation are applied. The following symbols define the set of values which may be used. The first three letters (of the suffix to BR_EULER) indicate the order in which rotations occur about which axes. The last letter indicates whether the rotations are static or relative.

```
BR_EULER_XYZ_S
BR_EULER_XYX_S
BR_EULER_XZY_S
BR_EULER_XZX_S
BR_EULER_YZX_S
BR_EULER_YZY_S
BR_EULER_YXZ_S
BR_EULER_YXY_S
BR_EULER_ZXY_S
BR_EULER_ZXZ_S
BR_EULER_ZYX_S
BR_EULER_ZYZ_S
BR_EULER_ZYX_R
BR_EULER_XYX_R
BR_EULER_YZX_R
BR_EULER_XZX_R
BR_EULER_XZY_R
BR_EULER_YZY_R
BR_EULER_ZXY_R
BR_EULER_YXY_R
BR_EULER_YXZ_R
BR_EULER_ZXZ_R
BR_EULER_XYZ_R
BR_EULER_ZYZ_R
```

### For Example

BR_EULER_XYZ_S is equivalent to the matrix transform:

$$\mathbf{R}_{x(a)}\mathbf{R}_{y(b)}\mathbf{R}_{z(c)}$$

BR_EULER_XZY_R is equivalent to the matrix transform:

$$\mathbf{R}_{y(c)}\mathbf{R}_{z(b)}\mathbf{R}_{x(a)}$$

See **br_matrix34**190.

# Arithmetic

See **br_transform**349.

## Conversion

*From Quaternions, Matrices and Transforms*

See **BrQuatToEuler()** 323, **BrMatrix34ToEuler()** 207, **BrTransformToTransform()** 353.

*To Quaternions, Matrices and Transforms*

See **BrEulerToQuat()** 123, **BrEulerToMatrix34()** 123, and **BrEulerToMatrix4()** 124 as described below.

Also see **BrTransformToTransform()** 353.

---

# BrEulerToQuat()

*Description:* Convert a **br_euler** 121 to a quaternion, that would have the same transformational effect.

*Declaration:* **br_quat* BrEulerToQuat(br_quat* q, br_euler* euler)**

*Arguments:* **br_quat * q**

A pointer to the destination quaternion to receive the conversion.

**br_euler * euler**

A pointer to the source Euler angle.

*Result:* **br_quat ***

Returns `q` for convenience.

---

# BrEulerToMatrix34()

*Description:* Convert a **br_euler** 121 to a 3D affine matrix, that would have the same transformational effect.

*Declaration:* **br_matrix34* BrEulerToMatrix34(br_matrix34* mat, const br_euler* euler)**

*Arguments:* **br_matrix34 * mat**

A pointer to the destination matrix to receive the conversion.

**const br_euler * euler**

A pointer to the source Euler angle.

*Result:* **br_matrix34 ***

Returns `mat` for convenience.

## BrEulerToMatrix4()

*Description:* Convert a **br_euler**[121] to a 3D affine matrix, that would have the same transformational effect.

*Declaration:* **br_matrix4* BrEulerToMatrix4(br_matrix4* mat, const br_euler* euler)**

*Arguments:* **br_matrix4 * mat**

A pointer to the destination matrix to receive the conversion.

**const br_euler * euler**

A pointer to the source Euler angle.

*Result:* **br_matrix4 ***

Returns `mat` for convenience.

*Remarks:* Equivalent to **BrEulerToMatrix34()**[123].

## Copy/Assign

Copy by assignment. Freely reference.

## Initialisation

Initialise by member (use `BR_ANGLE_DEG()` or `BR_ANGLE_RAD()`) or structure assignment. Zero is a valid order.

# br_face

## The Structure

The face data structure, describing a single triangular face.

### *The* ***typedef***

(See *model.h* for precise declaration and ordering)

| | | |
|---|---|---|
| **br_uint_16** | **vertices[3]** | ***Vertices around face*** |
| **br_material *** | **material** | ***Face material*** |
| **br_uint_16** | **smoothing** | ***Controls smoothing between shared edges*** |
| **br_uint_8** | **flags** | ***Face flags*** |

### *Related Structures*

See **br_model**[232] for the structure in which vertices are used.

## Members

### br_uint_16 vertices[3]

An array of vertex indices specifying the vertices of this face. This defines a polygon of the model's the surface. The order in which vertices are listed is important. The primary, visible side of a face from the viewpoint has its vertices listed in anticlockwise order.

See **br_model**[232] and **br_vertex**[378].

### br_material material

Pointer to the material structure associated with this face. Note that if this is NULL and the face is part of a model actor's model, then the model actor's material (as specified or inherited) will be used.

### br_uint_16 smoothing

A 16 bit field in which each bit represents a smoothing group. If, when smooth-shading a surface, two adjacent faces share a smoothing group, the edge between them will be smooth.

### br_uint_8 flags

Face flags, indicating whether the edges of the face abut co-planar faces, and thus do not need to be drawn in the wire-frame render style BR_RSTYLE_EDGES. The following table describes each flag.

| Flag | Meaning |
|---|---|
| BR_FACEF_COPLANAR_0 | The face adjoining edge 0 is co-planar with this face. |
| BR_FACEF_COPLANAR_1 | The face adjoining edge 1 is co-planar with this face. |
| BR_FACEF_COPLANAR_2 | The face adjoining edge 2 is co-planar with this face. |

## Copy/Assign

Face structures should not be copied by assignment – copy member-wise instead. Note that faces so copied, will still need the models that refer to them to be updated before they are involved in rendering.

## Access & Maintenance

Members may be freely accessed. The structure should remain valid while it is referenced by any model. Some private members are modified if **BrModelUpdate()** 241 is applied to the model that references the face. Any changes to members will not affect models that use the face until **BrModelUpdate()** 241 is called. **BrModelUpdate()** 241 must be called before rendering if any changes have been made to faces of models that are in the registry.

## Referencing & Lifetime

Be careful of referencing faces especially ones allocated by **BrModelAllocate()** 243, they are liable to be moved around during **BrModelUpdate()** 241, say. Faces are generally only allocated as arrays completely describing a model. Always access using indexing from the model's `faces` member.

The structure should remain valid while it is referenced by any model. If constructed by **BrModelAllocate()** 243 it will be destroyed when the model it was constructed with is destroyed (or upon **BrModelUpdate()** 241 if `BR_MODF_KEEP_ORIGINAL` had not been set beforehand).

## Initialisation

If not constructed by **BrModelAllocate()** 243, ensure the structure is first zeroed, e.g using `memset(...,0,sizeof(br_face))`. Set the members appropriately. Models that refer to such faces must have the flag `BR_MODF_KEEP_ORIGINAL` set. Models referring to initialised faces must be updated before they are involved in rendering.

## Construction & Destruction

If this structure is constructed by **BrModelAllocate()** 243 (or indirectly by **BrModelLoad()** 247) it will be destroyed with the model (or upon **BrModelUpdate()** 241 if `BR_MODF_KEEP_ORIGINAL` had not been set beforehand). It may be constructed (and appropriately destroyed) any other way (as long as `BR_MODF_KEEP_ORIGINAL` is set in the model).

## Supplementary

When constructed by **BrModelAllocate()**[243] faces are allocated from the "FACES" memory class. It is probably better to organise any enumeration around models (see **br_model**[232]).

## Import & Export

Faces can be imported and exported with models.

# br_filesystem

## The Structure

BRender routes all filing system calls through an instance of this structure. The syntax of each call-back function corresponds exactly with the standard C library calls. This allows the user to tailor BRender's file system characteristics to suit any platform.

See **BrFilesystemSet()** 130 for details of how to specify a particular filing system handler.

### *The* ***typedef***

(See *brfile.h* for precise declaration and ordering)

Informational

**brfile_attributes_cbfn * attributes***Function to inquire capabilities of filing system*

Filing functions

| | | |
|---|---|---|
| **brfile_open_read_cbfn *** | **open_read** | ***Function to open a file for reading*** |
| **brfile_open_write_cbfn *** | **open_write** | ***Function to open a file for writing*** |
| **brfile_close_cbfn *** | **close** | ***Function to close an opened file*** |
| **brfile_eof_cbfn *** | **eof** | ***Function to check for end of file*** |
| **brfile_getchr_cbfn *** | **getchr** | ***Function to read one character*** |
| **brfile_putchr_cbfn *** | **putchr** | ***Function to write one character*** |
| **brfile_read_cbfn *** | **read** | ***Function to read a block*** |
| **brfile_write_cbfn *** | **write** | ***Function to write a block*** |
| **brfile_getline_cbfn *** | **getline** | ***Function to read a line of text*** |
| **brfile_putline_cbfn *** | **putline** | ***Function to write a line of text*** |
| **brfile_advance_cbfn *** | **advance** | ***Function to advance through a stream*** |

Supplementary

| | | |
|---|---|---|
| **char *** | **identifier** | ***Name of filing system*** |

## Members

*Informational*

### brfile_attributes_cbfn *attributes

This is a pointer to the function to obtain attributes of the filing system (see **brfile_attributes_cbfn**382). This is called by **BrFileAttributes()** 57 (See Filing System Support, page 57).

*Filing Functions*

### brfile_open_read_cbfn * open_read

This is a pointer to the function to open a file for reading (see **brfile_open_read_cbfn**387). This is called by **BrFileOpenRead()**60 (See Filing System Support, page 57).

### brfile_open_write_cbfn * open_write

This is a pointer to the function to open a file for writing (see **brfile_open_write_cbfn**389). This is called by **BrFileOpenWrite()**57 (See Filing System Support, page 57).

### brfile_close_cbfn * close

This is a pointer to the function to close a file (see **brfile_close_cbfn**383). This is called by **BrFileClose()**63 (See Filing System Support, page 57).

### brfile_eof_cbfn * eof

This is a pointer to the function to check for end of file (see **brfile_eof_cbfn**384). This is called by **BrFileEof()**63 (See Filing System Support, page 57).

### brfile_getchr_cbfn * getchr

This is a pointer to the function to read one character (see **brfile_getchr_cbfn**385). This is called by **BrFileGetChar()**62 (See Filing System Support, page 57).

### brfile_putchr_cbfn * putchr

This is a pointer to the function to write one character (see **brfile_putchr_cbfn**390). This is called by **BrFilePutChar()**59 (See Filing System Support, page 57).

### brfile_read_cbfn * read

This is a pointer to the function to read a block (see **brfile_read_cbfn**392). This is called by **BrFileRead()**61 (See Filing System Support, page 57).

### brfile_write_cbfn * write

This is a pointer to the function to write a block (see **brfile_write_cbfn**393). This is called by **BrFileWrite()**58 (See Filing System Support, page 57).

### brfile_getline_cbfn * getline

This is a pointer to the function to read a line of text (see **brfile_getline_cbfn**$_{386}$). This is called by **BrFileGetLine()**$_{61}$ (See Filing System Support, page 57).

### brfile_putline_cbfn * putline

This is a pointer to the function to write a line of text (see **brfile_putline_cbfn**$_{391}$). This is called by **BrFilePutLine()**$_{59}$ (See Filing System Support, page 57).

### brfile_advance_cbfn * advance

This is a pointer to the function to advance through a stream (see **brfile_getchr_cbfn**$_{385}$). This is called by **BrFileGetChar()**$_{62}$ (See Filing System Support, page 57).

*Supplementary*

### char * identifier

Pointer to unique, zero terminated, character string (or NULL if not required). A string constant is recommended.

## Operations

---

## BrFilesystemSet()

*Description:* Install a new file system.

*Declaration:* **const br_filesystem* BrFilesystemSet(const br_filesystem* newfs)**

*Arguments:* **const br_filesystem * newfs**

A pointer to an instance of a **br_filesystem**$_{128}$ structure.

*Result:* **const br_filesystem ***

Returns a pointer to the old **br_filesystem**$_{128}$ structure.

---

## Copy/Assign

Beware of copying the structure if identifier has been allocated from the heap.

## Access & Maintenance

While the structure is the current handler (most recently passed to **BrFilesystemSet()** 130) the members should not be changed. No maintenance required.

## Referencing & Lifetime

The structure must remain valid until a new handler is passed to **BrFilesystemSet()** 130.

## Initialisation

The members should be set before the structure is passed to **BrFilesystemSet()** 130.

## Construction & Destruction

The structure should ideally be statically constructed.

## Supplementary

The `identifier` may be used to determine the current handler in use.

# br_fixed_[ls][su][f]

## The Integral Types

These are BRender's current fixed point types. The application programmer should never use these types directly. The documentation for these types is only included for completeness and is not meant to validate their use. The representation of fixed point values and the validity of operations upon them is likely to change in the future.

### *The* ***typedef***

(See *fixed.h* for precise declarations)

| | | |
|---|---|---|
| **long** | **br_fixed_ls** | ***Fixed – Long Signed(15.16)*** |
| **short** | **br_fixed_lsf** | ***Fixed – Long Signed Fraction(0.15*)*** |
| **unsigned long** | **br_fixed_lu** | ***Fixed – Long Unsigned(16.16)*** |
| **unsigned short** | **br_fixed_luf** | ***Fixed – Long Unsigned Fraction(0.16)*** |
| **short** | **br_fixed_ss** | ***Fixed – Short Signed (7.8)*** |
| **char** | **br_fixed_ssf** | ***Fixed – Short Signed Fraction (0.7*)*** |
| **unsigned short** | **br_fixed_su** | ***Fixed – Short Unsigned (8.8)*** |
| **unsigned char** | **br_fixed_suf** | ***Fixed – Short Unsigned Fraction(0.8)*** |

## Arithmetic

Fixed point values may be negated.

Two fixed point values of the same type may be added and subtracted.

A fixed point value may be multiplied or divided by a standard integral type.

The following functions all accept **br_fixed_ls**$_{132}$ arguments (except where otherwise stated).

**br_fixed_ls BrFixedAbs(br_fixed_ls a)**

Return the equivalent of `abs(a)`.

**br_fixed_ls BrFixedMul(br_fixed_ls a, br_fixed_ls b)**

Return the equivalent of `a*b`.

**br_fixed_ls BrFixedMac2(br_fixed_ls a, br_fixed_ls b, br_fixed_ls c, br_fixed_ls d)**

Return the equivalent of `a*b + c*d`.

**br_fixed_ls BrFixedMac3(br_fixed_ls a, br_fixed_ls b, br_fixed_ls c, br_fixed_ls d,br_fixed_ls e, br_fixed_ls f)**

Return the equivalent of `a*b + c*d + e*f`.

**br_fixed_ls BrFixedMac4(br_fixed_ls a, br_fixed_ls b, br_fixed_ls c,**

* Note that signed fractions have one fewer bit than the fractional component of their non-fractional counterparts to make way for the sign bit.

**br_fixed_ls d,br_fixed_ls e, br_fixed_ls f, br_fixed_ls g, br_fixed_ls h)**

Return the equivalent of `a*b + c*d + e*f + g*h`.

**br_fixed_ls BrFixedSqr(br_fixed_ls a)**

Return the equivalent of `a*a`.

**br_fixed_ls BrFixedSqr2(br_fixed_ls a, br_fixed_ls b)**

Return the equivalent of `a*a + b*b`.

**br_fixed_ls BrFixedSqr3(br_fixed_ls a, br_fixed_ls b, br_fixed_ls c)**

Return the equivalent of `a*a + b*b + c*c`.

**br_fixed_ls BrFixedSqr4(br_fixed_ls a, br_fixed_ls b, br_fixed_ls c, br_fixed_ls d)**

Return the equivalent of `a*a + b*b + c*c + d*d`.

**br_fixed_ls BrFixedLength2(br_fixed_ls a, br_fixed_ls b)**

Return the equivalent of `sqrt(a*a + b*b)`.

**br_fixed_ls BrFixedLength3(br_fixed_ls a, br_fixed_ls b, br_fixed_ls c)**

Return the equivalent of `sqrt(a*a + b*b + c*c)`.

**br_fixed_ls BrFixedLength4(br_fixed_ls a, br_fixed_ls b, br_fixed_ls c, br_fixed_ls d)**

Return the equivalent of `sqrt(a*a + b*b + c*c + d*d)`.

**br_fixed_ls BrFixedRLength2(br_fixed_ls a, br_fixed_ls b)**

Return the equivalent of `1/sqrt(a*a + b*b)` (low precision).

**br_fixed_ls BrFixedRLength3(br_fixed_ls a, br_fixed_ls b, br_fixed_ls c)**

Return the equivalent of `1/sqrt(a*a + b*b + c*c)` (low precision).

**br_fixed_ls BrFixedRLength4(br_fixed_ls a, br_fixed_ls b, br_fixed_ls c, br_fixed_ls d)**

Return the equivalent of `1/sqrt(a*a + b*b + c*c + d*d)` (low precision).

**br_fixed_ls BrFixedDiv(br_fixed_ls a, br_fixed_ls b)**

Return the equivalent of `a/b`.

**br_fixed_ls BrFixedDivF(br_fixed_ls a, br_fixed_ls b)**

Return the equivalent of `a/b * 2^31`.

**br_fixed_ls BrFixedDivR(br_fixed_ls a, br_fixed_ls b)**

Return the equivalent of `a/b` (rounded towards zero).

**br_fixed_ls BrFixedMulDiv(br_fixed_ls a, br_fixed_ls b, br_fixed_ls c)**

Return the equivalent of `a*b/c`.

**br_fixed_ls BrFixedMac2Div(br_fixed_ls a, br_fixed_ls b, br_fixed_ls c, br_fixed_ls d,br_fixed_ls e)**

Return the equivalent of `(a*b + c*d)/e`.

**br_fixed_ls BrFixedMac3Div(br_fixed_ls a, br_fixed_ls b, br_fixed_ls c,**

**br_fixed_ls d,br_fixed_ls e, br_fixed_ls f, br_fixed_ls g)**

Return the equivalent of `(a*b + c*d + e*f)/g`.

**br_fixed_ls BrFixedMac4Div(br_fixed_ls a, br_fixed_ls b, br_fixed_ls c, br_fixed_ls d,br_fixed_ls e, br_fixed_ls f, br_fixed_ls g, br_fixed_ls h, br_fixed_ls i)**

Return the equivalent of `(a*b + c*d + e*f + g*h)/i`.

**br_fixed_ls BrFixedRcp(br_fixed_ls a)**

Return the equivalent of `1.0/a`.

**br_fixed_ls BrFixedFMac2(br_fixed_lsf a, br_fixed_ls b, br_fixed_lsf c, br_fixed_ls d)**

Return the equivalent of `a*b + c*d` (`a` & `c` are fractions).

**br_fixed_ls BrFixedFMac3(br_fixed_lsf a, br_fixed_ls b, br_fixed_lsf c, br_fixed_ls d,br_fixed_lsf e, br_fixed_ls f)**

Return the equivalent of `a*b + c*d + e*f` (`a`, `c` & `e` are fractions).

**br_fixed_ls BrFixedFMac4(br_fixed_ls a, br_fixed_ls b, br_fixed_ls c, br_fixed_ls d,br_fixed_ls e, br_fixed_ls f, br_fixed_ls g, br_fixed_ls h)**

Return the equivalent of `a*b + c*d + e*f + g*h` (`a`, `c`, `e`, & `g` are fractions).

**br_fixed_ls BrFixedSin(br_angle a)**

Return the equivalent of `sin(a)` (see **br_angle**[105]).

**br_fixed_ls BrFixedCos(br_angle a)**

Return the equivalent of `cos(a)` (see **br_angle**[105]).

**br_angle BrFixedASin(br_fixed_ls s)**

Return the equivalent of `asin(s)` (see **br_angle**[105]).

**br_angle BrFixedACos(br_fixed_ls c)**

Return the equivalent of `acos(c)` (see **br_angle**[105]).

**br_angle BrFixedATan2(br_fixed_ls x, br_fixed_ls y)**

Return the equivalent of `atan2(x,y)` (see **br_angle**[105]).

**br_angle BrFixedATan2Fast(br_fixed_ls x, br_fixed_ls y)**

Return the equivalent of `atan2(a)` (low precision) (see **br_angle**[105]).

**br_fixed_ls BrFixedSqrt(br_fixed_ls a)**

Return the equivalent of `sqrt(a)`.

**br_fixed_ls BrFixedPow(br_fixed_ls a, br_fixed_ls b)**

Return the equivalent of `pow(a,b)`.

## Comparison

All standard comparison operators may be applied between the same types. Comparison with zero is also valid for all types.

# Conversion

## *From 1*

The following macros evaluate to unity in each type, e.g. `BR_ONE_LS` may be cast to **`br_fixed_ls`**$_{1.32}$ and represents unity for that type. However, note that for fractional types, unity can not be stored; the macro is intended for use as a factor.

`BR_ONE_LS`
`BR_ONE_LSF`
`BR_ONE_LU`
`BR_ONE_LUF`
`BR_ONE_SS`
`BR_ONE_SSF`
`BR_ONE_SU`
`BR_ONE_SUF`

## *From integral types*

`BrIntToFixed(i)`

Converts `i` from a `short` or `unsigned short` to a value suitable to cast to **`br_fixed_ls`**$_{1.32}$ or **`br_fixed_lu`**$_{1.32}$ respectively.

`BrFloatToFixed(f)`

Converts `f` from an floating point value to a value of type **`br_scalar`**$_{34.2}$ (itself **`br_fixed_ls`**$_{1.32}$)

## *From Other BRender Types*

**`br_fixed_ls BrScalarToFixed(br_scalar s)`**

Converts `s` from scalar type to fixed type.

## *To integral types*

`BrFixedToInt(`**`i`**`)`

Converts `i` from a value of type **`br_fixed_ls`**$_{1.32}$ or **`br_fixed_lu`**$_{1.32}$ to a value suitable to cast to a `short` or `unsigned short` respectively.

`BrFixedToFloat(s)`

Converts `s` from a value of type **`br_fixed_ls`**$_{1.32}$ or **`br_fixed_lu`**$_{1.32}$ to a value suitable to cast to a `float`.

## *To Other BRender Types*

**`br_scalar BrFixedToScalar(br_fixed_ls f)`**

Converts `f` from fixed type to scalar type.

`br_fixed_[ls][su][f]`

## Copy/Assign

Only assign zero, though Fractions may be assigned to non-Fractions of the same type, e.g. '`lsf` to '`ls`. Use conversions in all other cases.

# br_font

## The Structure

The font data structure, describing a BRender font. Up to 223 bit mapped characters are supported. The font does not necessarily need to accord with ASCII character codes, however the non-printing ASCII codes (0–31 & 127) of the 256 codes possible are reserved for such a purpose.

Three ASCII fonts (covering codes 32-126) are predefined by BRender:
`BrFontFixed3x5` 3 pixels wide by 5 high, fixed pitch font
`BrFontProp4x6` 4 pixels wide by 6 high, pixel proportional font
`BrFontProp7x9` 7 pixels wide by 9 high, pixel proportional font

### *The* ***typedef***

(See *brfont.h* for precise declaration and ordering)

| | | |
|---|---|---|
| **br_uint_32** | **flags** | ***Properties of the font*** |
| **br_uint_16** | **glyph_x** | ***Pixel width of fixed pitch characters*** |
| **br_uint_16** | **glyph_y** | ***Pixel height of characters*** |
| **br_int_16** | **spacing_x** | ***Horizontal pixel spacing of characters*** |
| **br_int_16** | **spacing_y** | ***Vertical pixel spacing of characters*** |
| **br_int_8 *** | **width** | ***Pixel widths of proportional characters*** |
| **br_uint_16 *** | **encoding** | ***Offsets to data for each character*** |
| **br_uint_8 *** | **glyphs** | ***Pointer to bit mapped character data*** |

### *Related Functions*

See **BrPixelmapText()** 283, **BrPixelmapTextF()** 283, **BrPixelmapTextWidth()** 284, **BrPixelmapTextHeight()** 285.

## Members

### br_uint_32 flags

The `flags` member contains various details about the properties of the font such as whether it is proportional or not.

The presence of the flag whose value is defined by the symbol `BR_FONTF_PROPORTIONAL` indicates that the spacing between characters is proportional to their widths (`width` is used), otherwise each character is regularly spaced (`glyph_x` is used).

### br_uint_16 glyph_x

The width of characters in fixed pitch fonts. Note that there is an implicit single pixel gap between characters.

### br_uint_16 glyph_y

The height of the font in pixels (the number of pixel rows making up the largest character).

### br_int_16 spacing_x

The width in pixels between horizontally adjacent character co-ordinates. This is not currently implemented by BRender, but could be used to determine the spacing between columns of text when interpreting ASCII **HTAB** say.

### br_int_16 spacing_y

The height in pixels between vertically adjacent character co-ordinates. This is not currently implemented by BRender, but could be used to determine the spacing of between rows of text when interpreting ASCII **VTAB** or **CRLF** say.

### br_int_8 * width

Pointer to an array of 256 widths in pixels of each character (in proportional fonts). Values for ASCII control codes (0-31 & 127) have reserved meanings. Note that there is an implicit single pixel gap between characters.

### br_uint_16 * encoding

Pointer to an array of 256 offsets to each character's bit map (values for ASCII control codes (0-31 & 127) have reserved meanings), with a set bit indicating character foreground colour, and a cleared bit indicating transparent.

Each character is formed of a number of rows of bytes. The number of rows in each character is equal to `glyph_y`. The number of bytes in each row is given by the integer formula below, where *Width* is the pixel width of the character (`glyph_x` in the case of fixed pitch fonts and `width[`*char*`]` in the case of proportional fonts).

$$\text{Number of bytes in each row} = \frac{Width + 7}{8}$$

Therefore, in proportional fonts, it is possible that some characters will have a different number of bytes per row.

The character's pixels are arranged in memory such that the character's top left hand corner occupies the most significant bit of the first byte. Naturally a character row may use fewer pixels than all of the bits of the bytes it occupies, in such a case the least significant bits of the last byte of each row will be unused. For example, say the ASCII letter 'E' was stored in a font called `my_font` as a 10x18 character. It would be stored in 18 pairs of bytes (36 bytes) at an address pointed to by `my_font.glyphs+my_font.encoding['E']`. The first byte would contain the pixels for the left hand 8 dots of the top row. The second byte's two most significant bits would contain the remaining two dots of the top row. Subsequent bytes would contain pixels for lower rows in a similar fashion.

`br_uint_8 * glyphs`

Pointer to raw font data containing bit maps for each character. In proportional fonts the first byte should not be used for character data so that an `encoding` offset of zero can be considered `NULL`. For the same purpose, in fixed pitch fonts, there should be a dummy character definition at an `encoding` offset of zero. Note that the space character should not use this definition, but have its own (therefore at a non-zero offset*).

## Copy/Assign

May be freely copied. However, be careful of a structure assignment if the lifetime of the original font data is unknown.

Given that font data can be arranged in a convoluted manner, there is not necessarily a single most efficient way of copying it. One way to copy font data is to first determine the extent of the data by scanning the list of offsets and obtaining the maximum extent of each character's data (including the character's width). Another way is to copy font data character by character.

## Access & Maintenance

The font and its data are not guaranteed to be writable, but otherwise members may be freely accessed (although this should be avoided during text operations). Do not modify the supplied fonts (though their data may be copied and subsequently modified if desired).

## Referencing & Lifetime

The font should be maintained while references to it exist.

## Initialisation

Initialise the entire structure to zero (using `memset(,0,sizeof(br_font))`, say) and ensure all the above members are set appropriately. The elements of `width` and `encoding` corresponding to ASCII control codes (0-31 & 127) should be set to zero. All of the remaining 223 elements must have valid values. Character data can be stored in any order, may overlap and need not be contiguous. In proportional fonts, codes for which no character is defined should have a `width` of zero and a zero `encoding` offset. In fixed pitch fonts, codes for which no character is defined should have a zero `encoding` offset (therefore a dummy character definition is required at an offset of zero).

## Import & Export

There are no specific font load/save functions. See Filing System Support for details of BRender functions that could be used to assist font import and export.

Do not be beguiled by the apparent simplicity of storing fonts within monochrome pixel maps. There are a few points at which this idea has difficulties. For instance, a monochrome pixel map may require

* Given that overlap is permitted, only the first byte is 'wasted' as the space definition can have an offset of 1.

4 byte alignment of pixel rows, whereas a font requires single byte alignment. Also, with proportional fonts, characters may not necessarily all have the same number of bytes per row, so it may not be possible to fit them all into a neat column.

# br_fraction

## The Integral Type

The **br_fraction**$_{141}$ type can be used to represent numbers in the range [-1,+1)*. Although used internally, this type is not currently used by the BRender API.

Under the floating point library, **br_fraction**$_{141}$ is a `float`. Under the fixed point library, **br_fraction**$_{141}$ is a 16 bit signed fixed point number.

### *The* ***typedef***

(See *scalar.h* for precise declaration)

| | | |
|---|---|---|
| **float** | **br_fraction** | ***Floating point Signed Fraction*** |
| **br_fixed_lsf** | **br_fraction** | ***Fixed – Long Signed Fraction(0.15)*** |

## Arithmetic

No standard operators are supported. Moreover, there are few macros that directly support fractional arithmetic. Convert to **br_scalar**$_{342}$ and use that type's arithmetic macros instead.

The following macros are provided to assist with fractional vector arithmetic (see **br_fraction**$_{141}$). In the following macros, arguments `a`, `c`, `e` and `g` are of type **br_fraction**$_{141}$, `b`, `d`, `f` and `h` are of type **br_scalar**$_{342}$. The macro computes a **br_scalar**$_{342}$ result.

`BR_FMAC2(a,b,c,d)`

Return the equivalent of `a * b + c * d .`

`BR_FMAC3(a,b,c,d,e,f)`

Return the equivalent of `a * b + c * d + e * f .`

`BR_FMAC4(a,b,c,d,e,f,g,h)`

Return the equivalent of `a * b + c * d + e * f + g * h .`

## Comparison

Equality and comparison with zero are valid. Other standard comparison operators may be implemented by macros in future versions, but are currently valid.

## Conversion

### *From Numeric Constants*

`BR_FRACTION(x)`

Convert `x` from any numeric constant to **br_fraction**$_{141}$.

---

* Maximum value is thus `BR_ONE_LS-BR_SCALAR_EPSILON` (in the fixed point library).

`br_fraction`

`BR_SCALAR_EPSILON`

Smallest positive fractional value.

### *From Integral Types*

To convert from integral types, use **`br_scalar`**[342] as an intermediary.

### *From* **`br_scalar`**[342]

**`br_fraction BrScalarToFraction(br_scalar s)`**

Converts `s` from **`br_scalar`**[342] to **`br_fraction`**[141]. It is up to the application to ensure the value is in the required range.

### *To Integral Types*

To convert to integral types, use **`br_scalar`**[342] as an intermediary.

### *To* **`br_scalar`**[342]

**`br_scalar BrFractionToScalar(br_fraction f)`**

Converts `s` from **`br_fraction`**[141] to **`br_scalar`**[342]. It is up to the application to ensure the value is in the required range.

## Copy/Assign

Only assign zero, or fractions. Use conversions in all other cases.

# br_fvector2

## The Structure

This is the two ordinate, fractional vector structure. It is not recommended for use, and is only documented here for completeness.

### *The* ***typedef***

(See *vector.h* for precise declaration and ordering)

| | | |
|---|---|---|
| **br_fraction** | **v[2]** | ***Ordinates (0=x, 1=y)*** |

## Members

### br_fraction v[2]

First and second ordinate. Conventionally, the first ordinate is the x axis component, and the second, the y axis component.

## Arithmetic

No approved arithmetic.

## Copy/Assign

No approved assignment.

## Initialisation

The following macro may be used as a static initialiser.

**BR_FVECTOR2(a,b)**

Macro expands to {BR_FRACTION(a),BR_FRACTION(b)}.

All other initialisation should use member-wise initialisers.

# br_fvector3

## The Structure

This is the three ordinate fractional vector structure, typically used for storing surface normals. It is not intended for use by the application and is documented here only for completeness.

### *The* **typedef**

(See *vector.h* for precise declaration and ordering)

| | | |
|---|---|---|
| **br_fraction** | **v[3]** | ***Ordinates (0=x, 1=y, 2=z)*** |

### *Related Structures*

See **br_vector3**[367].

This structure is currently used by **br_vertex**[378], and **br_face**[125]. It's presence must not be relied upon.

## Members

### br_fraction v[3]

First, second and third ordinate. Conventionally, the first ordinate is the x-axis component, the second, the y axis component, and the third, the z axis component.

## Arithmetic

No approved arithmetic.

## Copy/Assign

No approved assignment.

## Initialisation

The following macro may be used as a static initialiser.

BR_FVECTOR3(a,b,c)

Macro expands to {BR_FRACTION(a), BR_FRACTION(b), BR_FRACTION(c)}.

All other initialisation should use member-wise initialisers.

# br_fvector4

## The Structure

This is the four ordinate fractional vector structure. It is not intended for use by the application and is documented here only for completeness.

### *The* **typedef**

(See *vector.h* for precise declaration and ordering)

**br_fraction** **v[4]** ***Ordinates (0=x, 1=y, 2=z, 3=w)***

### *Related Structures*

See **br_vector4**[376].

## Members

### br_fraction v[4]

First, second, third, and fourth ordinate.

## Arithmetic

No approved arithmetic.

## Copy/Assign

No approved assignment.

## Initialisation

The following macro may be used as a static initialiser.

**BR_FVECTOR4(a,b,c,d)**

Macro expands to {BR_FRACTION(a), BR_FRACTION(b), BR_FRACTION(c), BR_FRACTION(d)}.

All other initialisation should use member-wise initialisers.

# br_int_8/16/32

## The Integral Type

BRender's signed integer types. Use this type where the integer word length is critical.

### *The* **`typedef`**

(See *`compiler.h`* for precise declaration)

| | | |
|---|---|---|
| **`signed char`** | **`br_int_8`** | ***8 bit signed integer*** |
| **`signed short`** | **`br_int_16`** | ***16 bit signed integer*** |
| **`signed long`** | **`br_int_32`** | ***32 bit signed integer*** |

## Arithmetic

All standard C arithmetic operators are valid as with standard integer types.

## Comparison

All standard C comparison operators are valid as with standard integer types.

## Conversion

Use casts as with any other standard C type.

## Copy/Assign

Freely assign. Use as a standard C type, as this type is only concerned with ensuring specific sizes – not representation.

# br_light

## The Structure

This structure is used to specify the properties of light actors. See Light Actors.

### *The* **typedef**

(See *light.h* for precise declaration and ordering)

#### Behaviour

| | | |
|---|---|---|
| **br_uint_8** | **type** | ***Light type*** |
| **br_colour** | **colour** | ***Light colour*** |

#### Attenuation

| | | |
|---|---|---|
| **br_scalar** | **attenuation_c** | ***Light intensity*** |
| **br_scalar** | **attenuation_l** | ***Linear attenuation factor for point and spot lights*** |
| **br_scalar** | **attenuation_q** | ***Quadratic attenuation factor for point and spot lights*** |

#### Spot Lighting

| | | |
|---|---|---|
| **br_angle** | **cone_inner** | ***The angle giving the cone of full-intensity light cast by a spot*** |
| **br_angle** | **cone_outer** | ***The angle giving the cone of illumination of a spot*** |

#### Supplementary

| | | |
|---|---|---|
| **char *** | **identifier** | ***Light name*** |
| **void *** | **user** | ***User data (application dependent)*** |

### *Related Functions*

#### Scene Modelling

See **BrLightEnable()** 87, and **BrLightDisable()** 88.

# Members

### *Behaviour*

## br_uint_8 type

This member defines the type of the light. Note that different light types can affect rendering performance in different ways. Pre-lighting textures is the fastest means of lighting objects in a scene, ambient lighting comes next, then the three active lighting methods: direct, point and spot lights (in order of increasing processing requirements). The more lights in a scene*, the more computation required.

The three active light types are described in the following table:

| Light Type Symbol | Behaviour |
|---|---|
| BR_LIGHT_DIRECT | A directed light source. The light is infinitely distant and shines along the negative z axis of the light actor. |
| BR_LIGHT_POINT | A point light source, radiating in all directions. |
| BR_LIGHT_SPOT | A spot light, with both spatial location and direction. A spot light has a cone of illumination and shines along the negative z axis of the light actor. |

The type member also encodes a flag value BR_LIGHT_VIEW which can be combined with any of the three light types using the 'inclusive-or' operation.When the flag is present, lighting calculations are performed in view space rather than model space. This is slower, but prevents odd anomalies if non-uniform scalings are used on models.

The bit mask value BR_LIGHT_TYPE can be used to extract the light type using the 'and' operation.

## br_colour colour

If the rendering engine supports it, a light may be of a specific colour. Typically supported when rendering in 'true colour'.

### *Attenuation*

The linear and quadratic attenuation factors determine the degree by which light intensity falls off with respect to distance and squared distance from the light source. If $d$ is distance, and $A_c$, $A_l$, $A_q$ are the constant, linear and quadratic attenuation factors, then light intensity $L$ is given by:

$$L \propto \frac{1}{A_c + A_l d + A_q d^2}$$

If a light actor is created with a default specification, the default values supplied for $A_c$, $A_l$, $A_q$ are 1.0, 0.0 and 0.0 respectively.

* The maximum number of enabled light actors in a scene is defined by the symbol BR_MAX_LIGHTS.

### br_scalar attenuation_c

Constant attenuation component. Light intensity is inversely proportional to this value.

### br_scalar attenuation_l

Linear attenuation factor. Light intensity is inversely proportional to distance factored by this value. It only applies to point and spot lights.

### br_scalar attenuation_q

Quadratic attenuation factor. Light intensity is inversely proportional to distance factored by the square of this value. It only applies to point and spot lights.

#### *Spot Lighting*

Spot lights are defined in terms of an inner, fully lit cone of light and an outer, penumbral cone of light diminishing from fully lit to unlit. The intensity fall off between the inner and outer cones is linear.

The effect of having the outer cone angle smaller than the inner cone angle is undefined.

### br_angle cone_inner

The inner cone of a spot light is defined by the angle between the cone's axis and its circumference. This represents the region within which surfaces will be fully lit (subject to attenuation).

### br_angle cone_outer

The outer cone of a spot light is defined by the angle between the cone's axis and its circumference. This represents the region within which surfaces will be partially lit (subject to attenuation). The light level falls off linearly from the fully lit level at the inner cone to zero at the limit of the outer cone.

#### *Supplementary*

### char * identifier

Pointer to unique, zero terminated, character string (or `NULL` if not required).

If `identifier` is set by **`BrActorLoad()`**$_{97}$ or **`BrActorLoadMany()`**$_{99}$ it will have been constructed using **`BrResStrDup()`**$_{49}$.

### void * user

This member may be used by the application for its own purposes. It is initialised to `NULL` upon allocation (if allocated by BRender), and not accessed by BRender thereafter.

br_light

## Copy/Assign

The light may be freely copied.

## Access & Maintenance

Lights can be modified at any time, though changes will not have any effect unless the light is enabled (see **BrLightEnable()**87). The members should always accord with the light type.

## Referencing & Lifetime

If not constructed by **BrActorAllocate()**91 it may be multiply referenced (given a sufficient lifetime). The light structure should be maintained at least as long as it is referenced by any actor.

## Initialisation

Only members applicable to the light type need be initialised.

E.g. `BrActorAllocate(BR_ACTOR_LIGHT,NULL)` currently initialises a **br_light**147 structure as follows (after performing `memset(,0,sizeof(br_light))`):

```
br_light* light;
...
light->type=BR_LIGHT_DIRECT;
light->colour=BR_COLOUR_RGB(255,255,255);
light->attenuation_c=BR_SCALAR(1.0);
```

## Construction & Destruction

The light structure can either be constructed/destroyed by the application, or by supplying `NULL` to **BrActorAllocate()**91.

## Supplementary

Lights created by **BrActorAllocate()**91 are allocated from the `BR_MEMORY_LIGHT ("LIGHT")` resource class, and hence can benefit from the resource class supplementary functions.

## Import & Export

Lights are imported and exported with actors that specify them. See **BrActorLoad()**97 and **BrActorSave()**97.

## Platform Specific

Some platforms directly support some types of light in hardware, though there is often a significant limit upon how many lights of each type are supported (also see BR_MAX_LIGHTS).

# br_map_enum_cbfn

## The Call-Back Function

This type defines a function, supplied to **BrMapEnum()** 294, and to be called by it for a selection of maps.

### *The* ***typedef***

(See *fwproto.h* for a precise declaration)

**br_uint_32 br_map_enum_cbfn(br_pixelmap*,void*)***Enumerator*

## Specification

---

## CBFnMapEnum()

*Description:* An application defined call-back function accepting a pixel map and an application supplied argument (as supplied to **BrMapEnum()** 294).

*Declaration:* **br_uint_32 BR_CALLBACK CBFnMapEnum(br_pixelmap* map, void* arg)**

*Arguments:* **br_pixelmap * map**

One of the maps selected by **BrMapEnum()** 294.

**void * arg**

The argument supplied to **BrMapEnum()** 294.

*Preconditions:* BRender has completed initialisation.

*Effects:* Application defined. Avoid adding or removing maps within this function.

*Result:* **br_uint_32**

Any non-zero value will terminate the enumeration and be returned by **BrMapEnum()** 294. Return zero to continue the enumeration.

*See Also:* **BrMapEnum()** 294, **BrMapFind()** 295.

---

# br_map_find_cbfn

## The Call-Back Function

This type defines a function, registered with **BrMapFindHook()** 295, to be called when **BrMapFind()** 295 or **BrMapFindMany()** 295 fail to find any map.

### *The* ***typedef***

(See *fwproto.h* for a precise declaration)

**br_pixelmap*** **br_map_find_cbfn(const char*)** ***Find (when* BrMapFind() *fails)***

## Specification

---

## CBFnMapFind()

*Description:* An application defined call-back function used when **BrMapFind()** 295 or **BrMapFindMany()** 295 fail.

*Declaration:* **br_pixelmap* BR_CALLBACK CBFnMapFind(const char* name)**

*Arguments:* **const char * name**

The search pattern supplied to **BrMapFind()** 295 or **BrMapFindMany()** 295 that did not match any map.

*Preconditions:* BRender has completed initialisation. No map has an identifier that successfully matches the search pattern.

*Effects:* Application defined.

*Result:* **br_pixelmap ***

Either return an existing map that is deemed appropriate for the search pattern, or NULL if there isn't one. This value will be returned by **BrMapFind()** 295 or **BrMapFindMany()** 295.

*Remarks:* This could either be used to supply a default map or to create a map. If maps were created on demand, then this function could search another list of available maps (but not yet created) and see if the pattern matched any of them, if it did, one of them could be registered and returned. Note that there is no way to supply more than one map.

*See Also:* **BrMapFind()** 295, **BrMapFindMany()** 295, **BrMapFindHook()** 295, **BrMapFindFailedLoad()** 296.

---

# br_material

## The Structure

A structure describing the appearance of a material that can be applied to a surface.

### *The* ***typedef***

(See *material.h* for precise declaration and ordering)

Behaviour

| | | |
|---|---|---|
| **br_uint_32** | **flags** | ***Flags determining how this material is rendered*** |

Lighting

| | | |
|---|---|---|
| **br_ufraction** | **ka** | ***The ambient lighting contribution*** |
| **br_ufraction** | **kd** | ***The directional lighting contribution*** |
| **br_ufraction** | **ks** | ***The specular lighting contribution*** |
| **br_scalar** | **power** | ***Specular power – the 'spread' of a specular highlight*** |

Colour

| | | |
|---|---|---|
| **br_colour** | **colour** | ***Material colour, when rendering in 'true' colour*** |
| **br_uint_8** | **index_base** | ***The unlit index (or row of a shade table)*** |
| **br_uint_8** | **index_range** | ***The range to the fully lit index (or row of a shade table)*** |
| **br_pixelmap *** | **index_shade** | ***A shade table (for indexed textures only)*** |
| **br_pixelmap *** | **index_blend** | ***A blend table (for indexed textures only)*** |

Texture

| | | |
|---|---|---|
| **br_pixelmap *** | **colour_map** | ***An optional pixel map based texture*** |
| **br_matrix23** | **map_transform** | ***Transform to apply to texture map*** |

Supplementary

| | | |
|---|---|---|
| **char *** | **identifier** | ***Material name*** |
| **void *** | **user** | ***User data (application dependent)*** |

### *Related Functions*

Scene Modelling

See **BrEnvironmentSet()** 87, **BrModelApplyMap()** 235, **BrModelFitMap()** 238.

Scene Rendering

See **BrZbModelRender()** 253.

*Related Structures*

Scene Modelling

See **br_actor**$_{76}$, **br_model**$_{232}$, **br_face**$_{125}$, **br_vertex**$_{378}$.

Scene Rendering

See **br_renderbounds_cbfn**$_{315}$, **br_model_custom_cbfn**$_{251}$.

# Members

*Behaviour*

## br_uint_32 flags

This member determines how faces using the material are rendered, in terms of other members and aspects of the scene.

| Flag Symbol | Behaviour |
|---|---|
| BR_MATF_LIGHT | The material is lit – affected by lights in the scene |
| BR_MATF_PRELIT | The material is pre-lit – colours are taken directly from models' vertex structures (see **br_vertex**$_{378}$). Any lights are ignored. |
| BR_MATF_SMOOTH | Any lighting is applied using Gouraud shading. Lighting levels are linearly interpolated between vertices. Otherwise, the same lighting level is used across the face |
| BR_MATF_DITHER | Effectively applies a filter to the texture map to soften transitions between texels – most noticeable when a texel covers several screen pixels. A carefully chosen noise component is added to the u,v texel co-ordinates (a comparable effect to bilinear interpolation). Note that transparent pixels will also be dithered. |
| BR_MATF_ENVIRONMENT_I | Texels are calculated by casting a ray from the viewpoint (extended to infinity) and reflecting it off the face, out to an enclosing sphere (onto which the supplied texture has been mapped) |
| BR_MATF_ENVIRONMENT_L | Texels are calculated by casting a ray from the (local) viewpoint and reflecting it off the face, out to an enclosing sphere (onto which the supplied texture has been mapped) |
| BR_MATF_PERSPECTIVE | The texture is rendered with correct perspective (as opposed to using linear interpolation) |
| BR_MATF_DECAL | Both the texture mapped and non-texture mapped materials are drawn, the non-texture mapped material only appearing beneath what would have been transparent elements of the texture map.<br>For example, this could be used to add symbols or logos to a smooth shaded model |
| BR_MATF_BLEND | The blend table is utilised (only for indexed textures) (see index_blend) |
| BR_MATF_ALWAYS_VISIBLE | Faces using the material will always be visible, and so back-face culling need not be performed for such faces |
| BR_MATF_TWO_SIDED | The material has two sides, and lighting calculations are performed for both of them |
| BR_MATF_FORCE_FRONT | The material is forced to be in front of all other materials |

The effects of various combinations of the first three flags are not particularly obvious so are described in the following table. The Texture column indicates whether the material is effectively textured. The Colour column indicates whether the pixel map rendered to is indexed or 'true' colour (it is assumed that any texture shares this property). The 'Pixels Set To' column describes how each pixel of a face using the material is set. Note that 'texel' is the term used to refer to the element of the texture map corresponding to a particular screen pixel.

| No. | Texture | Colour | PRELIT | LIGHT | SMOOTH | Pixels Set To |
|---|---|---|---|---|---|---|
| 1 | ✗ | 'True' | ✗ | ✗ | – | colour |
| 2 | ✗ | 'True' | ✗ | ✓ | ✗ | colour, uniformly illuminated by average face lighting |
| 3 | ✗ | 'True' | ✗ | ✓ | ✓ | colour, illuminated by linearly interpolated lighting between vertices |
| 4 | ✗ | 'True' | ✓ | – | ✗ | Average prelit vertex colour |
| 5 | ✗ | 'True' | ✓ | – | ✓ | Linearly interpolated between vertex prelit colours |
| 6 | ✗ | Indexed | ✗ | ✗ | – | index_base |
| 7 | ✗ | Indexed | ✗ | ✓ | ✗ | index_base + index_range × (average face lighting) |
| 8 | ✗ | Indexed | ✗ | ✓ | ✓ | index_base + index_range × (linearly interpolated lighting between vertices) |
| 9 | ✗ | Indexed | ✓ | – | ✗ | index_base + index_range × (average vertex prelit index ÷ 256) |
| 10 | ✗ | Indexed | ✓ | – | ✓ | index_base + index_range × (linear interpolation between vertex prelit indices ÷ 256) |
| 11 | ✓ | 'True' | ✗ | ✗ | – | Texel |
| 12 | ✓ | 'True' | ✗ | ✓ | ✗ | Texel uniformly illuminated by average face lighting |
| 13 | ✓ | 'True' | ✗ | ✓ | ✓ | Texel illuminated by linearly interpolated lighting between vertices |
| 14 | ✓ | 'True' | ✓ | – | ✗ | Texel illuminated by average prelit vertex colour |
| 15 | ✓ | 'True' | ✓ | – | ✓ | Texel illuminated by colour linearly interpolated between vertex prelit colours |
| 16 | ✓ | Indexed | ✗ | ✗ | – | Texel |
| 17 | ✓ | Indexed | ✗ | ✓ | ✗ | Shade table: column [texel], row[index_base + index_range × (average face lighting)] |
| 18 | ✓ | Indexed | ✗ | ✓ | ✓ | Shade table: column [texel], row[index_base + index_range × (linearly interpolated lighting between vertices)] |
| 19 | ✓ | Indexed | ✓ | – | ✗ | Shade table: column [texel], row[index_base + index_range × (average vertex prelit index)] |
| 20 | ✓ | Indexed | ✓ | – | ✓ | Shade table: column [texel], row[index_base + index_range × (linear interpolation between vertex prelit indices)] |

*Table showing effects of different combinations of material lighting flags*

*Lighting*

When a material is lit (BR_MATF_LIGHT is set, but not BR_MATF_PRELIT) a lighting calculation is performed to calculate the light reaching the viewer. This will depend upon the face's orientation with respect to the viewer and the light sources.

The following formula* shows how the lighting $\lambda$ of a face depends upon $\theta$, the angle at the face between the light source and the face normal, and $\phi$, the angle at the face between the viewer and the reflected light ray. Note that with high k values it is apparently possible for $\lambda$ to be as large as 3, however $\lambda$ is clamped to the range [0,1).

$$\lambda = k_{ambient} + k_{diffuse}\cos\theta + k_{specular}\cos^{p}\phi$$

The other four variables in the equation are listed respectively, below.

## br_ufraction ka

The ambient lighting contribution for lit materials. This is the amount of light assumed to be reflected from other objects and lighting in general, i.e. not from light actors. This means that even in a scene with all lights disabled, a lit material will still be visible if it has a non-zero ambient lighting contribution.

Zero can produce a material whose illumination is highly dependent upon light sources, whereas higher values can give ever fluorescent or luminous effects.

A typical sunny scene might have most materials with a significant ambient contribution, whereas a dusk scene might have a much lower one, and a moonlit one, probably zero.

## br_ufraction kd

The diffuse lighting contribution for lit materials. This determines how much of the reflected light is made up of the component dependent upon the angle of the face to the direction of the light illuminating it. The closer the face comes to being perpendicular to the light source, the more light the face receives, and thus the more diffuse light that can be reflected.

Zero can give a shiny surface, whereas higher values can give surfaces a more matt appearance.

## br_ufraction ks

The specular lighting contribution for this lit materials. This determines how much reflected light is made up of the component dependent upon the angle between the reflected light source and the direction of viewer (naturally, if the angle is zero, the component will be at its maximum).

The greater the value, the more visible highlights will be.

## br_scalar power

This member applies a power to the specular lighting contribution.

---

* An approximation to the formula may be used on some platforms, particularly with respect to the specular term.

The greater the value, the sharper any highlights will be. A typical value is 20.

*Colour*

### br_colour colour

When rendering in 'true' colour, the value of this member is taken as the basic colour of the face, which may of course be affected by lighting (if BR_MATF_LIGHT is set, but not BR_MATF_PRELIT).

### br_uint_8 index_base

When rendering in indexed colour, this member determines the lower value of the index range used to colour a face. When lit, the light level is factored with index_range to obtain an index between index_base and index_base+index_range-1. Without lighting, index_base is used to set every pixel of the face.

This member only applies to materials without textures - with textures, this member is ignored.

### br_uint_8 index_range

When rendering in indexed colour, this member determines the number of index values that can be selected with a given lighting level, starting at index_base.

This member only applies to materials without textures - with textures, this member is ignored.

### br_pixelmap * index_shade

Materials with indexed colour texture maps can only be lit if they are accompanied by an appropriate shade table. The shade table is simply a way of tabulating the output pixel given a particular texel and a particular lighting level. The texel generally indexes the column and the lighting level, the row. The shade table must therefore have the full complement of columns necessary for the pixel size of the texture map. The shade table may have any number of rows, as the lighting level directly selects the row.

Thus an 8 bit indexed colour texture map requires a shade table with 256 columns and two or more rows (one being redundant). A selection of shade table types for use with 8 bit textures are described below.

`BR_PMT_INDEX_8 shade table to BR_PMT_INDEX_8 output`

The shade table converts 8 bit texels into 8 bit lit pixel values, by using the pixel in the shade table at the column given by the texel and the row given by the monochrome lighting level.

`BR_PMT_RGB_555 shade table to BR_PMT_RGB_555 output`

The shade table converts 8 bit texels into 15 bit lit pixel values, by using the pixel in the shade table at the column given by the texel and the row given by the monochrome lighting level.

`BR_PMT_RGBX_888 shade table to BR_PMT_INDEX_8 output`

The shade table converts 8 bit texels into 8 bit lit pixel values, by using a single 256 column wide table, but with 24 bit colour values in order that true colour lighting can be applied. The red and green components of each value are actually column indices used to obtain a blue component which is produced as the output pixel. The table is thus read three times.

1. X1=Red component of pixel in shade table at column(Texel), row(Red light level)
2. X2=Green component of pixel in shade table at column(X1), row(Green light level)
3. X3=Blue component of pixel in shade table at column(X2), row(Blue light level)
4. Ouput pixel is X3

## br_pixelmap * index_blend

Blending is a way of making the destination pixel depend upon the existing contents of the output buffer. The blend table like the shade table tabulates the output pixel given a particular texel and the pixel already in the output buffer. The texel indexes the column and the existing output pixel the row. The shade table must therefore have the full complement of rows as well as columns necessary for the pixel size of the texture map.

Thus an 8 bit indexed colour texture map blend table needs 256 columns and 256 rows. The blend table is only used if `BR_MATF_BLEND` is set, and if set this member should not be `NULL`.

Given that the blend table tabulates an output for every possible pair of input pixels, any function can be represented, e.g. exclusive-or, inverse, inclusive-or, mask, average, etc.

The blend table is typically used for translucence effects, e.g. ghosts, flames, frosted glass, etc.

Note that in the case of the depth buffer renderer, while the output pixel will only be written if it is nearer than the existing pixel, it will never modify the depth buffer value. This is because it is assumed to be an intangible surface. Therefore, it may be necessary to render such materials in a separate stage.

Note that if both a blend table and shade table is used (the material has an indexed texture which is lit and blended), the shade table is applied first, followed by the blend table.

Blend tables, as with shade tables, also need to be added to the registry before use.

### *Texture*

#### br_pixelmap * colour_map

A pointer to a pixel map containing a pattern with which to cover faces using this material.

Pixels that are zero in the pixel map are not further processed for colour information, and are effectively transparent. A face is not rendered where such pixels appear on its surface, nor are any corresponding values written to any depth buffer. Note though, that a 2D pick function will still pick a transparent face. For more sophisticated transparency effects see the description of `index_blend` above.

Note that indexed colour textures must also have a corresponding shade table.

#### br_matrix23 map_transform

The transform to apply to texture co-ordinates. This enables textures to be rotated, scaled, sheared, and translated. Moreover, this transform can be continuously modified, thus providing animated texture effects.

See **`BrModelApplyMap()`**[235] for details of how texture maps can be applied to a model's faces.

### *Supplementary*

#### char * identifier

Pointer to unique, zero terminated, character string (or `NULL` if not required). Can be used as a handle to retrieve a pointer to the material. Not intended for intensive use. Typically used to collect pointers to materials loaded using **`BrMaterialLoad()`**[168] and added to the registry using **`BrMaterialAdd()`**[161]. Also ideal for diagnostic purposes.

A non-unique string can be supplied, but which of a set of materials having the same string will be matched by search functions (See **`BrMaterialFind()`**[165]), is undefined. Also in consideration of searching, it is not recommended that non-alphabetic characters are used, especially Slash ('/'), Asterisk ('*'), and Query ('?'), which are used for pattern matching.

This member can be modified by the programmer at any time.

If `identifier` is set by **`BrMaterialLoad()`**[168] or **`BrMaterialLoadMany()`**[168] it will have been constructed using **`BrResStrDup()`**[56].

`void* user`

This member may be used by the application for its own purposes. It is initialised to `NULL` upon allocation, and not accessed by BRender thereafter.

## Copy/Assign

The **`br_material`**[154] structure should not be copied directly, e.g. by structure assignment. If a similar material is required, a new one should be allocated and pertinent members copied individually. Care may be needed in copying `identifier`.

## Access & Maintenance

Materials must be added to the registry if they are involved in rendering a scene. They should not be modified during rendering.

Materials that have been added to the registry may be accessed by BRender during rendering.

If any changes are made to materials involved in rendering, they must be updated before the next rendering in which they are involved.

---

# BrMaterialAdd()

*Description:* Add a material to the registry, updating it as necessary. All materials must be added to the registry before they are subsequently involved in rendering.

*Declaration:* **`br_material* BrMaterialAdd(br_material* material)`**

*Arguments:* **`br_material * material`**

A pointer to a material.

*Result:* **`br_material *`**

Returns a pointer to the added item, else `NULL` if unsuccessful.

*See Also:* **`BrMaterialUpdate()`**[162], **`BrMaterialAddMany()`**[161], **`BrMaterialLoad()`**[168], **`BrMaterialFind()`**[165], **`BrMaterialRemove()`**[162].

---

# BrMaterialAddMany()

*Description:* Add a number of materials to the registry, updating them as necessary.

*Declaration:* **`br_uint_32 BrMaterialAddMany(br_material* const* materials, int n)`**

*Arguments:* **`br_material * const * materials`**

A pointer to an array of pointers to materials.

**`int n`**

Number of materials to add to the registry.

*Result:* **br_uint_32**

Returns the number of materials added successfully.

*See Also:* **BrMaterialUpdate()** 162, **BrMaterialAdd()** 161, **BrMaterialRemove()** 162, **BrMaterialRemoveMany()** 163

---

# BrMaterialUpdate()

*Description:* Update a material that has changed in some respect since the previous update of this material (or **BrMaterialAdd()** 161).

*Declaration:* **void BrMaterialUpdate(br_material* material, br_uint_16 flags)**

*Arguments:* **br_material * material**

A pointer to a material.

**br_uint_16 flags**

Item update flags. In general, BR_MATU_ALL should be used. However, the following table describes when to use more specific update flags. Note that flags can be combined using the 'Or' operation (BR_MATU_ALL is such a combination of the other flags, for your convenience).

| Flag | To Be Used When |
|---|---|
| BR_MATU_MAP_TRANSFORM | The map_transform has been changed |
| BR_MATU_RENDERING | The flags member has been changed. |
| BR_MATU_LIGHTING | Lighting or Colour parameters have been changed. |
| BR_MATU_COLOURMAP | Elements of the texture at colour_map have been changed |
| BR_MATU_ALL | The change is unknown or wholesale. This includes the case when an entirely different texture map is being used, i.e. colour_map is set to a different pointer |

*See Also:* **BrMaterialAdd()** 161.

---

# BrMaterialRemove()

*Description:* Remove a material from the registry.

*Declaration:* **br_material* BrMaterialRemove(br_material* material)**

*Arguments:* **br_material * material**

A pointer to a material.

*Result:* **br_material ***

Returns a pointer to the item removed.

*See Also:* **BrMaterialAdd()** 161

## BrMaterialRemoveMany()

*Description:* Remove a number of materials from the registry.

*Declaration:*
```
br_uint_32
BrMaterialRemoveMany(br_material* const* materials, int n)
```

*Arguments:* **br_material * materials**

A pointer to an array of pointers materials.

**int n**

Number of materials to remove from the registry.

*Result:* **br_uint_32**

Returns the number of items removed successfully.

*See Also:* **BrMaterialAddMany()** 161

# Referencing & Lifetime

Materials may be multiply referenced. The texture map and any shade table may be referenced by more than one material as long as their lifetimes are longer than the materials that refer to them. Materials must have been added to the registry if they will be involved in rendering. The material must be maintained while it is in the registry or being referenced.

# Initialisation

The material is automatically initialised to zero by **BrMaterialAllocate()** 163. Members should then be set appropriately. Re-initialisation is not recommended – destroy and reconstruct.

# Construction & Destruction

Apart from import and platform specific functions, materials should only be constructed by the following BRender function. Destruction should naturally be performed by the corresponding 'free' function, usually **BrMaterialFree()** 164. Note that a material should be removed from the registry before being destroyed.

## BrMaterialAllocate()

*Description:* Allocate a new material.

*Declaration:*
```
br_material* BrMaterialAllocate(const char* name)
```

*Arguments:* **const char * name**

String to initialise the `identifier` member to.

*Result:* **br_material ***

Returns a pointer to the new material, or NULL if unsuccessful.

## BrMaterialFree()

*Description:* Deallocate a material and any associated memory.

*Declaration:* **void BrMaterialFree(br_material* m)**

*Arguments:* **br_material * m**

A pointer to a material.

# Supplementary

## BrMaterialCount()

*Description:* Count the number of registered materials whose names match a given search pattern. The search pattern can include the standard wild cards '*' and '?'.

*Declaration:* **br_uint_32 BrMaterialCount(const char* pattern)**

*Arguments:* **const char * pattern**

Search pattern.

*Result:* **br_uint_32**

Returns the number of items matching the search pattern.

*See Also:* **BrMaterialEnum()** 164, **BrMaterialFind()** 165

## BrMaterialEnum()

*Description:* Calls a call-back function for every material in the registry matching a given search pattern. The call-back is passed a pointer to each matching material, and its second argument is an optional pointer supplied by the user. The search pattern can include the standard wild cards '*' and '?'. The call-back itself returns a **br_uint_32** *358* value. The enumeration will halt at any stage if the return value is non-zero.

*Declaration:* **br_uint_32 BrMaterialEnum(const char* pattern, br_material_enum_cbfn* callback, void* arg)**

*Arguments:* **const char * pattern**

Search pattern.

**br_material_enum_cbfn * callback**

A pointer to a call-back function.

**void * arg**

An optional argument to pass to the call-back function.

*Result:* **br_uint_32**

Returns the first non-zero call-back return value, or zero if all matching items are enumerated.

*Example:*

```
br_uint_32 BR_CALLBACK test_callback(br_material* material, void*
   arg)
{  br_uint_32 count;
...
   return(count);
}
...
{  br_uint_32 enum;
...
   enum = BrMaterialEnum("material",&test_callback,NULL);
}
```

## BrMaterialFind()

*Description:* Find a material in the registry by name. A call-back function can be setup to be called if the search is unsuccessful. The search pattern can include the standard wild cards '*' and '?'.

*Declaration:* **br_material* BrMaterialFind(const char* pattern)**

*Arguments:* **const char * pattern**

Search pattern.

*Result:* **br_material ***

Returns a pointer to the material if found, otherwise NULL. If a call-back exists and is called, the call-back's return value is returned.

*See Also:* **BrMaterialFindHook()** 166, **BrMaterialFindMany()** 165

## BrMaterialFindMany()

*Description:* Find a number of materials in the registry by name. The search pattern can include the standard wild cards '*' and '?'.

*Declaration:* **br_uint_32 BrMaterialFindMany(const char* pattern, br_material** materials, int max)**

*Arguments:* **const char * pattern**

Search pattern.

**br_material ** materials**

A pointer to an array of pointers to materials.

**int max**

Maximum number of materials to find.

*Result:* **br_uint_32**

Returns the number of materials found. The pointer array is filled with pointers to the found materials.

*See Also:* **BrMaterialFind()** 165, **BrMaterialFindHook()** 166

---

# BrMaterialFindHook()

*Description:* Functions to set up a call-back.

*Declaration:* **br_material_find_cbfn* BrMaterialFindHook(br_material_find_cbfn* hook)**

*Arguments:* **br_material_find_cbfn * hook**

A pointer to a call-back function.

*Effects:* If **BrMaterialFind()** 165 is unsuccessful and a call-back has been set up, the call-back is passed the search pattern as its only argument. The call-back should then return a pointer to a substitute or default item.

For example, a call-back could be set up to return a default material if the desired material cannot be found in the registry.

The function **BrMaterialFindFailedLoad()** 167 is provided and will probably be sufficient in many cases.

*Result:* **br_material_find_cbfn ***

Returns a pointer to the old call-back function.

*Example:*

```
br_material BR_CALLBACK * test_callback(const char* pattern)
{  br_material* default_material;
...
   return(default_material);
}
...
{  br_material* material;
...
   BrMaterialFindHook(&test_callback);
   material = BrMaterialFind("non_existent_material");
```

```
}
```

*See Also:* **BrMaterialFindFailedLoad()** 167

## BrMaterialFindFailedLoad()

*Description:* This function is provided as a suitable function to supply to **BrMaterialFindHook()** 166.

*Declaration:* **br_material* BrMaterialFindFailedLoad(const char* name)**

*Arguments:* **const char * name**

The name supplied to **BrMaterialFind()** 165.

*Effects:* Attempts to load the material from the filing system using `name` as the filename. Searches current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined). If successful, sets this name as the identifier of the loaded material and adds the material to the registry.

*Result:* **br_material ***

Returns a pointer to the material, if found, else `NULL`.

*Example:*

```
BrMaterialFindHook(BrMaterialFindFailedLoad);
```

# Import & Export

## BrMaterialFileCount()

*Description:* Locate a given file and count the number of materials in it.

*Declaration:* **br_uint_32 BrMaterialFileCount(const char* filename, br_uint_16* num)**

*Arguments:* **const char * filename**

Name of the file containing the materials to count.

**br_uint_16 * num**

Pointer to the variable in which to store the number of materials counted in the file. If `NULL`, the file will still be located and appropriate success returned, but no count will be made.

*Effects:* Searches for `filename`, if no path specified with file looks in current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined). If a file is found, will count the number of materials stored in it.

*Result:* **br_uint_32**

Returns zero if the file was found (even if it is not a materials file), non-zero otherwise.

---

# BrMaterialLoad()

*Description:* Load a material. Note that it is not added to the registry.

*Declaration:* **br_material* BrMaterialLoad(const char* filename)**

*Arguments:* **const char * filename**

Name of the file containing the material to load.

*Effects:* Searches for `filename`, if no path specified with file looks in current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined).

*Result:* **br_material ***

Returns a pointer to the loaded material, or `NULL` if unsuccessful.

*See Also:* **BrMaterialLoadMany()** 168, **BrMaterialSave()** 170, **BrMaterialAdd()** 161.

---

# BrMaterialLoadMany()

*Description:* Load a number of materials. Note that they are not added to the registry.

*Declaration:* **br_uint_32 BrMaterialLoadMany(const char* filename, br_material** materials, br_uint_16 num)**

*Arguments:* **const char * filename**

Name of the file containing the materials to load.

**br_material ** materials**

A non-`NULL` pointer to an array of pointers to materials.

**br_uint_16 num**

Maximum number of materials to load.

*Effects:* Searches for `filename`, if no path specified with file looks in current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined).

*Result:* **br_uint_32**

Returns the number of materials loaded successfully. The pointer array supplied, is filled with pointers to the loaded materials.

*See Also:* To determine how many materials are stored in a file see **BrMaterialFileCount()** 167.

## BrFmtScriptMaterialLoad()

*Description:* Load a material from a material script. Note that all maps and tables in a script should be already loaded and registered. If not, this function can be combined with **BrMapFindHook()** 295 and **BrTableFindHook()** 299 to facilitate rapid setup of materials with textures.

*Declaration:* **br_material* BrFmtScriptMaterialLoad(const char* filename)**

*Arguments:* **const char * filename**

Name of the file containing the material script.

*Effects:* Searches for `filename`, if no path specified with file looks in current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined).

*Result:* **br_material ***

Returns a pointer to the loaded material, or `NULL` if unsuccessful.

*Example:* Material scripts are formatted as follows:

```
# Comment
#
# Fields may be given in any order or omitted
# (a sensible default will be supplied)
# Extra white spaces are ignored

material=
[  name = "foo";
   flags=
   [
   light,prelit,smooth,environment,environment_local,perspective,
      decal,always_visible,two_sided,force_z_0
   ];
   colour = [10,10,20];
   ambient = 0.10;
   diffuse = 0.70;
   specular = 0.40;
   power = 30;
   map_transform = [[1,0], [0,1], [0,0]];
   index_base = 0;
   index_range = 0;
   colour_map = "brick";
   index_shade = "shade.tab";
];
```

## BrFmtScriptMaterialLoadMany()

*Description:* Load a number of materials from a material script.

*Declaration:*
```
br_uint_32
BrFmtScriptMaterialLoadMany(const char* filename,
br_material** materials, br_uint_16 num)
```

*Arguments:* **const char * filename**

Name of the file containing a number of concatenated material script entries.

**br_material ** materials**

A non-NULL pointer to an array of pointers to materials.

**br_uint_16 num**

Maximum number of materials to load.

*Effects:* Searches for filename, if no path specified with file looks in current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined).

*Result:* **br_uint_32**

Return the number of materials loaded successfully. The pointer array if supplied, is filled with pointers to the loaded materials.

## BrMaterialSave()

*Description:* Save a material to a file.

*Declaration:*
```
br_uint_32 BrMaterialSave(const char* filename,
const br_material* material)
```

*Arguments:* **const char * filename**

Name of the file to save the material to.

**const br_material * material**

A pointer to a material.

*Effects:* Writes the material to a file*.

*Result:* **br_uint_32**

Returns NULL if the material could not be saved.

*See Also:* **BrWriteModeSet()** 64

## BrMaterialSaveMany()

*Description:* Save a number of materials to a file.

* Any existing file of the same name is overwritten.

*Declaration:* **br_uint_32 BrMaterialSaveMany(const char* filename, const br_material* const* materials, br_uint_16 num)**

*Arguments:* **const char * filename**

Name of the file to save the materials to.

**const br_material * const * materials**

A pointer to an array of pointers to materials. If NULL, all registered materials are saved (irrespective of num).

**br_uint_16 num**

Number of materials to save.

*Effects:* Writes the materials to a file*.

*Result:* **br_uint_32**

Returns the number of materials saved successfully.

*See Also:* **BrWriteModeSet()** 64

## Platform Specific

The supported combinations of material flags vary from platform to platform. The supported dimensions of texture maps are also platform dependent.

* Any existing file of the same name is overwritten.

# br_material_enum_cbfn

## The Call-Back Function

This type defines a function, supplied to **BrMaterialEnum()** 164, and to be called by it for a selection of materials.

### *The* ***typedef***

(See *fwproto.h* for a precise declaration)

**br_uint_32 br_material_enum_cbfn(br_material*, void *)***Enumerator*

## Specification

---

## CBFnMaterialEnum()

*Description:* An application defined call-back function accepting a material and an application supplied argument (as supplied to **BrMaterialEnum()** 164).

*Declaration:* **br_uint_32 BR_CALLBACK CBFnMaterialEnum(br_material* mat, void* arg)**

*Arguments:* **br_material * mat**

One of the resource classes selected by **BrMaterialEnum()** 164.

**void * arg**

The argument supplied to **BrMaterialEnum()** 164.

*Preconditions:* BRender has completed initialisation.

*Effects:* Application defined. Avoid adding or removing materials within this function.

*Result:* **br_uint_32**

Any non-zero value will terminate the enumeration and be returned by **BrMaterialEnum()** 164. Return zero to continue the enumeration.

*See Also:* **BrMaterialEnum()** 164, **BrMaterialFind()** 165.

---

# br_material_find_cbfn

## The Call-Back Function

This type defines a function, registered with **BrMaterialFindHook()** [166], to be called when **BrMaterialFind()** [165] or **BrMaterialFindMany()** [165] fail to find any material.

### *The* ***typedef***

(See *fwproto.h* for a precise declaration)

```
br_material*            br_material_find_cbfn(const char*)
```

*Find (when* **BrMaterialFind()** *fails)*

## Specification

---

## CBFnMaterialFind()

*Description:* An application defined call-back function used when **BrMaterialFind()** [165] or **BrMaterialFindMany()** [165] fail.

*Declaration:* **br_material* BR_CALLBACK CBFnMaterialFind(const char* name)**

*Arguments:* **const char * name**

The search pattern supplied to **BrMaterialFind()** [165] or **BrMaterialFindMany()** [165] that did not match any material.

*Preconditions:* BRender has completed initialisation. No material has an identifier that successfully matches the search pattern.

*Effects:* Application defined.

*Result:* **br_material ***

Either return an existing material that is deemed appropriate for the search pattern, or NULL if there isn't one. This value will be returned by **BrMaterialFind()** [165] or **BrMaterialFindMany()** [165].

*Remarks:* This could either be used to supply a default material or to create a material. If materials were created on demand, then this function could search another list of available materials (but not yet created) and see if the pattern matched any of them, if it did, one of them could be registered and returned. Note that there is no way to supply more than one material.

*See Also:* **BrMaterialFind()** [165], **BrMaterialFindMany()** [165], **BrMaterialFindHook()** [166], **BrMaterialFindFailedLoad()** [167].

---

# br_matrix23

## The Structure

A two column, three row, scalar array, used as a 2D affine matrix, typically for texture map transformations (translation, scaling, shearing, rotation). Functions are provided to allow it be used as though it were an integral type. It has the following form:

$$\begin{pmatrix} m_{00} & m_{01} \\ m_{10} & m_{11} \\ m_{20} & m_{21} \end{pmatrix}$$

Note that this is effectively used as a 3×3 matrix, but omitting the redundant, third column for storage purposes. Thus:

$$\begin{pmatrix} m_{00} & m_{01} & 0 \\ m_{10} & m_{11} & 0 \\ m_{20} & m_{21} & 1 \end{pmatrix}$$

It is applied to homogenous 2D co-ordinates, which similarly omit the third element for sake of economy.

It can be noted that the bottom row has a translational effect. Also note, that the matrix determinant represents the area change effected.

### *The* ***typedef***

(See *matrix.h* for precise declaration and ordering)

| | | |
|---|---|---|
| **br_scalar** | **m[3][2]** | ***Three rows of two columns*** |

### *Related Structures*

See **br_material**[154].

## Members

### br_scalar m[3][2]

Each element of the matrix can be freely and individually accessed.

This matrix can also be thought of as an array of three **br_vector2**[359] structures, e.g. br_vector2 m[3]. Thus m[row] can be cast as (br_vector2*).

# Arithmetic

## BrMatrix23Mul()

*Description:* Multiply two matrices together and place the result in a third matrix. Equivalent to the expression:

$$\mathbf{A} \Leftarrow \mathbf{BC}$$

*Declaration:* **`void BrMatrix23Mul(br_matrix34* A, const br_matrix34* B, const br_matrix34* C)`**

*Arguments:* **`br_matrix23 * A`**

A pointer to the destination matrix (must be different from both sources).

**`const br_matrix23 * B`**

Pointer to the left hand source matrix.

**`const br_matrix23 * C`**

Pointer to the right hand source matrix.

*Remarks:* The result in `A` is equivalent to the following:

$$\begin{pmatrix} b_{00} & b_{01} & 0 \\ b_{10} & b_{11} & 0 \\ b_{20} & b_{21} & 1 \end{pmatrix}\begin{pmatrix} c_{00} & c_{01} & 0 \\ c_{10} & c_{11} & 0 \\ c_{20} & c_{21} & 1 \end{pmatrix} \equiv \begin{pmatrix} b_{00}c_{00} + b_{01}c_{10} & b_{00}c_{01} + b_{01}c_{11} & 0 \\ b_{10}c_{00} + b_{11}c_{10} & b_{10}c_{01} + b_{11}c_{11} & 0 \\ b_{20}c_{00} + b_{21}c_{10} + c_{20} & b_{20}c_{01} + b_{21}c_{11} + c_{21} & 1 \end{pmatrix}$$

*See Also:* **`BrMatrix23Pre()`** 179, **`BrMatrix23Post()`** 182

## BrMatrix23Inverse()

*Description:* Compute the inverse of the supplied matrix. Equivalent to the expression:

$$\mathbf{A} \Leftarrow \mathbf{B}^{-1}$$

*Declaration:* **`br_scalar BrMatrix23Inverse(br_matrix23* A, const br_matrix23* B)`**

*Arguments:* **`br_matrix23 * A`**

A pointer to the destination matrix (must be different from source).

**`const br_matrix23 * B`**

A pointer to the source matrix.

*Result:* **`br_scalar`**

If the inverse exists, the determinant of the source matrix is returned. If there is no inverse, scalar zero is returned.

*Remarks:* The result in `A` is equivalent to the following:

$$\begin{pmatrix} b_{00} & b_{01} & 0 \\ b_{10} & b_{11} & 0 \\ b_{20} & b_{21} & 1 \end{pmatrix}^{-1} \equiv \begin{pmatrix} \frac{b_{11}}{|\mathbf{B}|} & \frac{-b_{01}}{|\mathbf{B}|} & 0 \\ \frac{-b_{10}}{|\mathbf{B}|} & \frac{b_{00}}{|\mathbf{B}|} & 0 \\ \frac{(b_{10}b_{21} - b_{11}b_{20})}{|\mathbf{B}|} & \frac{(b_{01}b_{20} - b_{00}b_{21})}{|\mathbf{B}|} & 1 \end{pmatrix}$$

*See Also:* **`BrMatrix23LPInverse()`** 176.

# BrMatrix23LPInverse()

*Description:* Compute the inverse of the supplied length preserving* transformation matrix. The resulting matrix is undefined for non-length preserving matrices.

Equivalent to the expression:

$$\mathbf{A}_{LP} \Leftarrow \mathbf{B}_{LP}^{-1}$$

*Declaration:*
```
void BrMatrix23LPInverse(br_matrix23* A,
const br_matrix23* B)
```

*Arguments:* **`br_matrix23 * A`**

A pointer to the destination matrix (must be different from source).

**`const br_matrix23 * B`**

A pointer to the source matrix.

*Remarks:* The result in `A` is equivalent to the following:

$$\begin{pmatrix} b_{00} & b_{01} & 0 \\ b_{10} & b_{11} & 0 \\ b_{20} & b_{21} & 1 \end{pmatrix}_{LP}^{-1} \equiv \begin{pmatrix} b_{11} & -b_{01} & 0 \\ -b_{10} & b_{00} & 0 \\ b_{10}b_{21} - b_{11}b_{20} & b_{01}b_{20} - b_{00}b_{21} & 1 \end{pmatrix}_{LP}$$

*See Also:* **`BrMatrix23Inverse()`** 175.

# BrMatrix23ApplyP()

*Description:* Applies a transform to a 2D point. Equivalent to the expression:

$$P_A \Leftarrow P_B\mathbf{C}$$

* Note that length preserving also applies to the sign of lengths, not just their magnitude, i.e. a reflection is not length preserving in this case.

*Declaration:* **void BrMatrix23ApplyP(br_vector2* A, const br_vector2* B, const br_matrix23* C)**

*Arguments:* **br_vector2 * A**

A pointer to the destination vector (must be different from source, and not part of transform), to hold the transformed point.

**const br_vector2 * B**

A pointer to the source vector, holding the point to be transformed.

**const br_matrix23 * C**

A pointer to the transform matrix to be applied.

*Remarks:* The result in `A` is equivalent to the following:

$$(x_B \quad y_B \quad 1)\begin{pmatrix} c_{00} & c_{01} & 0 \\ c_{10} & c_{11} & 0 \\ c_{20} & c_{21} & 1 \end{pmatrix} \equiv (x_B c_{00} + y_B c_{10} + c_{20} \quad x_B c_{01} + y_B c_{11} + c_{21} \quad 1)$$

# BrMatrix23ApplyV()

*Description:* Applies a transform to a 2D vector, i.e. as for a point but without translation components (a vector has no location). Equivalent to the expression:

$$\mathbf{v}_A \Leftarrow \mathbf{v}_B\mathbf{C}$$

*Declaration:* **void BrMatrix23ApplyV(br_vector2* A, const br_vector2* B, const br_matrix23* C)**

*Arguments:* **br_vector2 * A**

A pointer to the destination vector (must be different from source, and not part of transform), to hold the transformed vector.

**const br_vector2 * B**

A pointer to the source vector, holding the vector to be transformed.

**const br_matrix23 * C**

A pointer to the transform matrix to be applied.

*Remarks:* The result in `A` is equivalent to the following:

$$(x_B \quad y_B \quad 0)\begin{pmatrix} c_{00} & c_{01} & 0 \\ c_{10} & c_{11} & 0 \\ c_{20} & c_{21} & 1 \end{pmatrix} \equiv (x_B c_{00} + y_B c_{10} \quad x_B c_{01} + y_B c_{11} \quad 0)$$

## BrMatrix23TApplyP()

*Description:* Applies a transposed transform to a 2D point. Equivalent to the expression:

$$P_A \Leftarrow P_B \mathbf{C}^t$$

*Declaration:* **`void BrMatrix23TApplyP(br_vector2* A, const br_vector2* B, const br_matrix23* C)`**

*Arguments:* **`br_vector2 * A`**

A pointer to the destination vector (must be different from source, and not part of transform), to hold the transformed point.

**`const br_vector2 * B`**

A pointer to the source vector, holding the point to be transformed.

**`const br_matrix23 * C`**

A pointer to the transform matrix to be applied transposed – the translation elements are presumed zero or irrelevant.

*Remarks:* The result in `A` is equivalent to the following:

$$\begin{pmatrix} c_{00} & c_{01} & 0 \\ c_{10} & c_{11} & 0 \\ - & - & 1 \end{pmatrix} \begin{pmatrix} x_B \\ y_B \\ 1 \end{pmatrix} \equiv \begin{pmatrix} c_{00}x_B + c_{01}y_B \\ c_{10}x_B + c_{11}y_B \\ 1 \end{pmatrix}$$

## BrMatrix23TApplyV()

*Description:* Applies a transposed transform to a 2D vector, i.e. as for a point but without translation components (a vector has no location). Equivalent to the expression:

$$\mathbf{v}_A \Leftarrow \mathbf{v}_B \mathbf{C}^t$$

*Declaration:* **`void BrMatrix23TApplyV(br_vector2* A, const br_vector2* B, const br_matrix23* C)`**

*Arguments:* **`br_vector2 * A`**

A pointer to the destination vector (must be different from source, and not part of transform), to hold the transformed vector.

**`const br_vector2 * B`**

A pointer to the source vector, holding the vector to be transformed.

**`const br_matrix23 * C`**

A pointer to the transform matrix to be applied transposed – the translation elements are presumed zero or irrelevant.

*Remarks:* The result in `A` is equivalent to the following:

$$\begin{pmatrix} c_{00} & c_{01} & 0 \\ c_{10} & c_{11} & 0 \\ - & - & 1 \end{pmatrix}\begin{pmatrix} x_B \\ y_B \\ 0 \end{pmatrix} \equiv \begin{pmatrix} c_{00}x_B + c_{01}y_B \\ c_{10}x_B + c_{11}y_B \\ 0 \end{pmatrix}$$

---

# BrMatrix23Pre()

*Description:* Pre-multiply one matrix by another. Equivalent to the expression:

$$\mathbf{A} \Leftarrow \mathbf{BA}$$

*Declaration:* **`void BrMatrix23Pre(br_matrix23* A, const br_matrix23* B)`**

*Arguments:* **`br_matrix23 * A`**

A pointer to the subject matrix (may be same as `B`).

**`const br_matrix23 * B`**

A pointer to the pre-multiplying matrix.

*Remarks:* The result in `A` is equivalent to the following:

$$\begin{pmatrix} b_{00} & b_{01} & 0 \\ b_{10} & b_{11} & 0 \\ b_{20} & b_{21} & 1 \end{pmatrix}\begin{pmatrix} a_{00} & a_{01} & 0 \\ a_{10} & a_{11} & 0 \\ a_{20} & a_{21} & 1 \end{pmatrix} \equiv \begin{pmatrix} b_{00}a_{00} + b_{01}a_{10} & b_{00}a_{01} + b_{01}a_{11} & 0 \\ b_{10}a_{00} + b_{11}a_{10} & b_{10}a_{01} + b_{11}a_{11} & 0 \\ b_{20}a_{00} + b_{21}a_{10} + a_{20} & b_{20}a_{01} + b_{21}a_{11} + a_{21} & 1 \end{pmatrix}$$

*See Also:* **`BrMatrix23Post()`** 182, **`BrMatrix23Mul()`** 175

---

# BrMatrix23PreTranslate()

*Description:* Pre-multiply a matrix by a translation transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{T}_{xy}\mathbf{M}$$

*Declaration:* **`void BrMatrix23PreTranslate(br_matrix23* mat, br_scalar dx, br_scalar dy)`**

*Arguments:* **`br_matrix23 * mat`**

A pointer to the subject matrix.

**`br_scalar dx`**

The x axis component used to form the translation matrix.

**`br_scalar dy`**

The y axis component used to form the translation matrix.

*Remarks:* The result in `mat` is equivalent to the following:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \delta_x & \delta_y & 1 \end{pmatrix}\begin{pmatrix} m_{00} & m_{01} & 0 \\ m_{10} & m_{11} & 0 \\ m_{20} & m_{21} & 1 \end{pmatrix} \equiv \begin{pmatrix} m_{00} & m_{01} & 0 \\ m_{00} & m_{01} & 0 \\ \delta_x m_{00} + \delta_y m_{10} + m_{20} & \delta_x m_{01} + \delta_y m_{11} + m_{21} & 1 \end{pmatrix}$$

*See Also:* **BrMatrix23PostTranslate()** 183, **BrMatrix23Translate()** 187

---

# BrMatrix23PreScale()

*Description:* Pre-multiply a matrix by a scaling transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{S}_{xy}\mathbf{M}$$

*Declaration:* **void BrMatrix23PreScale(br_matrix23* mat, br_scalar sx, br_scalar sy)**

*Arguments:* **br_matrix23 * mat**

A pointer to the subject matrix.

**br_scalar sx**

Scaling component along the x axis.

**br_scalar sy**

Scaling component along the y axis.

*Remarks:* The result in `mat` is equivalent to the following:

$$\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} m_{00} & m_{01} & 0 \\ m_{10} & m_{11} & 0 \\ m_{20} & m_{21} & 1 \end{pmatrix} \equiv \begin{pmatrix} s_x m_{00} & s_x m_{01} & 0 \\ s_y m_{10} & s_y m_{11} & 0 \\ m_{20} & m_{21} & 1 \end{pmatrix}$$

*See Also:* **BrMatrix23PostScale()** 183, **BrMatrix23Scale()** 188

---

# BrMatrix23PreShearX()

*Description:* Pre-multiply a matrix by an x invariant shearing transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{Z}_X\mathbf{M}$$

*Declaration:* **void BrMatrix23PreShearX(br_matrix23* mat, br_scalar sy)**

*Arguments:* **br_matrix23 * mat**

A pointer to the subject matrix.

**br_scalar sy**

Shear factor by which the x co-ordinate is included in the transformed y co-ordinate.

*Remarks:* The result in `mat` is equivalent to the following:

$$\begin{pmatrix} 1 & \sigma_y & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} m_{00} & m_{01} & 0 \\ m_{10} & m_{11} & 0 \\ m_{20} & m_{21} & 1 \end{pmatrix} \equiv \begin{pmatrix} m_{00}+\sigma_y m_{10} & m_{01}+\sigma_y m_{11} & 0 \\ m_{10} & m_{11} & 0 \\ m_{20} & m_{21} & 1 \end{pmatrix}$$

*See Also:* **BrMatrix23PostShearX()** 184, **BrMatrix23ShearX()** 180

---

# BrMatrix23PreShearY()

*Description:* Pre-multiply a matrix by a y invariant shearing transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{Z}_Y\mathbf{M}$$

*Declaration:* **void BrMatrix23PreShearY(br_matrix23* mat, br_scalar sx)**

*Arguments:* **br_matrix23 * mat**

A pointer to the subject matrix.

**br_scalar sx**

Shear factor by which the y co-ordinate is included in the transformed x co-ordinate.

*Remarks:* The result in `mat` is equivalent to the following:

$$\begin{pmatrix} 1 & 0 & 0 \\ \sigma_x & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} m_{00} & m_{01} & 0 \\ m_{10} & m_{11} & 0 \\ m_{20} & m_{21} & 1 \end{pmatrix} \equiv \begin{pmatrix} m_{00} & m_{01} & 0 \\ \sigma_x m_{00}+m_{10} & \sigma_x m_{01}+m_{11} & 0 \\ m_{20} & m_{21} & 1 \end{pmatrix}$$

*See Also:* **BrMatrix23PostShearY()** 184, **BrMatrix23ShearY()** 189

## BrMatrix23PreRotate()

*Description:* Pre-multiply a matrix by a rotational transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{R}_{\theta_Z}\mathbf{M}$$

*Declaration:* **void BrMatrix23PreRotate(br_matrix23* mat, br_angle rz)**

*Arguments:* **br_matrix23 * mat**

A pointer to the subject matrix.

**br_angle rz**

The clockwise angle about the z axis used to form the rotation matrix.

*Remarks:* The result in `mat` is equivalent to the following:

$$\begin{pmatrix} \cos\theta_z & \sin\theta_z & 0 \\ -\sin\theta_z & \cos\theta_z & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} m_{00} & m_{01} & 0 \\ m_{10} & m_{11} & 0 \\ m_{20} & m_{21} & 1 \end{pmatrix} \equiv \begin{pmatrix} m_{00}\cos\theta_z + m_{10}\sin\theta_z & m_{01}\cos\theta_z + m_{11}\sin\theta_z & 0 \\ -m_{00}\sin\theta_z + m_{10}\cos\theta_z & -m_{01}\sin\theta_z + m_{11}\cos\theta_z & 0 \\ m_{20} & m_{21} & 1 \end{pmatrix}$$

*See Also:* **BrMatrix23PostRotate()** 185, **BrMatrix23Rotate()** 189

## BrMatrix23Post()

*Description:* Post-multiply one matrix by another. Equivalent to the expression:

$$\mathbf{A} \Leftarrow \mathbf{AB}$$

*Declaration:* **void BrMatrix23Post(br_matrix23* A, const br_matrix23* B)**

*Arguments:* **br_matrix23 * A**

A pointer to the subject matrix (may be same as `B`).

**const br_matrix23 * B**

A pointer to the post-multiplying matrix.

*Remarks:* The result in `A` is equivalent to the following:

$$\begin{pmatrix} a_{00} & a_{01} & 0 \\ a_{10} & a_{11} & 0 \\ a_{20} & a_{21} & 1 \end{pmatrix}\begin{pmatrix} b_{00} & b_{01} & 0 \\ b_{10} & b_{11} & 0 \\ b_{20} & b_{21} & 1 \end{pmatrix} \equiv \begin{pmatrix} a_{00}b_{00} + a_{01}b_{10} & a_{00}b_{01} + a_{01}b_{11} & 0 \\ a_{10}b_{00} + a_{11}b_{10} & a_{10}b_{01} + a_{11}b_{11} & 0 \\ a_{20}b_{00} + a_{21}b_{10} + b_{20} & a_{20}b_{01} + a_{21}b_{11} + b_{21} & 1 \end{pmatrix}$$

*See Also:* **BrMatrix23Pre()** 179, **BrMatrix23Mul()** 175.

## BrMatrix23PostTranslate()

*Description:* Post-multiply a matrix by a translation transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{MT}_{xy}$$

*Declaration:* **void BrMatrix23PostTranslate(br_matrix23* mat, br_scalar dx, br_scalar dy)**

*Arguments:* **br_matrix23 * mat**

A pointer to the subject matrix.

**br_scalar dx**

The x axis component used to form the translation matrix.

**br_scalar dy**

The y axis component used to form the translation matrix.

*Remarks:* The result in `mat` is equivalent to the following:

$$\begin{pmatrix} m_{00} & m_{01} & 0 \\ m_{10} & m_{11} & 0 \\ m_{20} & m_{21} & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \delta_x & \delta_y & 1 \end{pmatrix} \equiv \begin{pmatrix} m_{00} & m_{01} & 0 \\ m_{10} & m_{11} & 0 \\ m_{20}+\delta_x & m_{21}+\delta_y & 1 \end{pmatrix}$$

*See Also:* **BrMatrix23PreTranslate()** 179, **BrMatrix23Translate()** 187.

## BrMatrix23PostScale()

*Description:* Post-multiply a matrix by a scaling transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{MS}_{xy}$$

*Declaration:* **void BrMatrix23PostScale(br_matrix23* mat, br_scalar sx, br_scalar sy)**

*Arguments:* **br_matrix23 * mat**

A pointer to the subject matrix.

**br_scalar sx**

Scaling component along the x axis.

**br_scalar sy**

Scaling component along the y axis.

*Remarks:* The result in `mat` is equivalent to the following:

$$\begin{pmatrix} m_{00} & m_{01} & 0 \\ m_{10} & m_{11} & 0 \\ m_{20} & m_{21} & 1 \end{pmatrix}\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \equiv \begin{pmatrix} m_{00}s_x & m_{01}s_y & 0 \\ m_{10}s_x & m_{11}s_y & 0 \\ m_{20}s_x & m_{21}s_y & 1 \end{pmatrix}$$

*See Also:* **BrMatrix23PreScale()** 180, **BrMatrix23Scale()** 188.

---

# BrMatrix23PostShearX()

*Description:* Post-multiply a matrix by an x invariant shearing transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{MZ}_X$$

*Declaration:* **void BrMatrix23PostShearX(br_matrix23* mat, br_scalar sy)**

*Arguments:* **br_matrix23 * mat**

A pointer to the subject matrix.

**br_scalar sy**

Shear factor by which the x co-ordinate is included in the transformed y co-ordinate.

*Remarks:* The result in `mat` is equivalent to the following:

$$\begin{pmatrix} m_{00} & m_{01} & 0 \\ m_{10} & m_{11} & 0 \\ m_{20} & m_{21} & 1 \end{pmatrix}\begin{pmatrix} 1 & \sigma_y & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \equiv \begin{pmatrix} m_{00} & m_{00}\sigma_y + m_{01} & 0 \\ m_{10} & m_{10}\sigma_y + m_{11} & 0 \\ m_{20} & m_{20}\sigma_y + m_{21} & 1 \end{pmatrix}$$

*See Also:* **BrMatrix23PreShearX()** 180, **BrMatrix23ShearX()** 188.

---

# BrMatrix23PostShearY()

*Description:* Post-multiply a matrix by a y invariant shearing transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{MZ}_Y$$

*Declaration:* **void BrMatrix23PostShearY(br_matrix23* mat, br_scalar sx)**

*Arguments:* **br_matrix23 * mat**

A pointer to the subject matrix.

**br_scalar sx**

Shear factor by which the y co-ordinate is included in the transformed x co-ordinate.

*Remarks:* The result in `mat` is equivalent to the following:

$$\begin{pmatrix} m_{00} & m_{01} & 0 \\ m_{10} & m_{11} & 0 \\ m_{20} & m_{21} & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ \sigma_x & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \equiv \begin{pmatrix} m_{00}+m_{01}\sigma_x & m_{01} & 0 \\ m_{10}+m_{11}\sigma_x & m_{11} & 0 \\ m_{20}+m_{21}\sigma_x & m_{21} & 1 \end{pmatrix}$$

*See Also:* **BrMatrix23PreShearY()** 184, **BrMatrix23ShearY()** 189.

---

# BrMatrix23PostRotate()

*Description:* Post-multiply a matrix by a rotational transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{M}\mathbf{R}_{\theta_Z}$$

*Declaration:* **void BrMatrix23PostRotate(br_matrix23* mat, br_angle rz)**

*Arguments:* **br_matrix23 * mat**

A pointer to the subject matrix.

**br_angle rz**

The clockwise angle about the z axis used to form the rotation matrix.

*Remarks:* The result in `mat` is equivalent to the following:

$$\begin{pmatrix} m_{00} & m_{01} & 0 \\ m_{10} & m_{11} & 0 \\ m_{20} & m_{21} & 1 \end{pmatrix}\begin{pmatrix} \cos\theta_z & \sin\theta_z & 0 \\ -\sin\theta_z & \cos\theta_z & 0 \\ 0 & 0 & 1 \end{pmatrix} \equiv \begin{pmatrix} m_{00}\cos\theta_z - m_{01}\sin\theta_z & m_{00}\sin\theta_z + m_{01}\cos\theta_z & 0 \\ m_{10}\cos\theta_z - m_{11}\sin\theta_z & m_{10}\sin\theta_z + m_{11}\cos\theta_z & 0 \\ m_{20}\cos\theta_z - m_{21}\sin\theta_z & m_{20}\sin\theta_z + m_{21}\cos\theta_z & 1 \end{pmatrix}$$

*See Also:* **BrMatrix23PreRotate()** 182, **BrMatrix23Rotate()** 189.

---

## Copy/Assign

Although copy by structure assignment currently works, use **BrMatrix23Copy()** 185 to ensure compatibility.

---

# BrMatrix23Copy()

*Description:* Copy a matrix. Equivalent to the expression:

$$\mathbf{A} \Leftarrow \mathbf{B}$$

*Declaration:* **void BrMatrix23Copy(br_matrix23* A, const br_matrix23* B)**

*Arguments:* **br_matrix23 * A**

A pointer to the destination matrix (may be the same as source – though redundant).

**const br_matrix23 * B**

A pointer to the source matrix.

---

## Access & Maintenance

Members may be freely accessed. Maintenance is only required for length preserving matrices that have been modified.

---

# BrMatrix23LPNormalise()

*Description:* Normalise a length preserving* matrix. Equivalent to the expression:

$$A_{LP} \Leftarrow \mathrm{Norm}(\mathbf{B}_{-LP})$$

*Declaration:*
```
void BrMatrix23LPNormalise(br_matrix23* A,
const br_matrix23* B)
```

*Arguments:* **br_matrix23 * A**

A pointer to the destination matrix, which must not point to the source matrix.

**const br_matrix23 * B**

A pointer to the source matrix.

*Effects:* The destination matrix is the source matrix adjusted so that it represents a length preserving transformation.

*Remarks:* This function is typically applied to a length preserving matrix which has undergone a long sequence of operations, to ensure that the final matrix is still truly length-preserving.

---

## Referencing & Lifetime

This structure may be freely referenced, though take care if there is potential to supply the same matrix as more than one argument to the same function.

---

* Note that length preserving also applies to the sign of lengths, not just their magnitude, i.e. a reflection is not length preserving in this case.

## Initialisation

No static initialisers are provided, however, three `BR_VECTOR2()` macros would serve as well.

All other initialisation should use the functions **`BrMatrix23Identity()`**[187], **`BrMatrix23Translate()`**[187], **`BrMatrix23Scale()`**[188], **`BrMatrix23Shear[X|Y]()`**[188|189], **`BrMatrix23Rotate()`**[189], or **`BrMatrix23Copy()`**[185].

---

# BrMatrix23Identity()

*Description:* Set the specified matrix to the identity transformation matrix. Equivalent to:

$$\mathbf{M} \Leftarrow \mathbf{I} \equiv \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

*Declaration:* **`void BrMatrix23Identity(br_matrix23* mat)`**

*Arguments:* **`br_matrix23 * mat`**

A pointer to the destination matrix.

*Effects:* Stores the identity matrix at the destination.

---

# BrMatrix23Translate()

*Description:* Set the specified matrix to a matrix representing a specific translation. Equivalent to:

$$\mathbf{M} \Leftarrow \mathbf{T}_{xy} \equiv \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \delta_x & \delta_y & 1 \end{pmatrix}$$

*Declaration:* **`void BrMatrix23Translate(br_matrix23* mat, br_scalar dx, br_scalar dy)`**

*Arguments:* **`br_matrix23 * mat`**

A pointer to the destination matrix.

**`br_scalar dx`**

Translation component along the x axis.

**`br_scalar dy`**

Translation component along the y axis.

*See Also:* **`BrMatrix23PreTranslate()`**[179], **`BrMatrix23PostTranslate()`**[183].

## BrMatrix23Scale()

*Description:* Set the specified matrix to a matrix representing a specific scaling. Equivalent to:

$$\mathbf{M} \Leftarrow \mathbf{S}_{xy} \equiv \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

*Declaration:* **void BrMatrix23Scale(br_matrix23* mat, br_scalar sx, br_scalar sy)**

*Arguments:* **br_matrix23 * mat**

A pointer to the destination matrix.

**br_scalar sx**

Scaling component along the x axis.

**br_scalar sy**

Scaling component along the y axis.

*See Also:* **BrMatrix23PreScale()** 180, **BrMatrix23PostScale()** 183.

## BrMatrix23ShearX()

*Description:* Set the specified matrix to a matrix representing a shear, invariant along the x axis. Thus values of y co-ordinates will be scaled in proportion to the value of the x co-ordinate. Equivalent to:

$$\mathbf{M} \Leftarrow \mathbf{Z}_X \equiv \begin{pmatrix} 1 & \sigma_y & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

*Declaration:* **void BrMatrix23ShearX(br_matrix23* mat, br_scalar sy)**

*Arguments:* **br_matrix23 * mat**

A pointer to the destination matrix.

**br_scalar sy**

Shear factor by which the x co-ordinate is included in the transformed y co-ordinate.

*See Also:* **BrMatrix23PreShearX()** 180, **BrMatrix23PostShearX()** 184.

## BrMatrix23ShearY()

*Description:* Set the specified matrix to a matrix representing a shear, invariant along the y axis. Thus values of x co-ordinates will be scaled in proportion to the value of the y co-ordinate. Equivalent to:

$$\mathbf{M} \Leftarrow \mathbf{Z}_Y \equiv \begin{pmatrix} 1 & 0 & 0 \\ \sigma_x & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

*Declaration:* **`void BrMatrix23ShearX(br_matrix23* mat, br_scalar sx)`**

*Arguments:* **`br_matrix23 * mat`**

A pointer to the destination matrix.

**`br_scalar sx`**

Shear factor by which the y co-ordinate is included in the transformed x co-ordinate.

*See Also:* **`BrMatrix23PreShearY()`** 181, **`BrMatrix23PostShearY()`** 184.

## BrMatrix23Rotate()

*Description:* Set the specified matrix to a matrix representing a rotation about the z axis though a specified angle. Equivalent to:

$$\mathbf{M} \Leftarrow \mathbf{R}_{\theta_Z} \equiv \begin{pmatrix} \cos\theta_Z & \sin\theta_Z & 0 \\ -\sin\theta_Z & \cos\theta_Z & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

*Declaration:* **`void BrMatrix23Rotate(br_matrix23* mat, br_angle rz)`**

*Arguments:* **`br_matrix23 * mat`**

A pointer to the destination matrix.

**`br_angle rz`**

Clockwise rotation about the z axis.

*See Also:* **`BrMatrix23PreRotate()`** 182, **`BrMatrix23PostRotate()`** 185.

# br_matrix34

## The Structure

A three column, four row, scalar array, used as a 3D affine matrix for general purpose 3D transformations (translation, scaling, shearing, rotation). Functions are provided to allow it be used as though it were an integral type. It has the following form:

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \\ m_{30} & m_{31} & m_{32} \end{pmatrix}$$

Note that this is effectively used as a 4×4 matrix, but omitting the redundant, fourth column for storage purposes. Thus:

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} & 0 \\ m_{10} & m_{11} & m_{12} & 0 \\ m_{20} & m_{21} & m_{22} & 0 \\ m_{30} & m_{31} & m_{32} & 1 \end{pmatrix}$$

It is applied to homogenous 3D co-ordinates, which similarly omit the fourth element for sake of economy.

It can be noted that the bottom row has a translational effect. Also note, that the matrix determinant represents the volume change effected.

### *The* ***typedef***

(See *matrix.h* for precise declaration and ordering)

| | | |
|---|---|---|
| **br_scalar** | **m[4][3]** | ***Four rows of three columns*** |

### *Related Functions*

#### Scene Modelling

See **BrActorToActorMatrix34()** 84.

#### Maths

See **BrMatrix4Pre34()** 222, **BrMatrix4Copy34()** 226.

Miscellaneous

See **BrMatrix34RollingBall()**[69].

*Related Structures*

See **br_matrix4**[217], **br_material**[154], **br_pick3d_cbfn**[275].

## Members

### br_scalar m[4][3]

Each element of the matrix can be freely and individually accessed.

This matrix can also be thought of as an array of four **br_vector3**[367] structures, e.g. `br_vector3 m[4]`. Thus `m[row]` can be cast as `(br_vector3*)`.

## Arithmetic

---

# BrMatrix34Mul()

*Description:* Multiply two matrices together and place the result in a third matrix. Equivalent to the expression:

$$\mathbf{A} \Leftarrow \mathbf{BC}$$

*Declaration:* **void BrMatrix34Mul(br_matrix34* A, const br_matrix34* B, const br_matrix34* C)**

*Arguments:* **br_matrix34 * A**

A pointer to the destination matrix (must be different from both sources).

**const br_matrix34 * B**

Pointer to the left hand source matrix.

**const br_matrix34 * C**

Pointer to the right hand source matrix.

*Remarks:* The result in `A` is equivalent to the following:

$$\begin{pmatrix} b_{00} & b_{01} & b_{02} & 0 \\ b_{10} & b_{11} & b_{12} & 0 \\ b_{20} & b_{21} & b_{22} & 0 \\ b_{30} & b_{31} & b_{32} & 1 \end{pmatrix} \begin{pmatrix} c_{00} & c_{01} & c_{02} & 0 \\ c_{10} & c_{11} & c_{12} & 0 \\ c_{20} & c_{21} & c_{22} & 0 \\ c_{30} & c_{31} & c_{32} & 1 \end{pmatrix} \equiv$$

$$\begin{pmatrix} b_{00}c_{00} + b_{01}c_{10} + b_{02}c_{20} & b_{00}c_{01} + b_{01}c_{11} + b_{02}c_{21} & b_{00}c_{02} + b_{01}c_{12} + b_{02}c_{22} & 0 \\ b_{10}c_{00} + b_{11}c_{10} + b_{12}c_{20} & b_{10}c_{01} + b_{11}c_{11} + b_{12}c_{21} & b_{10}c_{02} + b_{11}c_{12} + b_{12}c_{22} & 0 \\ b_{20}c_{00} + b_{21}c_{10} + b_{22}c_{20} & b_{20}c_{01} + b_{21}c_{11} + b_{22}c_{21} & b_{20}c_{02} + b_{21}c_{12} + b_{22}c_{22} & 0 \\ b_{30}c_{00} + b_{31}c_{10} + b_{32}c_{20} + c_{30} & b_{30}c_{01} + b_{31}c_{11} + b_{32}c_{21} + c_{31} & b_{30}c_{02} + b_{31}c_{12} + b_{32}c_{22} + c_{32} & 1 \end{pmatrix}$$

*See Also:* **`BrMatrix34Pre()`** 197, **`BrMatrix34Post()`** 202

---

# BrMatrix34Inverse()

*Description:* Compute the inverse of the supplied 3D affine matrix. Equivalent to the expression:

$$\mathbf{A} \Leftarrow \mathbf{B}^{-1}$$

*Declaration:*
```
br_scalar BrMatrix34Inverse(br_matrix34* A,
const br_matrix34* B)
```

*Arguments:* **`br_matrix34 * A`**

A pointer to the destination matrix (must be different from source).

**`const br_matrix34 * B`**

A pointer to the source matrix.

*Result:* **`br_scalar`**

If the inverse exists, the determinant of the source matrix is returned. If there is no inverse, scalar zero is returned.

*Remarks:* Remember that while an inverse may be obtained using `double` precision arithmetic, this does not necessarily mean that it can using the **`br_scalar`** 342 type. This difference is most marked between fixed and floating point BRender libraries.

*See Also:* **`BrMatrix34LPInverse()`** 193.

## BrMatrix34LPInverse()

*Description:* Compute the inverse of the supplied length preserving* transformation matrix. The resulting matrix is undefined for non-length preserving matrices.

Equivalent to the expression:

$$\mathbf{A}_{LP} \Leftarrow \mathbf{B}_{LP}^{-1}$$

*Declaration:*
```
void BrMatrix34LPInverse(br_matrix34* A,
const br_matrix34* B)
```

*Arguments:* **br_matrix34 * A**

A pointer to the destination matrix (must be different from source).

**const br_matrix34 * B**

A pointer to the source matrix.

*See Also:* **BrMatrix34Inverse()** 192.

## BrMatrix34Apply()

*Description:* Applies a transform to a 3D point which may have non-unity homogenous co-ordinates. Equivalent to the expression:

$$P_A \Leftarrow P_B\mathbf{C}$$

*Declaration:*
```
void BrMatrix34Apply(br_vector3* A, const br_vector4* B,
const br_matrix34* C)
```

*Arguments:* **br_vector3† * A**

A pointer to the destination vector (must be different from source, and not part of transform), to hold the transformed point.

**const br_vector4 * B**

A pointer to the source vector, holding the point to be transformed.

**const br_matrix34 * C**

A pointer to the transform matrix to be applied.

---

* Note that length preserving also applies to the sign of lengths, not just their magnitude, i.e. a reflection is not length preserving in this case.

† The reason the destination vector argument is a **br_vector3** and not a **br_vector4**, is because it is sometimes useful to supply a **br_vector3**, which would be quite invalid if the argument specified a **br_vector4**.

*Remarks:* Be aware that the fourth element of the resulting vector is only implicit, and if required must either be copied (if the destination is actually a **br_vector4**$_{376}$) or used to scale down the first three elements. This all depends on the purpose for which this function is called.

The result in `A` is equivalent to the following:

$$\begin{pmatrix} x_B & y_B & z_B & w_B \end{pmatrix} \begin{pmatrix} c_{00} & c_{01} & c_{02} & 0 \\ c_{10} & c_{11} & c_{12} & 0 \\ c_{20} & c_{21} & c_{22} & 0 \\ c_{30} & c_{31} & c_{32} & 1 \end{pmatrix} \equiv$$

$$\begin{pmatrix} x_B c_{00} + y_B c_{10} + z_B c_{20} + w_B c_{30} & x_B c_{01} + y_B c_{11} + z_B c_{21} + w_B c_{31} & x_B c_{02} + y_B c_{12} + z_B c_{22} + w_B c_{32} & \{w_B\} \end{pmatrix}$$

---

# BrMatrix34ApplyP()

*Description:* Applies a transform to a 3D point. Equivalent to the expression:

$$P_A \Leftarrow P_B \mathbf{C}$$

*Declaration:* **void BrMatrix34ApplyP(br_vector3* A, const br_vector3* B, const br_matrix34* C)**

*Arguments:* **br_vector3 * A**

A pointer to the destination vector (must be different from source, and not part of transform), to hold the transformed point.

**const br_vector3 * B**

A pointer to the source vector, holding the point to be transformed.

**const br_matrix34 * C**

A pointer to the transform matrix to be applied.

*Remarks:* The result in `A` is equivalent to the following:

$$\begin{pmatrix} x_B & y_B & z_B & 1 \end{pmatrix} \begin{pmatrix} c_{00} & c_{01} & c_{02} & 0 \\ c_{10} & c_{11} & c_{12} & 0 \\ c_{20} & c_{21} & c_{22} & 0 \\ c_{30} & c_{31} & c_{32} & 1 \end{pmatrix} \equiv$$

$$\begin{pmatrix} x_B c_{00} + y_B c_{10} + z_B c_{20} + c_{30} & x_B c_{01} + y_B c_{11} + z_B c_{21} + c_{31} & x_B c_{02} + y_B c_{12} + z_B c_{22} + c_{32} & 1 \end{pmatrix}$$

## BrMatrix34ApplyV()

*Description:* Applies a transform to a 3D vector, i.e. as for a point but without translation components (a vector has no location). Equivalent to the expression:

$$\mathbf{v}_A \Leftarrow \mathbf{v}_B\mathbf{C}$$

*Declaration:* **`void BrMatrix34ApplyV(br_vector3* A, const br_vector3* B, const br_matrix34* C)`**

*Arguments:* **`br_vector3 * A`**

A pointer to the destination vector (must be different from source, and not part of transform), to hold the transformed vector.

**`const br_vector3 * B`**

A pointer to the source vector, holding the vector to be transformed.

**`const br_matrix34 * C`**

A pointer to the transform matrix to be applied.

*Remarks:* The result in `A` is equivalent to the following:

$$\begin{pmatrix} x_B & y_B & z_B & 0 \end{pmatrix} \begin{pmatrix} c_{00} & c_{01} & c_{02} & 0 \\ c_{10} & c_{11} & c_{12} & 0 \\ c_{20} & c_{21} & c_{22} & 0 \\ c_{30} & c_{31} & c_{32} & 1 \end{pmatrix} \equiv$$

$$\begin{pmatrix} x_Bc_{00} + y_Bc_{10} + z_Bc_{20} + c_{30} & x_Bc_{01} + y_Bc_{11} + z_Bc_{21} + c_{31} & x_Bc_{02} + y_Bc_{12} + z_Bc_{22} + c_{32} & 0 \end{pmatrix}$$

## BrMatrix34TApply()

*Description:* Applies a transform to a transposed 3D point which may have non-unity homogenous co-ordinates. Equivalent to the expression:

$$P_A \Leftarrow P_B\mathbf{C}^t$$

*Declaration:* **`void BrMatrix34TApply(br_vector4* A, const br_vector4* B, const br_matrix34* C)`**

*Arguments:* **`br_vector4 * A`**

A pointer to the destination vector (must be different from source), to hold the transformed point.

**`const br_vector4 * B`**

A pointer to the source vector, holding the point to be transformed.

**`const br_matrix34 * C`**

A pointer to the transform matrix to be applied transposed.

*Remarks:* The result in `A` is equivalent to the following:

$$\begin{pmatrix} c_{00} & c_{01} & c_{02} & 0 \\ c_{10} & c_{11} & c_{12} & 0 \\ c_{20} & c_{21} & c_{22} & 0 \\ c_{30} & c_{31} & c_{32} & 1 \end{pmatrix} \begin{pmatrix} x_B \\ y_B \\ z_B \\ w_B \end{pmatrix} \equiv \begin{pmatrix} c_{00}x_B + c_{01}y_B + c_{02}z_B \\ c_{10}x_B + c_{11}y_B + c_{12}z_B \\ c_{20}x_B + c_{21}y_B + c_{22}z_B \\ c_{30}x_B + c_{31}y_B + c_{32}z_B + w_B \end{pmatrix}$$

## BrMatrix34TApplyP()

*Description:* Applies a transposed transform to a 3D point. Equivalent to the expression:

$$P_A \Leftarrow P_B \mathbf{C}^t$$

*Declaration:* **`void BrMatrix34TApplyP(br_vector3* A, const br_vector3* B, const br_matrix34* C)`**

*Arguments:* **`br_vector3 * A`**

A pointer to the destination vector (must be different from source, and not part of transform), to hold the transformed point.

**`const br_vector3 * B`**

A pointer to the source vector, holding the point to be transformed.

**`const br_matrix34 * C`**

A pointer to the transform matrix to be applied transposed – the translation elements are presumed zero or irrelevant.

*Remarks:* The result in `A` is equivalent to the following:

$$\begin{pmatrix} c_{00} & c_{01} & c_{02} & 0 \\ c_{10} & c_{11} & c_{12} & 0 \\ c_{20} & c_{21} & c_{22} & 0 \\ - & - & - & 1 \end{pmatrix} \begin{pmatrix} x_B \\ y_B \\ z_B \\ 1 \end{pmatrix} \equiv \begin{pmatrix} c_{00}x_B + c_{01}y_B + c_{02}z_B \\ c_{10}x_B + c_{11}y_B + c_{12}z_B \\ c_{20}x_B + c_{21}y_B + c_{22}z_B \\ 1 \end{pmatrix}$$

## BrMatrix34TApplyV()

*Description:* Applies a transposed transform to a 3D vector, i.e. as for a point but without translation components (a vector has no location). Equivalent to the expression:

$$\mathbf{v}_A \Leftarrow \mathbf{v}_B \mathbf{C}^t$$

*Declaration:* **`void BrMatrix34TApplyV(br_vector3* A, const br_vector3* B, const br_matrix34* C)`**

*Arguments:* **br_vector3 * A**

A pointer to the destination vector (must be different from source, and not part of transform), to hold the transformed vector.

**const br_vector3 * B**

A pointer to the source vector, holding the vector to be transformed.

**const br_matrix34 * C**

A pointer to the transform matrix to be applied transposed – the translation elements are presumed zero or irrelevant.

*Remarks:* The result in `A` is equivalent to the following:

$$\begin{pmatrix} c_{00} & c_{01} & c_{02} & 0 \\ c_{10} & c_{11} & c_{12} & 0 \\ c_{20} & c_{21} & c_{22} & 0 \\ - & - & - & 1 \end{pmatrix} \begin{pmatrix} x_B \\ y_B \\ z_B \\ 0 \end{pmatrix} \equiv \begin{pmatrix} c_{00}x_B + c_{01}y_B + c_{02}z_B \\ c_{10}x_B + c_{11}y_B + c_{12}z_B \\ c_{20}x_B + c_{21}y_B + c_{22}z_B \\ 0 \end{pmatrix}$$

## BrMatrix34Pre()

*Description:* Pre-multiply one matrix by another. Equivalent to the expression:

$$\mathbf{A} \Leftarrow \mathbf{BA}$$

*Declaration:* **void BrMatrix34Pre(br_matrix34* A, const br_matrix34* B)**

*Arguments:* **br_matrix34 * A**

A pointer to the subject matrix (may be same as `B`).

**const br_matrix34 * B**

A pointer to the pre-multiplying matrix.

*See Also:* **BrMatrix34Post()** 202, **BrMatrix34Mul()** 191

## BrMatrix34PreTranslate()

*Description:* Pre-multiply a matrix by a translation transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{T}_{xyz}\mathbf{M}$$

*Declaration:* **void BrMatrix34PreTranslate(br_matrix34* mat, br_scalar dx, br_scalar dy, br_scalar dz)**

*Arguments:* **br_matrix34 * mat**

A pointer to the subject matrix.

**br_scalar dx**

The x axis component used to form the translation matrix.

**br_scalar dy**

The y axis component used to form the translation matrix.

**br_scalar dz**

The z axis component used to form the translation matrix.

*See Also:* **BrMatrix34PostTranslate()** 202, **BrMatrix34Translate()** 211

---

# BrMatrix34PreScale()

*Description:* Pre-multiply a matrix by a scaling transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{S}_{xyz}\mathbf{M}$$

*Declaration:* **void BrMatrix34PreScale(br_matrix34* mat, br_scalar sx, br_scalar sy, br_scalar sz)**

*Arguments:* **br_matrix34 * mat**

A pointer to the subject matrix.

**br_scalar sx**

Scaling component along the x axis.

**br_scalar sy**

Scaling component along the y axis.

**br_scalar sz**

Scaling component along the z axis.

*See Also:* **BrMatrix34PostScale()** 203, **BrMatrix34Scale()** 212

---

# BrMatrix34PreShearX()

*Description:* Pre-multiply a matrix by an x invariant shearing transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{Z}_{X}\mathbf{M}$$

*Declaration:* **void BrMatrix34PreShearX(br_matrix34* mat, br_scalar sy, br_scalar sz)**

*Arguments:* **br_matrix34 * mat**

A pointer to the subject matrix.

**br_scalar sy**

Shear factor by which the x co-ordinate is included in the transformed y co-ordinate.

**br_scalar sz**

Shear factor by which the x co-ordinate is included in the transformed z co-ordinate.

*See Also:* **BrMatrix34PostShearX()** 203, **BrMatrix34ShearX()** 212

---

# BrMatrix34PreShearY()

*Description:* Pre-multiply a matrix by a y invariant shearing transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{Z}_Y\mathbf{M}$$

*Declaration:* **void BrMatrix34PreShearY(br_matrix34* mat, br_scalar sx, br_scalar sz)**

*Arguments:* **br_matrix34 * mat**

A pointer to the subject matrix.

**br_scalar sx**

Shear factor by which the y co-ordinate is included in the transformed x co-ordinate.

**br_scalar sz**

Shear factor by which the y co-ordinate is included in the transformed z co-ordinate.

*See Also:* **BrMatrix34PostShearY()** 204, **BrMatrix34ShearY()** 213

---

# BrMatrix34PreShearZ()

*Description:* Pre-multiply a matrix by a z invariant shearing transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{Z}_Z\mathbf{M}$$

*Declaration:* **void BrMatrix34PreShearY(br_matrix34* mat, br_scalar sx, br_scalar sy)**

*Arguments:* **br_matrix34 * mat**

A pointer to the subject matrix.

**br_scalar sx**

Shear factor by which the z co-ordinate is included in the transformed x co-ordinate.

**br_scalar sy**

Shear factor by which the z co-ordinate is included in the transformed y co-ordinate.

*See Also:* **BrMatrix34PostShearZ()** 204, **BrMatrix34ShearZ()** 214

---

# BrMatrix34PreRotate()

*Description:* Pre-multiply a matrix by a vector specified axis, rotational transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{R}_{\theta_a}\mathbf{M}$$

*Declaration:* **void BrMatrix34PreRotate(br_matrix34* mat, br_angle r, const br_vector3* axis)**

*Arguments:* **br_matrix34 * mat**

A pointer to the subject matrix.

**br_angle r**

The angle about the specified axis used to form the rotation matrix. A positive angle represents a clockwise rotation (with the vector pointing at you).

**const br_vector3***

The arbitrary (normalised) axis vector about which the rotation occurs.

*See Also:* **BrMatrix34PostRotate()** 204, **BrMatrix34Rotate()** 216

---

# BrMatrix34PreRotateX()

*Description:* Pre-multiply a matrix by an x axis rotational transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{R}_{\theta_X}\mathbf{M}$$

*Declaration:* **void BrMatrix34PreRotateX(br_matrix34* mat, br_angle rx)**

*Arguments:* **br_matrix34 * mat**

A pointer to the subject matrix.

**br_angle rx**

The angle about the x axis used to form the rotation matrix. A positive angle represents a clockwise rotation (looking toward the origin).

*See Also:* **BrMatrix34PostRotateX()** 205, **BrMatrix34RotateX()** 214

## BrMatrix34PreRotateY()

*Description:* Pre-multiply a matrix by a y axis, rotational transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{R}_{\theta_Y}\mathbf{M}$$

*Declaration:* **void BrMatrix34PreRotateY(br_matrix34* mat, br_angle ry)**

*Arguments:* **br_matrix34 * mat**

A pointer to the subject matrix.

**br_angle ry**

The angle about the y axis used to form the rotation matrix. A positive angle represents a clockwise rotation (looking towards the origin).

*See Also:* **BrMatrix34PostRotateY()** 205, **BrMatrix34RotateY()** 215

## BrMatrix34PreRotateZ()

*Description:* Pre-multiply a matrix by a z axis, rotational transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{R}_{\theta_Z}\mathbf{M}$$

*Declaration:* **void BrMatrix34PreRotateZ(br_matrix34* mat, br_angle rz)**

*Arguments:* **br_matrix34 * mat**

A pointer to the subject matrix.

**br_angle rz**

The angle about the z axis used to form the rotation matrix. A positive angle represents a clockwise rotation (looking towards the origin).

*See Also:* **BrMatrix34PostRotateZ()** 206, **BrMatrix34RotateZ()** 215

## BrMatrix34PreTransform()

*Description:* Pre-multiply a matrix by a generic transform. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{M}_{\mathbf{T}}\mathbf{M}$$

*Declaration:* **void BrMatrix34PreTransform(br_matrix34* mat,**
**const br_transform* xform)**

*Arguments:* **br_matrix34 * mat**

A pointer to the subject matrix.

**const br_transform * xform**

The pre-multiplying generic transform.

*Effects:* The transform is first converted to a general 3x4 transform matrix using **BrTransformToMatrix34()** 353 and then applied as a pre-multiplying matrix using **BrMatrix34Pre()** 197.

*See Also:* **BrMatrix34PostTransform()** 206.

---

# BrMatrix34Post()

*Description:* Post-multiply one matrix by another. Equivalent to the expression:

$$\mathbf{A} \Leftarrow \mathbf{AB}$$

*Declaration:* **void BrMatrix34Post(br_matrix34* A, const br_matrix34* B)**

*Arguments:* **br_matrix34 * A**

A pointer to the subject matrix (may be same as B).

**const br_matrix34 * B**

A pointer to the post-multiplying matrix.

*See Also:* **BrMatrix34Pre()** 197, **BrMatrix34Mul()** 191

---

# BrMatrix34PostTranslate()

*Description:* Post-multiply a matrix by a translation transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{MT}_{xyz}$$

*Declaration:* **void BrMatrix34PostTranslate(br_matrix34* mat, br_scalar dx, br_scalar dy, br_scalar dz)**

*Arguments:* **br_matrix34 * mat**

A pointer to the subject matrix.

**br_scalar dx**

The x axis component used to form the translation matrix.

**br_scalar dy**

The y axis component used to form the translation matrix.

**br_scalar dz**

The z axis component used to form the translation matrix.

*See Also:* **BrMatrix34PreTranslate()** 197, **BrMatrix34Translate()** 211

## BrMatrix34PostScale()

*Description:* Post-multiply a matrix by a scaling transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{MS}_{xyz}$$

*Declaration:* **void BrMatrix34PostScale(br_matrix34* mat, br_scalar sx, br_scalar sy, br_scalar sz)**

*Arguments:* **br_matrix34 * mat**

A pointer to the subject matrix.

**br_scalar sx**

Scaling component along the x axis.

**br_scalar sy**

Scaling component along the y axis.

**br_scalar sz**

Scaling component along the z axis.

*See Also:* **BrMatrix34PreScale()** 198, **BrMatrix34Scale()** 212

## BrMatrix34PostShearX()

*Description:* Post-multiply a matrix by an x invariant shearing transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{MZ}_{X}$$

*Declaration:* **void BrMatrix34PostShearX(br_matrix34* mat, br_scalar sy, br_scalar sz)**

*Arguments:* **br_matrix34 * mat**

A pointer to the subject matrix.

**br_scalar sy**

Shear factor by which the x co-ordinate is included in the transformed y co-ordinate.

**br_scalar sz**

Shear factor by which the x co-ordinate is included in the transformed z co-ordinate.

*See Also:* **BrMatrix34PreShearX()** 198, **BrMatrix34ShearX()** 212

## BrMatrix34PostShearY()

*Description:* Post-multiply a matrix by a y invariant shearing transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{M}\mathbf{Z}_Y$$

*Declaration:* **void BrMatrix34PostShearY(br_matrix34* mat, br_scalar sx, br_scalar sz)**

*Arguments:* **br_matrix34 * mat**

A pointer to the subject matrix.

**br_scalar sx**

Shear factor by which the y co-ordinate is included in the transformed x co-ordinate.

**br_scalar sz**

Shear factor by which the y co-ordinate is included in the transformed z co-ordinate.

*See Also:* **BrMatrix34PreShearY()** 199, **BrMatrix34ShearY()** 213

## BrMatrix34PostShearZ()

*Description:* Post-multiply a matrix by a z invariant shearing transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{M}\mathbf{Z}_Z$$

*Declaration:* **void BrMatrix34PostShearY(br_matrix34* mat, br_scalar sx, br_scalar sy)**

*Arguments:* **br_matrix34 * mat**

A pointer to the subject matrix.

**br_scalar sx**

Shear factor by which the z co-ordinate is included in the transformed x co-ordinate.

**br_scalar sy**

Shear factor by which the z co-ordinate is included in the transformed y co-ordinate.

*See Also:* **BrMatrix34PreShearZ()** 199, **BrMatrix34ShearZ()** 214

## BrMatrix34PostRotate()

*Description:* Post-multiply a matrix by a vector specified axis, rotational transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{MR}_{\theta_a}$$

*Declaration:* **void BrMatrix34PostRotate(br_matrix34* mat, br_angle r, const br_vector3* axis)**

*Arguments:* **br_matrix34 * mat**

A pointer to the subject matrix.

**br_angle r**

The angle about the specified axis used to form the rotation matrix.

**const br_vector3***

The arbitrary (normalised) axis vector about which the rotation occurs.

*See Also:* **BrMatrix34PreRotate()** 200, **BrMatrix34Rotate()** 216

## BrMatrix34PostRotateX()

*Description:* Post-multiply a matrix by an x axis rotational transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{MR}_{\theta_X}$$

*Declaration:* **void BrMatrix34PostRotateX(br_matrix34* mat, br_angle rx)**

*Arguments:* **br_matrix34 * mat**

A pointer to the subject matrix.

**br_angle rx**

The angle about the x axis used to form the rotation matrix. A positive angle represents a clockwise rotation (looking towards the origin).

*See Also:* **BrMatrix34PreRotateX()** 200, **BrMatrix34RotateX()** 214

## BrMatrix34PostRotateY()

*Description:* Post-multiply a matrix by a y axis, rotational transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{MR}_{\theta_Y}$$

*Declaration:* **void BrMatrix34PostRotateY(br_matrix34* mat, br_angle ry)**

*Arguments:* **br_matrix34 * mat**

A pointer to the subject matrix.

**br_angle ry**

The angle about the y axis used to form the rotation matrix. A positive angle represents a clockwise rotation (looking towards the origin).

*See Also:* **BrMatrix34PreRotateY()** 201, **BrMatrix34RotateY()** 216

---

# BrMatrix34PostRotateZ()

*Description:* Post-multiply a matrix by a z axis, rotational transform matrix. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{M}\mathbf{R}_{\theta_Z}$$

*Declaration:* **void BrMatrix34PostRotateZ(br_matrix34* mat, br_angle rz)**

*Arguments:* **br_matrix34 * mat**

A pointer to the subject matrix.

**br_angle rz**

The angle about the z axis used to form the rotation matrix. A positive angle represents a clockwise rotation (looking towards the origin).

*See Also:* **BrMatrix34PreRotateZ()** 201, **BrMatrix34RotateZ()** 215

---

# BrMatrix34PostTransform()

*Description:* Post-multiply a matrix by a generic transform. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{M}\mathbf{M}_{\mathbf{T}}$$

*Declaration:* **void BrMatrix34PostTransform(br_matrix34* mat, const br_transform* xform)**

*Arguments:* **br_matrix34 * mat**

A pointer to the subject matrix.

**const br_transform * xform**

The post-multiplying generic transform.

*Effects:* The transform is first converted to a general 3x4 transform matrix using **BrTransformToMatrix34()** 353 and then applied as a post-multiplying matrix using **BrMatrix34Post()** 202.

*See Also:* **BrMatrix34PreTransform()** 201.

---

## Conversion

Note that only matrices can represent the full gamut of translation, shearing, reflection, and scaling effects, some of these effects will be lost (or produce undefined behaviour) when converting into another transformation.

### *From Eulers, Quaternions and Transforms*

See **BrEulerToMatrix34()** 123, **BrQuatToMatrix34()** 323, **BrTransformToMatrix34()** 353.

Also see **BrTransformToTransform()** 353.

### *To Eulers, Quaternions and Transforms*

See **BrMatrix34ToEuler()** 207, **BrMatrix34ToQuat()** 207, **BrMatrix34ToTransform()** 208 as described below.

Also see **BrTransformToTransform()** 353.

---

# BrMatrix34ToEuler()

*Description:* Convert a 3D affine matrix to a Euler angle set, that would have the same rotational effect.

*Declaration:*
```
br_euler* BrMatrix34ToEuler(br_euler* euler,
const br_matrix34* mat)
```

*Arguments:* **br_euler * euler**

A pointer to the destination Euler angle set to receive the conversion. The Euler angle set's Euler order is used to determine each angle.

**const br_matrix34 * mat**

A pointer to the source matrix to convert from.

*Result:* **br_euler ***

Returns `euler` for convenience.

*Remarks:* Translation components of the matrix are lost in conversion.

---

# BrMatrix34ToQuat()

*Description:* Convert a 3D affine matrix to a quaternion, that would have the same rotational effect.

*Declaration:*
```
br_quat* BrMatrix34ToQuat(br_quat* q,
const br_matrix34* mat)
```

*Arguments:* **br_quat * q**

A pointer to the destination quaternion to receive the conversion.

**const br_matrix34 * mat**

A pointer to the source matrix to convert from.

*Result:* **br_quat * q**

Returns `q` for convenience.

*Remarks:* Translation components of the matrix are lost in conversion.

---

# BrMatrix34ToTransform()

*Description:* Convert a 3D affine matrix into a specific transform, that would have a similar transformational effect.

*Declaration:* **void BrMatrix34ToTransform(br_transform* xform, const br_matrix34* mat)**

*Arguments:* **br_transform * xform**

A pointer to the destination transform. The type member of the destination transform is retained and determines the method of conversion.

**const br_matrix34 * mat**

A pointer to the source matrix to be converted.

*Effects:* *When the transform is the identity*

The destination transform is left unchanged – no conversion necessary.

*When the transform is a translation*

The translation component of the matrix (its bottom row) is copied into the translation vector of the transform.

*When the transform is a Euler angle set*

Calls **BrMatrix34ToEuler()** 207 and copies the matrix's translation component (its bottom row) int to the translation vector of the transform.

*When the transform is a Look-Up*

The Up vector is left unchanged and should really be set before rather than after this conversion.

Copies the third row into the Look vector (if zero (0,0,0) then (0,1,0) is used instead).

The matrix's translation component is copied into the translation vector of the transform.

*When the transform is a quaternion*

Calls **BrMatrix34ToQuat()** 207 and copies the matrix's translation component (its bottom row) int to the translation vector of the transform.

*When the transform is a 3x4 matrix*

Directly copies the matrix into the transform.

*When the transform is a 3x4 length preserving matrix*

Directly copies the matrix into the transform and then calls **BrMatrix34LPNormalise()**[210].

---

## Copy/Assign

Although copy by structure assignment currently works, use **BrMatrix34Copy()**[209] or **BrMatrix34Copy4()**[209] to ensure compatibility.

---

# BrMatrix34Copy()

*Description:* Copy a matrix. Equivalent to the expression:

$$\mathbf{A} \Leftarrow \mathbf{B}$$

*Declaration:* **void BrMatrix34Copy(br_matrix34* A, const br_matrix34* B)**

*Arguments:* **br_matrix34 * A**

A pointer to the destination matrix (may be the same as source – though redundant).

**const br_matrix34 * B**

A pointer to the source matrix.

---

# BrMatrix34Copy4()

*Description:* Copy a 4x4 matrix into a 3x4 matrix, discarding right-hand column. Equivalent to the expression:

$$\mathbf{A} \Leftarrow \mathbf{B}$$

*Declaration:* **void BrMatrix34Copy4(br_matrix34* A, const br_matrix4* B)**

*Arguments:* **br_matrix34 * A**

A pointer to the destination matrix.

**const br_matrix4 * B**

A pointer to the source 4x4 matrix.

*See Also:* **BrMatrix4Copy34()**[226].

---

## Access & Maintenance

Members may be freely accessed. Maintenance is only required for length preserving matrices that have been modified.

---

# BrMatrix34LPNormalise()

*Description:* Normalise a length preserving* matrix. Equivalent to the expression:

$$\mathbf{A}_{LP} \Leftarrow \mathrm{Norm}(\mathbf{B}_{\sim LP})$$

*Declaration:*
```
void BrMatrix34LPNormalise(br_matrix34* A,
const br_matrix34* B)
```

*Arguments:* **br_matrix34 * A**

A pointer to the destination matrix, which must not point to the source matrix.

**const br_matrix34 * B**

A pointer to the source matrix.

*Effects:* The destination matrix is the source matrix adjusted so that it represents a length preserving transformation.

*Remarks:* This function is typically applied to a length preserving matrix which has undergone a long sequence of operations, to ensure that the final matrix is still truly length preserving (regardless of rounding errors).

---

## Referencing & Lifetime

This structure may be freely referenced, though take care if there is potential to supply the same matrix as more than one argument to the same function.

## Initialisation

No static initialisers are provided. However, four `BR_VECTOR3()` macros would serve as well.

All other initialisation should use **BrMatrix34Copy()** [209] or any of the following initialisation functions.

---

* Note that length preserving also applies to the sign of lengths, not just their magnitude, i.e. a reflection is not length preserving in this case.

## BrMatrix34Identity()

*Description:* Set the specified matrix to the identity transformation matrix. Equivalent to:

$$\mathbf{M} \Leftarrow \mathbf{I} \equiv \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

*Declaration:* **void BrMatrix34Identity(br_matrix34* mat)**

*Arguments:* **br_matrix34 * mat**

A pointer to the destination matrix.

*Effects:* Stores the identity matrix at the destination.

## BrMatrix34Translate()

*Description:* Set the specified matrix to a matrix representing a specific translation. Equivalent to:

$$\mathbf{M} \Leftarrow \mathbf{T}_{xyz} \equiv \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \delta_x & \delta_y & \delta_z & 1 \end{pmatrix}$$

*Declaration:* **void BrMatrix34Translate(br_matrix34* mat, br_scalar dx, br_scalar dy, br_scalar dz)**

*Arguments:* **br_matrix34 * mat**

A pointer to the destination matrix.

**br_scalar dx**

Translation component along the x axis.

**br_scalar dy**

Translation component along the y axis.

**br_scalar dz**

Translation component along the z axis.

*See Also:* **BrMatrix34PreTranslate()** 197, **BrMatrix34PostTranslate()** 202.

## BrMatrix34Scale()

*Description:* Set the specified matrix to a matrix representing a specific scaling. Equivalent to:

$$\mathbf{M} \Leftarrow \mathbf{S}_{xyz} \equiv \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

*Declaration:* **void BrMatrix34Scale(br_matrix34* mat, br_scalar sx, br_scalar sy, br_scalar sz)**

*Arguments:* **br_matrix34 * mat**

A pointer to the destination matrix.

**br_scalar sx**

Scaling component along the x axis.

**br_scalar sy**

Scaling component along the y axis.

**br_scalar sz**

Scaling component along the z axis.

*See Also:* **BrMatrix34PreScale()** 198, **BrMatrix34PostScale()** 203.

## BrMatrix34ShearX()

*Description:* Set the specified matrix to a matrix representing a shear, invariant along the x axis. Thus values of y and z co-ordinates will be scaled in proportion to the value of the x co-ordinate. Equivalent to:

$$\mathbf{M} \Leftarrow \mathbf{Z}_X \equiv \begin{pmatrix} 1 & \delta_y & \delta_z & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

*Declaration:* **void BrMatrix34ShearX(br_matrix34* mat, br_scalar sy, br_scalar sz)**

*Arguments:* **br_matrix34 * mat**

A pointer to the destination matrix.

**br_scalar sy**

Shear factor by which the x co-ordinate is included in the transformed y co-ordinate.

**br_scalar sz**

Shear factor by which the x co-ordinate is included in the transformed z co-ordinate.

*See Also:* **BrMatrix34PreShearX()** 198, **BrMatrix34PostShearX()** 203.

---

# BrMatrix34ShearY()

*Description:* Set the specified matrix to a matrix representing a shear, invariant along the y axis. Thus values of x and z co-ordinates will be scaled in proportion to the value of the y co-ordinate. Equivalent to:

$$\mathbf{M} \Leftarrow \mathbf{Z}_Y \equiv \begin{pmatrix} 1 & 0 & 0 & 0 \\ \sigma_x & 1 & \sigma_z & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

*Declaration:* **void BrMatrix34ShearX(br_matrix34* mat, br_scalar sx, br_scalar sz)**

*Arguments:* **br_matrix34 * mat**

A pointer to the destination matrix.

**br_scalar sx**

Shear factor by which the y co-ordinate is included in the transformed x co-ordinate.

**br_scalar sz**

Shear factor by which the y co-ordinate is included in the transformed z co-ordinate.

*See Also:* **BrMatrix34PreShearY()** 199, **BrMatrix34PostShearY()** 204.

## BrMatrix34ShearZ()

*Description:* Set the specified matrix to a matrix representing a shear, invariant along the z axis. Thus values of x and y co-ordinates will be scaled in proportion to the value of the z co-ordinate. Equivalent to:

$$\mathbf{M} \Leftarrow \mathbf{Z}_z \equiv \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \sigma_x & \sigma_y & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

*Declaration:* **void BrMatrix34ShearX(br_matrix34* mat, br_scalar sx, br_scalar sy)**

*Arguments:* **br_matrix34 * mat**

A pointer to the destination matrix.

**br_scalar sx**

Shear factor by which the z co-ordinate is included in the transformed x co-ordinate.

**br_scalar sy**

Shear factor by which the z co-ordinate is included in the transformed y co-ordinate.

*See Also:* **BrMatrix34PreShearZ()** 199, **BrMatrix34PostShearZ()** 204.

## BrMatrix34RotateX()

*Description:* Set the specified matrix to a matrix representing a rotation about the x axis though a specified angle. Equivalent to:

$$\mathbf{M} \Leftarrow \mathbf{R}_{\theta_X} \equiv \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_X & \sin\theta_X & 0 \\ 0 & -\sin\theta_X & \cos\theta_X & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

*Declaration:* **void BrMatrix34RotateX(br_matrix34* mat, br_angle rx)**

*Arguments:* **br_matrix34 * mat**

A pointer to the destination matrix.

**br_angle rx**

Rotation about the x axis.

*See Also:* **BrMatrix34PreRotateX()** 200, **BrMatrix34PostRotateX()** 205.

## BrMatrix34RotateY()

*Description:* Set the specified matrix to a matrix representing a rotation about the y axis though a specified angle. Equivalent to:

$$\mathbf{M} \Leftarrow \mathbf{R}_{\theta_Y} \equiv \begin{pmatrix} \cos\theta_Y & 0 & -\sin\theta_Y & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta_Y & 0 & \cos\theta_Y & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

*Declaration:* **void BrMatrix34RotateX(br_matrix34* mat, br_angle ry)**

*Arguments:* **br_matrix34 * mat**

A pointer to the destination matrix.

**br_angle ry**

Rotation about the y axis.

*See Also:* **BrMatrix34PreRotateY()** 201, **BrMatrix34PostRotateY()** 205.

## BrMatrix34RotateZ()

*Description:* Set the specified matrix to a matrix representing a rotation about the z axis though a specified angle. Equivalent to:

$$\mathbf{M} \Leftarrow \mathbf{R}_{\theta_Z} \equiv \begin{pmatrix} \cos\theta_Z & \sin\theta_Z & 0 & 0 \\ -\sin\theta_Z & \cos\theta_Z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

*Declaration:* **void BrMatrix34RotateZ(br_matrix34* mat, br_angle rz)**

*Arguments:* **br_matrix34 * mat**

A pointer to the destination matrix.

**br_angle rz**

Rotation about the z axis.

*See Also:* **BrMatrix34PreRotateZ()** 201, **BrMatrix34PostRotateZ()** 206.

---

# BrMatrix34Rotate()

*Description:* Set the specified matrix to a matrix representing a rotation about a given axis vector though a specified angle. Equivalent to:

$$\mathbf{M} \Leftarrow \mathbf{R}_{\theta_a}$$

*Declaration:* **void BrMatrix34Rotate(br_matrix34* mat, br_angle r, const br_vector3* a)**

*Arguments:* **br_matrix34 * mat**

A pointer to the destination matrix.

**br_angle r**

Rotation about the specified axis vector.

**const br_vector3 * a**

The arbitrary (normalised) axis vector about which the rotation occurs.

*See Also:* **BrMatrix34PreRotate()** 200, **BrMatrix34PostRotate()** 205.

---

# br_matrix4

## The Structure

A four column, four row, scalar array, used as a 3D affine matrix for general purpose 3D transformations (translation, scaling, shearing, rotation). Functions are provided to allow it be used as though it were an integral type. It has the following form:

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{pmatrix}$$

It can be noted that the bottom row has a translational effect. Also note, that the matrix determinant represents the volume change effected.

### *The* **`typedef`**

(See `matrix.h` for precise declaration and ordering)

| | | |
|---|---|---|
| **`br_scalar`** | **`m[4][4]`** | ***Four rows of four columns*** |

### *Related Functions*

#### Image Support

See **`BrActorToScreenMatrix4()`** 85.

#### Maths

See **`BrEulerToMatrix4()`** 124, **`BrQuatToMatrix4()`** 324, **`BrMatrix34Copy4()`** 209.

### *Related Structures*

See **`br_matrix34`** *190*, **`br_renderbounds_cbfn`** *315*.

## Members

### `br_scalar m[4][4]`

Each element of the matrix can be freely and individually accessed.

This matrix can also be thought of as an array of four **`br_vector4`** *376* structures, e.g. `br_vector4 m[4]`. Thus `m[row]` can be cast as `(br_vector4*)`.

# Arithmetic

## BrMatrix4Mul()

*Description:* Multiply two matrices together and place the result in a third matrix. Equivalent to the expression:

$$\mathbf{A} \Leftarrow \mathbf{BC}$$

*Declaration:* **void BrMatrix4Mul(br_matrix4* A, const br_matrix4* B, const br_matrix4* C)**

*Arguments:* **br_matrix4 * A**

A pointer to the destination matrix (must be different from both sources).

**const br_matrix4 * B**

Pointer to the left hand source matrix.

**const br_matrix4 * C**

Pointer to the right hand source matrix.

*Remarks:* The result in `A` is equivalent to the following:

$$\begin{pmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{pmatrix} \begin{pmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{pmatrix} \equiv$$

$$\begin{pmatrix} b_{00}c_{00}+b_{01}c_{10}+b_{02}c_{20}+b_{03}c_{30} & b_{00}c_{01}+b_{01}c_{11}+b_{02}c_{21}+b_{03}c_{31} & b_{00}c_{02}+b_{01}c_{12}+b_{02}c_{22}+b_{03}c_{32} & b_{00}c_{03}+b_{01}c_{13}+b_{02}c_{23}+b_{03}c_{33} \\ b_{10}c_{00}+b_{11}c_{10}+b_{12}c_{20}+b_{13}c_{30} & b_{10}c_{01}+b_{11}c_{11}+b_{12}c_{21}+b_{13}c_{31} & b_{10}c_{02}+b_{11}c_{12}+b_{12}c_{22}+b_{13}c_{32} & b_{10}c_{03}+b_{11}c_{13}+b_{12}c_{23}+b_{13}c_{33} \\ b_{20}c_{00}+b_{21}c_{10}+b_{22}c_{20}+b_{23}c_{30} & b_{20}c_{01}+b_{21}c_{11}+b_{22}c_{21}+b_{23}c_{31} & b_{20}c_{02}+b_{21}c_{12}+b_{22}c_{22}+b_{23}c_{32} & b_{20}c_{03}+b_{21}c_{13}+b_{22}c_{23}+b_{23}c_{33} \\ b_{30}c_{00}+b_{31}c_{10}+b_{32}c_{20}+b_{33}c_{30} & b_{30}c_{01}+b_{31}c_{11}+b_{32}c_{21}+b_{33}c_{31} & b_{30}c_{02}+b_{31}c_{12}+b_{32}c_{22}+b_{33}c_{32} & b_{30}c_{03}+b_{31}c_{13}+b_{32}c_{23}+b_{33}c_{33} \end{pmatrix}$$

*See Also:* **BrMatrix4Pre34()** 222, **BrMatrix4PreTransform()** 223

## BrMatrix4Inverse()

*Description:* Compute the inverse of the supplied 3D affine matrix. Equivalent to the expression:

$$\mathbf{A} \Leftarrow \mathbf{B}^{-1}$$

*Declaration:* **br_scalar BrMatrix4Inverse(br_matrix4* A, const br_matrix4* B)**

*Arguments:* **br_matrix4 * A**

A pointer to the destination matrix (must be different from source).

**const br_matrix4 * B**

A pointer to the source matrix.

*Result:* **br_scalar**

If the inverse exists, the determinant of the source matrix is returned. If there is no inverse, scalar zero is returned.

*Remarks:* Remember that while an inverse may be obtained using `double` precision arithmetic, this does not necessarily mean that it can using the **br_scalar**[342] type. This difference is most marked between fixed and floating point BRender libraries.

*See Also:* **BrMatrix4Adjoint()**[223], **BrMatrix4Determinant()**[224].

---

# BrMatrix4Apply()

*Description:* Applies a transform to a 3D point which may have non-unity homogenous co-ordinates. Equivalent to the expression:

$$P_A \Leftarrow P_B \mathbf{C}$$

*Declaration:* **void BrMatrix4Apply(br_vector4* A, const br_vector4* B, const br_matrix4* C)**

*Arguments:* **br_vector4 * A**

A pointer to the destination vector (must be different from source, and not part of transform), to hold the transformed point.

**const br_vector4 * B**

A pointer to the source vector, holding the point to be transformed.

**const br_matrix4 * C**

A pointer to the transform matrix to be applied.

*Remarks:* The result in `A` is equivalent to the following:

$$\begin{pmatrix} x_B & y_B & z_B & w_B \end{pmatrix} \begin{pmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{pmatrix} \equiv$$

$$\begin{pmatrix} x_B c_{00} + y_B c_{10} + z_B c_{20} + w_B c_{30} & x_B c_{01} + y_B c_{11} + z_B c_{21} + w_B c_{31} & x_B c_{02} + y_B c_{12} + z_B c_{22} + w_B c_{32} & x_B c_{03} + y_B c_{13} + z_B c_{23} + w_B c_{33} \end{pmatrix}$$

---

# BrMatrix4ApplyP()

*Description:* Applies a transform to a 3D point. Equivalent to the expression:

$$P_A \Leftarrow P_B \mathbf{C}$$

*Declaration:* **void BrMatrix4ApplyP(br_vector4* A, const br_vector3* B, const br_matrix4* C)**

*Arguments:* **br_vector4 * A**

A pointer to the destination vector (must be different from source, and not part of transform), to hold the transformed point.

**const br_vector3 * B**

A pointer to the source vector, holding the point to be transformed.

**const br_matrix4 * C**

A pointer to the transform matrix to be applied.

*Remarks:* The result in `A` is equivalent to the following:

$$\begin{pmatrix} 1 \end{pmatrix} \begin{pmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{pmatrix} \equiv$$

$$\begin{pmatrix} + z_B c_{20} + c_{30} & x_B c_{01} + y_B c_{11} + z_B c_{21} + c_{31} & x_B c_{02} + y_B c_{12} + z_B c_{22} + c_{32} & x_B c_{03} + y_B c_{13} + z_B c_{23} + c_{33} \end{pmatrix}$$

---

# BrMatrix4ApplyV()

*Description:* Applies a transform to a 3D vector, i.e. as for a point but without translation components (a vector has no location). Equivalent to the expression:

$$\mathbf{v}_A \Leftarrow \mathbf{v}_B \mathbf{C}$$

*Declaration:* **void BrMatrix4ApplyV(br_vector4* A, const br_vector3* B, const br_matrix4* C)**

*Arguments:* **br_vector4 * A**

A pointer to the destination vector (must be different from source, and not part of transform), to hold the transformed vector.

**const br_vector3 * B**

A pointer to the source vector, holding the vector to be transformed.

**const br_matrix4 * C**

A pointer to the transform matrix to be applied.

*Remarks:* The result in `A` is equivalent to the following:

$$\begin{pmatrix} x_B & y_B & z_B & 0 \end{pmatrix} \begin{pmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{pmatrix} \equiv$$

$$\begin{pmatrix} x_B c_{00} + y_B c_{10} + z_B c_{20} & x_B c_{01} + y_B c_{11} + z_B c_{21} & x_B c_{02} + y_B c_{12} + z_B c_{22} & x_B c_{03} + y_B c_{13} + z_B c_{23} \end{pmatrix}$$

## BrMatrix4TApply()

*Description:* Applies a transform to a transposed 3D point which may have non-unity homogenous co-ordinates. Equivalent to the expression:

$$\mathbf{P}_A \Leftarrow \mathbf{P}_B\mathbf{C}^t$$

*Declaration:* **`void BrMatrix4TApply(br_vector4* A, const br_vector4* B, const br_matrix4* C)`**

*Arguments:* **`br_vector4 * A`**

A pointer to the destination vector (must be different from source), to hold the transformed point.

**`const br_vector4 * B`**

A pointer to the source vector, holding the point to be transformed.

**`const br_matrix4 * C`**

A pointer to the transform matrix to be applied transposed.

*Remarks:* The result in `A` is equivalent to the following:

$$\begin{pmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{pmatrix} \begin{pmatrix} x_B \\ y_B \\ z_B \\ w_B \end{pmatrix} \equiv \begin{pmatrix} c_{00}x_B + c_{01}y_B + c_{02}z_B + c_{03}w_B \\ c_{10}x_B + c_{11}y_B + c_{12}z_B + c_{13}w_B \\ c_{20}x_B + c_{21}y_B + c_{22}z_B + c_{23}w_B \\ c_{30}x_B + c_{31}y_B + c_{32}z_B + c_{33}w_B \end{pmatrix}$$

## BrMatrix4TApplyP()

*Description:* Applies a transposed transform to a 3D point. Equivalent to the expression:

$$\mathbf{P}_A \Leftarrow \mathbf{P}_B\mathbf{C}^t$$

*Declaration:* **`void BrMatrix4TApplyP(br_vector4* A, const br_vector3* B, const br_matrix4* C)`**

*Arguments:* **`br_vector4 * A`**

A pointer to the destination vector (must be different from source, and not part of transform), to hold the transformed point.

**`const br_vector3 * B`**

A pointer to the source vector, holding the point to be transformed.

**`const br_matrix4 * C`**

A pointer to the transform matrix to be applied transposed.

*Remarks:* The result in `A` is equivalent to the following:

$$\begin{pmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{pmatrix} \begin{pmatrix} x_B \\ y_B \\ z_B \\ 1 \end{pmatrix} \equiv \begin{pmatrix} c_{00}x_B + c_{01}y_B + c_{02}z_B + c_{03} \\ c_{10}x_B + c_{11}y_B + c_{12}z_B + c_{13} \\ c_{20}x_B + c_{21}y_B + c_{22}z_B + c_{23} \\ c_{30}x_B + c_{31}y_B + c_{32}z_B + c_{33} \end{pmatrix}$$

---

# BrMatrix4TApplyV()

*Description:* Applies a transposed transform to a 3D vector, i.e. as for a point but without translation components (a vector has no location). Equivalent to the expression:

$$\mathbf{v}_A \Leftarrow \mathbf{v}_B\mathbf{C}^t$$

*Declaration:* **`void BrMatrix4TApplyV(br_vector4* A, const br_vector3* B, const br_matrix4* C)`**

*Arguments:* **`br_vector4 * A`**

A pointer to the destination vector (must be different from source, and not part of transform), to hold the transformed vector.

**`const br_vector3 * B`**

A pointer to the source vector, holding the vector to be transformed.

**`const br_matrix4 * C`**

A pointer to the transform matrix to be applied transposed.

*Remarks:* The result in `A` is equivalent to the following:

$$\begin{pmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{pmatrix} \begin{pmatrix} x_B \\ y_B \\ z_B \\ 0 \end{pmatrix} \equiv \begin{pmatrix} c_{00}x_B + c_{01}y_B + c_{02}z_B \\ c_{10}x_B + c_{11}y_B + c_{12}z_B \\ c_{20}x_B + c_{21}y_B + c_{22}z_B \\ c_{30}x_B + c_{31}y_B + c_{32}z_B \end{pmatrix}$$

---

# BrMatrix4Pre34()

*Description:* Pre-multiply one matrix by another. Equivalent to the expression:

$$\mathbf{A} \Leftarrow \mathbf{B}\mathbf{A}$$

*Declaration:* **`void BrMatrix4Pre34(br_matrix4* A, const br_matrix34* B)`**

*Arguments:* **br_matrix4 * A**

A pointer to the subject matrix (may be same as `B`).

**const br_matrix34 * B**

A pointer to the pre-multiplying matrix.

*Remarks:* The result in `A` is equivalent to the following:

$$\begin{pmatrix} b_{02} & 0 \\ b_{12} & 0 \\ b_{22} & 0 \\ b_{32} & 0 \end{pmatrix} \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \equiv$$

$$\begin{pmatrix} {}_1a_{10} + b_{02}a_{20} & b_{00}a_{01} + b_{01}a_{11} + b_{02}a_{21} & b_{00}a_{02} + b_{01}a_{12} + b_{02}a_{22} & b_{00}a_{03} + b_{01}a_{13} + b_{02}a_{23} \\ {}_1a_{10} + b_{12}a_{20} & b_{10}a_{01} + b_{11}a_{11} + b_{12}a_{21} & b_{10}a_{02} + b_{11}a_{12} + b_{12}a_{22} & b_{10}a_{03} + b_{11}a_{13} + b_{12}a_{23} \\ {}_1a_{10} + b_{22}a_{20} & b_{20}a_{01} + b_{21}a_{11} + b_{22}a_{21} & b_{20}a_{02} + b_{21}a_{12} + b_{22}a_{22} & b_{20}a_{03} + b_{21}a_{13} + b_{22}a_{23} \\ {}_1a_{10} + b_{32}a_{20} + a_{30} & b_{30}a_{01} + b_{31}a_{11} + b_{32}a_{21} + a_{31} & b_{30}a_{02} + b_{31}a_{12} + b_{32}a_{22} + a_{32} & b_{30}a_{03} + b_{31}a_{13} + b_{32}a_{23} + a_{33} \end{pmatrix}$$

*See Also:* **BrMatrix4Mul()** 218.

---

# BrMatrix4PreTransform()

*Description:* Pre-multiply a matrix by a generic transform. Equivalent to the expression:

$$\mathbf{M} \Leftarrow \mathbf{M}_T\mathbf{M}$$

*Declaration:* **void BrMatrix4PreTransform(br_matrix4* mat, const br_transform* xform)**

*Arguments:* **br_matrix4 * mat**

A pointer to the subject matrix.

**const br_transform * xform**

The pre-multiplying generic transform.

*Effects:* The transform is first converted to a general 3x4 transform matrix using **BrTransformToMatrix34()** 353 and then applied as a pre-multiplying matrix using **BrMatrix4Pre34()** 222.

---

# BrMatrix4Adjoint()

*Description:* Find the adjoint of a matrix – the transposed matrix of co-factors. Equivalent to the expression:

$$\mathbf{A} \Leftarrow \text{Adjoint}(\mathbf{B})$$

*Declaration:* **void BrMatrix4Adjoint(br_matrix4* A, const br_matrix4* B)**

*Arguments:* **br_matrix4 * A**

A pointer to the destination matrix (may be same as source).

**const br_matrix4 * B**

A pointer to the source matrix.

*See Also:* **BrMatrix4Inverse()** 218.

---

## BrMatrix4Determinant()

*Description:* Calculate the determinant of a matrix. Equivalent to the expression:

$$|\mathbf{M}|$$

*Declaration:* **br_scalar BrMatrix4Determinant(const br_matrix4* mat)**

*Arguments:* **const br_matrix4 * mat**

A pointer to the source matrix.

*Result:* **br_scalar**

The determinant of the source matrix.

*See Also:* **BrMatrix4Inverse()** 218.

---

# Conversion

### *From Eulers and Quaternions*

See **BrEulerToMatrix4()** 124, **BrQuatToMatrix4()** 324.

### *To Eulers and Quaternions*

See **BrMatrix4ToEuler()** 224, **BrMatrix4ToQuat()** 225 as described below.

---

## BrMatrix4ToEuler()

*Description:* Convert a 3D affine matrix to a Euler angle set, that would have the same rotational effect.

*Declaration:* **br_euler* BrMatrix4ToEuler(br_euler* euler,**
**const br_matrix4* mat)**

*Arguments:* **br_euler * euler**

A pointer to the destination Euler angle set to receive the conversion. The Euler angle set's Euler order is used to determine each angle.

**const br_matrix4 * mat**

A pointer to the source matrix to convert from.

*Result:* **br_euler ***

Returns `euler` for convenience.

*Remarks:* Translation and projective (fourth column) components of the matrix are lost in conversion.

---

## BrMatrix4ToQuat()

*Description:* Convert a 3D affine matrix to a quaternion, that would have the same rotational effect.

*Declaration:* **br_quat* BrMatrix4ToQuat(br_quat* q, const br_matrix4* mat)**

*Arguments:* **br_quat * q**

A pointer to the destination quaternion to receive the conversion.

**const br_matrix4 * mat**

A pointer to the source matrix to convert from.

*Result:* **br_quat * q**

Returns `q` for convenience.

*Remarks:* Translation and projective (fourth column) components of the matrix are lost in conversion.

---

# Copy/Assign

Although copy by structure assignment currently works, use **BrMatrix4Copy()** [225] to ensure compatibility.

---

## BrMatrix4Copy()

*Description:* Copy a matrix. Equivalent to the expression:

$$\mathbf{A} \Leftarrow \mathbf{B}$$

*Declaration:* **void BrMatrix4Copy(br_matrix4* A, const br_matrix4* B)**

*Arguments:* **br_matrix4 * A**

A pointer to the destination matrix (may be the same as source – though redundant).

**const br_matrix4 * B**

A pointer to the source matrix.

---

## BrMatrix4Copy34()

*Description:* Copy a 3x4 matrix into a 4x4 matrix. Equivalent to the expression:

$$\mathbf{A} \Leftarrow \mathbf{B}$$

*Declaration:* **void BrMatrix4Copy34(br_matrix4* A, const br_matrix34* B)**

*Arguments:* **br_matrix4 * A**

A pointer to the destination matrix.

**const br_matrix34 * B**

A pointer to the source 3x4 matrix.

*Effects:* The source is copied into the destination, and the fourth column of the destination is set to the implicit (0,0,0,1) column vector.

*See Also:* **BrMatrix34Copy4()** 209.

---

# Referencing & Lifetime

This structure may be freely referenced, though take care if there is potential to supply the same matrix as more than one argument to the same function.

# Initialisation

No static initialisers are provided. However, four `BR_VECTOR4()` macros would serve as well.

All other initialisation should use **BrMatrix4Copy()** 225 or any of the following initialisation functions.

## BrMatrix4Identity()

*Description:* Set the specified matrix to the identity transformation matrix. Equivalent to:

$$\mathbf{M} \Leftarrow \mathbf{I} \equiv \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

*Declaration:* **`void BrMatrix4Identity(br_matrix4* mat)`**

*Arguments:* **`br_matrix4 * mat`**

A pointer to the destination matrix.

*Effects:* Stores the identity matrix at the destination.

## BrMatrix4Scale()

*Description:* Set the specified matrix to a matrix representing a specific scaling. Equivalent to:

$$\mathbf{M} \Leftarrow \mathbf{S}_{xyz} \equiv \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

*Declaration:* **`void BrMatrix4Scale(br_matrix4* mat, br_scalar sx, br_scalar sy, br_scalar sz)`**

*Arguments:* **`br_matrix4 * mat`**

A pointer to the destination matrix.

**`br_scalar sx`**

Scaling component along the x axis.

**`br_scalar sy`**

Scaling component along the y axis.

**`br_scalar sz`**

Scaling component along the z axis.

# BrMatrix4Perspective()

*Description:* Generate a perspective transformation matrix, that can be used to convert from a camera actor's co-ordinate space into the homogenous screen space (assuming a centred projection). This maps the viewing volume into the rendering volume, a cuboid delimited by the homogenous screen co-ordinates (left, bottom, near) (-1,-1,+1) to (+1,+1,-1). The matrix created is equivalent to the following (α=field of view).

$$\mathbf{M} \Leftarrow \mathbf{Perspective} \equiv \begin{pmatrix} \frac{\cot\frac{1}{2}\alpha}{aspect} & 0 & 0 & 0 \\ 0 & \cot\frac{1}{2}\alpha & 0 & 0 \\ 0 & 0 & \frac{z_{yon}+z_{hither}}{z_{yon}-z_{hither}} & -1 \\ 0 & 0 & \frac{-2z_{yon}z_{hither}}{z_{yon}-z_{hither}} & 0 \end{pmatrix}$$

*Declaration:* **void BrMatrix4Perspective(br_matrix4* mat, br_angle field_of_view, br_scalar aspect, br_scalar hither, br_scalar yon)**

*Arguments:* **br_matrix4 * mat**

A pointer to the destination matrix to receive the perspective transform.

**br_angle field_of_view**

Field of view, i.e. the angle subtended at the camera between the top and bottom of the view volume.

**br_scalar aspect**

Scaling factor for width of viewing volume, i.e. ±1 in y view ordinates is mapped to the height of the output image, and ±`aspect` in x view ordinates is mapped to the width of the output image.

**br_scalar hither**

z ordinate of front of view volume. The value should be less than zero.

**br_scalar yon**

z ordinate of back of view volume. The value should be less than `hither`.

*Effects:* Calculates the camera to screen transformation matrix and stores it at `mat`.

*Remarks:* Note that `hither` and `yon` are of opposite sign to the comparable values of `hither_z` and `yon_z` that would be specified in **br_camera**[110]. This is because they are ordinates as opposed to distances.

Note that the front clip plane is at a homogenous screen z ordinate of -1.

*See Also:* **br_camera**[110], **BrActorToScreenMatrix4()**[85].

br_matrix4

# br_mode_test_cbfn

## The Call-Back Function

This type defines a mode test function, as required by **BrFileOpenRead()**[60].

### *The* ***typedef***

(See *brfile.h* for a precise declaration)

```
int                     br_mode_test_cbfn(const br_uint_8*, br_size_t)
```
*Test file's mode*

## Specification

---

## CBFnModeTest()

*Description:* An application defined call-back function determining the type of a file from a given number of characters (specified with the call-back function) from the start of the file.

*Declaration:*
```
int BR_CALLBACK CBFnModeTest(const br_uint_8* magics,
br_size_t n_magics)
```

*Arguments:*

**const br_uint_8 * magics**

Pointer to n_magics characters.

**br_size_t n_magics**

Number of characters pointed to.

*Effects:* Interpret the characters and determine the mode of the file (or file type).

*Result:* **int**

If the file is binary return BR_FS_MODE_BINARY, if text return BR_FS_MODE_TEXT, otherwise return BR_FS_MODE_UNKNOWN.

---

# br_model

## The Structure

BRender's model data structure, describing a mesh of triangles.

### *The* **typedef**

(See *model.h* for precise declaration and ordering)

#### Behaviour

| | | |
|---|---|---|
| **br_uint_16** | **flags** | ***Model flags*** |
| **br_model_custom_cbfn *** | **custom** | ***A custom model call-back function*** |

#### Geometry

| | | |
|---|---|---|
| **br_vertex *** | **vertices** | ***A pointer to an array of vertices*** |
| **br_uint_16** | **nvertices** | ***Number of vertices in the model*** |
| **br_face *** | **faces** | ***A pointer to an array of faces*** |
| **br_uint_16** | **nfaces** | ***Number of faces in the model*** |
| **br_vector3** | **pivot** | ***Offset of model's pivot point*** |

#### Computed Geometry

| | | |
|---|---|---|
| **br_scalar** | **radius** | ***The bounding radius of the model*** |
| **br_bounds** | **bounds** | ***The axis-aligned bounding box of the model*** |

#### Supplementary

| | | |
|---|---|---|
| **char *** | **identifier** | ***Model name*** |
| **void *** | **user** | ***An optional user-supplied pointer*** |

### *Related Functions*

#### Scene Rendering

See **Br[Zb|Zs]ModelRender()** 253|254.

### *Related Structures*

#### Scene Modelling

See **br_actor**$_{76}$, **br_material**$_{154}$, **br_face**$_{125}$, **br_vertex**$_{378}$.

#### Scene Rendering

See **br_renderbounds_cbfn**$_{315}$, **br_model_custom_cbfn**$_{251}$.

# Members

## *Behaviour*

### br_uint_16 flags

This member determines how the model's geometry is computed. Various flags can be combined using the 'Or' operation. They're described in the following table.

| Flag Symbol | Behaviour |
|---|---|
| BR_MODF_KEEP_ORIGINAL | Retain original vertices and faces during model update – otherwise these are freed and replaced by an optimised and equivalent set (very likely reordered) |
| BR_MODF_GENERATE_TAGS | Improve update speed at the expense of face and vertex tag tables. Only use in conjunction with BR_MODF_KEEP_ORIGINAL |
| BR_MODF_QUICK_UPDATE | Improve update speed at the expense of having models that may take longer to render |
| BR_MODF_DONT_WELD | Don't eliminate redundant vertices (having identical co-ordinates) |
| BR_MODF_CUSTOM | Invoke a custom call-back for this model |

### br_model_custom_cbfn * custom

If the BR_MODF_CUSTOM flag is specified, instead of being rendered, the function pointed to by custom is called. This may of course then call **BrZbModelRender()** 253, say. See **br_model_custom_cbfn** *251*.

## *Geometry*

### br_vertex * vertices

A list of vertex structures describing the model's geometry (also containing texture co-ordinates and pre-lighting). The vertices can be allocated at the same time as the model, otherwise vertices should point to a list with a sufficient lifetime (and BR_MODF_KEEP_ORIGINAL must be set).

### br_uint_16 nvertices

Number of vertices supplied in the list of vertices.

### br_face * faces

A list of face structures describing the model's surface in terms of its vertices (also containing smoothing information, edge flags, and materials). The faces can be allocated at the same time as the model, otherwise faces should point to a list with a sufficient lifetime (and BR_MODF_KEEP_ORIGINAL must be set).

### br_uint_16 nfaces

Number of faces supplied in the list of faces.

### br_vector3 pivot

Offset from model geometry origin to model origin. Effectively an offset which is subtracted from each model vertex. Alternatively, it may be thought of as a vector in the model's co-ordinate space defining the point at which the model attaches to its parent (assuming an identity transform).

This member is provided to facilitate centring geometry (thus not needing to modify vertex data), and thus enables such things as tighter bounding radii. It is not really intended to supplement the model actor transform, i.e. as another way of translating models.

*Computed Geometry*

### br_scalar radius

The maximum vertex length, thus the radius defining the smallest origin centred sphere enclosing the model. This is computed upon **BrModelAdd()** 240 and when the BR_MODU_RADIUS flag is specified to **BrModelUpdate()** 241.

### br_bounds bounds

The minimum and maximum x,y and z ordinates of the vertices, thus the minimal, axis aligned (orthogonal faced) cuboid enclosing the model. This is computed upon **BrModelAdd()** 240 and when the BR_MODU_BOUNDING_BOX flag is specified to **BrModelUpdate()** 241.

*Supplementary*

### char * identifier

Pointer to unique, zero terminated, character string (or NULL if not required). Can be used as a handle to retrieve a pointer to the model. Not intended for intensive use. Typically used to collect pointers to models loaded using **BrModelLoad()** 247 and added to the registry using **BrModelAdd()** 240. Also ideal for diagnostic purposes.

A non-unique string can be supplied, but which of a set of models having the same string will be matched by search functions (See **BrModelFind()** 244), is undefined. Also in consideration of searching, it is not recommended that non-alphabetic characters are used, especially Slash ('/'), Asterisk ('*'), and Query ('?'), which are used for pattern matching.

This member can be modified by the programmer at any time.

If identifier is set by **BrModelLoad()** 247 or **BrModelLoadMany()** 248 it will have been constructed using **BrResStrDup()** 49.

`void * user`

A member whose usage is entirely application dependent. It can be useful when writing custom model rendering functions (see **br_model_custom_cbfn**$_{251}$).

# Operations

## BrModelPick2D()

*Description:* Casts a ray into a model and calls a call-back for all faces that intersect the ray. This can be used in conjunction with **BrScenePick2D()**$_{86}$ to give face/edge/vertex picking.

*Declaration:*
```
int BrModelPick2D(br_model* model,
const br_material* material, const br_vector3* ray_pos,
const br_vector3* ray_dir, br_scalar t_near,
br_scalar t_far, br_modelpick2d_cbfn* callback, void* arg)
```

*Arguments:*

**br_model * model**

Non-NULL pointer to model.

**const br_material * material**

Non-NULL pointer to model's default material.

**const br_vector3 * ray_pos**

Non-NULL pointer to a 3D vector giving a starting position of a pick ray in the model's co-ordinate space.

**const br_vector3 * ray_dir**

Non-NULL pointer to a 3D vector giving the direction of the pick ray.

**br_scalar t_near, t_far**

Coefficients of `ray_dir` defining the section of the ray that should be considered. Intersections outside this section are ignored. `t_near` should be less than `t_far`.

**br_modelpick2d_cbfn * callback**

Non-NULL pointer to call-back function to be called for each face intersecting the specified ray section.

**void * arg**

An optional argument to pass to the call-back function.

*Preconditions:* Between **BrBegin()**$_{10}$ and **BrEnd()**$_{11}$. BRender has completed initialisation. The specified model and material have been updated.

*Effects:* The model's geometry is scanned to find faces that intersect the specified ray section. The supplied call-back is called for each intersecting face.

*Result:* **int**

If the call-back returns a non-zero value, traversal halts and that value is returned. Otherwise, zero is returned.

*Remarks:* This function is typically used within **BrScenePick2D()** [86] to refine selection further down to the face level. Perspective texture co-ordinates are also available to provide such things as 3D texture editing, or surface controls.

*Example:* The following code gives an example of call-backs that enable changing the material assigned to a face of a (non-inherited) model under a particular screen pixel.

```
int BR_CALLBACK MyPickNearestModelCallback(br_model* model, const
   br_material* material, const br_vector3* ray_pos, const
   br_vector3* ray_dir, br_scalar t, int f,int e, int v, const
   br_vector3* p, const br_vector2* map, my_pick_nearest* pn)
{  if (t<pn->t)/* has its own model & nearer */
   {  pn->t=t;
      pn->actor=pn->temp_actor;
      pn->model=model;
      pn->material=material;
      pn->point=*p;
      pn->face=f;
      pn->edge=e;
      pn->vertex=v;
      pn->map=*map;
   }
   return 0;
}

/* Test callback */
int BR_CALLBACK MyPickNearestCallback(br_actor* actor, const
   br_model* model, const br_material* material, const br_vector3*
   ray_pos, const br_vector3* ray_dir, br_scalar t_near,
   br_scalar t_far, my_pick_nearest* pn)
{  pn->temp_actor=actor;
   if (actor->model)
      BrModelPick2D(actor->model, material, ray_pos, ray_dir,
   t_near, t_far, MyPickNearestModelCallback, pn);
   return 0;
}
...
BrScenePick2D(test_world, observer, back_buffer, PickCursor_X,
   PickCursor_Y, MyPickNearestCallback, &PickNearest);
if(PickNearest.model)
{  PickNearest.model-
   >faces[PickNearest.face].material=pick_material;
   BrModelUpdate(PickNearest.model,BR_MODU_ALL);
```

```
}
...
```

*See Also:* **BrScenePick2D()** 86

---

# BrModelApplyMap()

*Description:* Generate texture co-ordinates (u,v) for a model's vertices, using a planar, spherical, cylindrical, disc or null mapping. The model's vertices can be pre-transformed by an optional matrix.

*Declaration:* **void BrModelApplyMap(br_model* model, int map_type, const br_matrix34* xform)**

*Arguments:* **br_model * model**

A pointer to a model.

**int map_type**

Mapping type. This determines how a texture is wrapped around a model. Each type is described in the following table.

| Map Type Symbol | Mapping | Texture Co-ordinates (*u*,*v*) |
|---|---|---|
| BR_APPLYMAP_NONE | None | $(0 \quad 0)$ |
| BR_APPLYMAP_PLANE | Planar | $(\frac{1}{2}(x+1) \quad \frac{1}{2}(y+1))$ |
| BR_APPLYMAP_DISC | Disc | $(\frac{1}{2\pi}\arctan\frac{-x}{z} \qquad \sqrt{x^2+y^2})$ |
| BR_APPLYMAP_CYLINDER | Cylindrical | $(\frac{1}{2\pi}\arctan\frac{-x}{z} \qquad \frac{1}{2}(y+1))$ |
| BR_APPLYMAP_SPHERE | Spherical | $\left(\frac{1}{2\pi}\arctan\frac{-x}{z} \quad 1-\frac{1}{\pi}\arctan\frac{y}{\sqrt{x^2+z^2}}\right)$ |

The disc mapping can be visualised by considering a cylindrical mapping, but shrinking one end of the cylinder to a point and then flattening it to form a disc.

The cylindrical mapping, predictably, can be visualised by imagining a texture wrapped around the outside of a cylinder. The spherical mapping is similar to a cylindrical mapping, but the ends of the cylinder are shrunk to single points.

**const br_matrix34 * xform**

A pointer to an optional matrix. If NULL, the identity transformation is used.

*See Also:* **BrModelFitMap()** 238

## BrModelFitMap()

*Description:* Generate a transformation which will map the bounds of a model onto a cube defined by the corner co-ordinates (-1,-1,-1) and (1,1,1). When passed to **BrModelApplyMap()** 235, texture co-ordinates will be generated which fit the model exactly. The two axes along which the mapping is applied must be specified.

*Declaration:*
```
br_matrix34* BrModelFitMap(const br_model* model,
int axis_0, int axis_1, br_matrix34* transform)
```

*Arguments:* **const br_model * model**

A pointer to a model.

**int axis_0, axis_1**

Mapping axes, defined as follows:

| Mapping Axis Symbol | Mapping is Applied Along |
|---|---|
| BR_FITMAP_PLUS_X | Positive x axis |
| BR_FITMAP_PLUS_Y | Positive y axis |
| BR_FITMAP_PLUS_Z | Positive z axis |
| BR_FITMAP_MINUS_X | Negative x axis |
| BR_FITMAP_MINUS_Y | Negative y axis |
| BR_FITMAP_MINUS_Z | Negative z axis |

**br_matrix34 * transform**

A pointer to the destination transformation matrix.

*Result:* **br_matrix34 ***

Returns a pointer to the destination transformation matrix as supplied (for convenience).

*See Also:* **BrModelApplyMap()** 235

## BrSceneModelLight()

*Description:* Generate prelit lighting values for the vertices of a given model, using the current rendering's lighting set-up.

*Declaration:*
```
void BrSceneModelLight(br_model* model,
const br_material* default_material, const br_actor* root,
const br_actor* a)
```

*Arguments:* **br_model * model**

Non-NULL pointer to model to calculate prelit values for.

**const br_material * default_material**

Non-NULL pointer to default material to use for model.

**const br_actor* root**

Pointer to root actor of scene, e.g. as supplied to **BrZbSceneRenderBegin()**[35]. If this function is called from within a custom model call-back, NULL may be used to indicate that the root effective for the current actor is to be used.

**const br_actor* a**

Pointer to actor defining the required co-ordinate space for the model to be prelit. If this function is called from within a custom model call-back, NULL may be used to indicate that the current actor should is to be used.

*Preconditions:* Between **BrBegin()**[10] and **BrEnd()**[11]. Currently rendering, e.g. between **BrZbSceneRenderBegin()**[35] and **BrZbSceneRenderEnd()**[37].

*Effects:* Works out lighting for the supplied model as though it were attached to the supplied actor. Modifies the appropriate prelit members of each vertex in accordance with the model and its material.

*Remarks:* Generally useful for scenes having little change in lighting. Lights may be enabled for the first frame, this function called to pre-light various models and then most or all lighting disabled for performance in subsequent frames. Note that each model whose vertices are so set will need a **BrModelUpdate()**[241] applied before the next rendering.

*See Also:* **br_vertex**[378]

---

## Copy/Assign

The **br_model**[232] structure should not be copied directly, e.g. by structure assignment. If a similar model is required, a new one should be allocated and pertinent members copied individually. Do not copy vertex and face lists by reference unless you have allocated them yourself. Care may be needed in copying identifier.

## Access & Maintenance

Models must be added to the registry if they are involved in rendering a scene. They should not be modified during rendering.

Models that have been added to the registry may be accessed by BRender during rendering.

If any changes are made to models involved in rendering, they must be updated before the next rendering in which they are involved.

## BrModelAdd()

*Description:* Add a model to the registry, updating it as necessary. All models must be added to the registry before they are subsequently involved in rendering.

*Declaration:* **br_model* BrModelAdd(br_model* model)**

*Arguments:* **br_model * model**

A pointer to a model.

*Result:* **br_model ***

Returns a pointer to the added model, else NULL if unsuccessful.

*See Also:* **BrModelUpdate()** 241, **BrModelAddMany()** 240, **BrModelLoad()** 247, **BrModelFind()** 244, **BrModelRemove()** 241.

## BrModelAddMany()

*Description:* Add a number of models to the registry, updating them as necessary.

*Declaration:* **br_uint_32 BrModelAddMany(br_model* const* models, int n)**

*Arguments:* **br_model * const * models**

A pointer to an array of pointers to models.

**int n**

Number of models to add to the registry.

*Result:* **br_uint_32**

Returns the number of models added successfully.

*See Also:* **BrModelUpdate()** 241, **BrModelAdd()** 240, **BrModelRemove()** 241, **BrModelRemoveMany()** 242

# BrModelUpdate()

*Description:* Update a model that has changed in some respect since the previous update of this model (or **BrModelAdd()** 240).

*Declaration:* **void BrModelUpdate(br_model* model, br_uint_16 flags)**

*Arguments:* **br_model * model**

A pointer to a model.

**br_uint_16 flags**

Model update flags. In general, BR_MODU_ALL should be used. However, the following table describes when to use more specific update flags. Note that flags can be combined using the 'Or' operation (BR_MODU_ALL is such a combination of the other flags, for your convenience).

| Update Flag Symbol | When to Use |
|---|---|
| BR_MODU_VERTICES | Any aspect of vertices changes, including: vertex co-ordinates, the number of vertices, pre-lighting values, texture co-ordinates |
| BR_MODU_FACES | Any aspect of faces changes, including: the set of vertices used by all faces, a face's vertex order, a face's set of vertices, a face's smoothing group or flags, the number of faces |
| BR_MODU_MATERIALS | A face's material pointer changes |
| BR_MODU_ALL | An unknown or wholesale change occurs |

*See Also:* **BrModelAdd()** 240.

# BrModelRemove()

*Description:* Remove a model from the registry.

*Declaration:* **br_model* BrModelRemove(br_model* model)**

*Arguments:* **br_model * model**

A pointer to a model.

*Result:* **br_model ***

Returns a pointer to the model removed.

*See Also:* **BrModelAdd()** 240

## BrModelRemoveMany()

*Description:* Remove a number of models from the registry.

*Declaration:* **br_uint_32 BrModelRemoveMany(br_model* const* models, int n)**

*Arguments:* **br_model * const * models**

A pointer to an array of pointers to models.

**int n**

Number of models to remove from the registry.

*Result:* **br_uint_32**

Returns the number of models removed successfully.

*See Also:* **BrModelAddMany()** 240

## Referencing & Lifetime

Models may be multiply referenced. The model may be referenced by more than one model actor as long as its lifetime is longer than the actors that refer to them. Models must have been added to the registry if they will be involved in rendering. The model must be maintained while it is in the registry or being referenced.

## Initialisation

The model is automatically initialised to zero by **BrModelAllocate()** 243. Members should then be set appropriately. Re-initialisation is not recommended – destroy and reconstruct.

## Construction & Destruction

Apart from import and platform specific functions, models should only be constructed by the following BRender function. Destruction should naturally be performed by the corresponding 'free' function, usually **BrModelFree()** 243. Note that a model should be removed from the registry before being destroyed.

## BrModelAllocate()

*Description:* Allocate a new model.

*Declaration:* **br_model* BrModelAllocate(const char* name, int nvertices, int nfaces)**

*Arguments:* **const char * name**

String to initialise the `identifier` member to.

**int nvertices**

Size of vertex list to allocate. This should be set to zero if maintaining the vertex list separately (in which case `BR_MODF_KEEP_ORIGINAL` should be immediately set in the returned structure).

**int nfaces**

Size of face list to allocate. This should be set to zero if maintaining the face list separately (in which case `BR_MODF_KEEP_ORIGINAL` should be immediately set in the returned structure).

*Result:* **br_model ***

Returns a pointer to the new model, or `NULL` if unsuccessful.

## BrModelFree()

*Description:* Deallocate a model and any associated memory.

*Declaration:* **void BrModelFree(br_model* m)**

*Arguments:* **br_model * m**

A pointer to a model.

# Supplementary

## BrModelCount()

*Description:* Count the number of registered models whose names match a given search pattern. The search pattern can include the standard wild cards '*' and '?'.

*Declaration:* **br_uint_32 BrModelCount(const char* pattern)**

*Arguments:* **const char * pattern**

Search pattern.

*Result:* **br_uint_32**

Returns the number of models matching the search pattern.

*See Also:* **BrModelEnum()** 244, **BrModelFind()** 244

---

# BrModelEnum()

*Description:* Calls a call-back function for every model in the registry matching a given search pattern. The call-back is passed a pointer to each matching model, and its second argument is an optional pointer supplied by the user. The search pattern can include the standard wild cards '*' and '?'. The call-back itself returns a **br_uint_32** 358 value. The enumeration will halt at any stage if the return value is non-zero.

*Declaration:* **br_uint_32 BrModelEnum(const char* pattern, br_model_enum_cbfn* callback, void* arg)**

*Arguments:* **const char * pattern**

Search pattern.

**br_model_enum_cbfn * callback**

A pointer to a call-back function.

**void * arg**

An optional argument to pass to the call-back function.

*Result:* **br_uint_32**

Returns the first non-zero call-back return value, or zero if all matching models are enumerated.

*Example:*

```
br_uint_32 BR_CALLBACK test_callback(br_model* model, void* arg)
{  br_uint_32 count;
...
   return(count);
}
...
{  br_uint_32 enum;
...
   enum = BrModelEnum("model",&test_callback,NULL);
}
```

---

# BrModelFind()

*Description:* Find a model in the registry by name. A call-back function can be setup to be called if the search is unsuccessful. The search pattern can include the standard wild cards '*' and '?'.

*Declaration:* **br_model* BrModelFind(const char* pattern)**

*Arguments:* **const char * pattern**

Search pattern.

*Result:* **br_model ***

Returns a pointer to the model if found, otherwise NULL. If a call-back exists and is called, the call-back's return value is returned.

*See Also:* **BrModelFindHook()** 245, **BrModelFindMany()** 245

## BrModelFindMany()

*Description:* Find a number of models in the registry by name. The search pattern can include the standard wild cards '*' and '?'.

*Declaration:*
```
br_uint_32 BrModelFindMany(const char* pattern,
br_model** models, int max)
```

*Arguments:* **const char * pattern**

Search pattern.

**br_model * * models**

A pointer to an array of pointers to models.

**int max**

Maximum number of models to find.

*Result:* **br_uint_32**

Returns the number of models found. The pointer array is filled with pointers to the found models.

*See Also:* **BrModelFind()** 244, **BrModelFindHook()** 245

## BrModelFindHook()

*Description:* Functions to set up a call-back.

*Declaration:*
```
br_model_find_cbfn*
BrModelFindHook(br_model_find_cbfn* hook)
```

*Arguments:* **br_model_find_cbfn * hook**

A pointer to a call-back function.

*Effects:* If **BrModelFind()** 244 is unsuccessful and a call-back has been set up, the call-back is passed the search pattern as its only argument. The call-back should then return a pointer to a substitute or default model.

For example, a call-back could be set up to return a default model if the desired model cannot be found in the registry.

The function **BrModelFindFailedLoad()** 246 is provided and will probably be sufficient in many cases.

*Result:* **br_model_find_cbfn ***

Returns a pointer to the old call-back function.

*Example:*

```
br_model BR_CALLBACK * test_callback(const char* pattern)
{  br_model* default_model;
...
   return(default_model);
}
...
{  br_model* model;
...
   BrModelFindHook(&test_callback);
   model = BrModelFind("non_existent_model");
}
```

*See Also:* **BrModelFindFailedLoad()** 246

# BrModelFindFailedLoad()

*Description:* This function is provided as a suitable function to supply to **BrModelFindHook()** 245.

*Declaration:* **br_model* BrModelFindFailedLoad(const char* name)**

*Arguments:* **const char * name**

The name supplied to **BrModelFind()** 244.

*Effects:* Attempts to load the model from the filing system using `name` as the filename. Searches in current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined). If successful, sets this name as the identifier of the loaded model and adds the model to the registry.

*Result:* **br_model ***

Returns a pointer to the model, if found, else `NULL`.

*Example:*

```
BrModelFindHook(BrModelFindFailedLoad);
```

## Import & Export

---

# BrModelFileCount()

*Description:* Locate a given file and count the number of models in it.

*Declaration:* **br_uint_32 BrModelFileCount(const char* filename, br_uint_16* num)**

*Arguments:* **const char * filename**

Name of the file containing the models to count.

**br_uint_16 * num**

Pointer to the variable in which to store the number of models counted in the file. If NULL, the file will still be located and appropriate success returned, but no count will be made.

*Effects:* Searches for filename, if no path specified with file looks in current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined). If a file is found, will count the number of models stored in it.

*Result:* **br_uint_32**

Returns zero if the file was found (even if it is not a models file), non-zero otherwise.

---

# BrModelLoad()

*Description:* Load a model. Note that it is not added to the registry.

*Declaration:* **br_model* BrModelLoad(const char* filename)**

*Arguments:* **const char * filename**

Name of the file containing the model to load.

*Effects:* Searches for filename, if no path specified with file looks in current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined).

*Result:* **br_model ***

Returns a pointer to the loaded model, or NULL if unsuccessful.

*See Also:* **BrModelLoadMany()** 248, **BrModelSave()** 249, **BrModelAdd()** 240.

## BrModelLoadMany()

*Description:* Load a number of models. Note that they are not added to the registry.

*Declaration:*
```
br_uint_32 BrModelLoadMany(const char* filename,
br_model** models, br_uint_16 num)
```

*Arguments:*

**const char * filename**

Name of the file containing the models to load.

**br_model ** models**

A non-NULL pointer to an array of pointers to models.

**br_uint_16 num**

Maximum number of models to load.

*Effects:* Searches for filename, if no path specified with file looks in current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined).

*Result:* **br_uint_32**

Returns the number of models loaded successfully. The pointer array if supplied, is filled with pointers to the loaded models.

*See Also:* See **BrModelFileCount()** 247 to determine the number of models in a file.

## BrFmtASCLoad()

*Description:* Import 3D studio models (geometry only). The models are neither updated nor registered.

*Declaration:*
```
br_uint_32 BrFmtASCLoad(const char* name,
br_model** mtable, br_uint_16 max_models)
```

*Arguments:*

**const char * name**

Name of the file containing the models.

**br_model ** mtable**

A pointer to an array of pointers to models, which will be filled as they are imported. If NULL, the models are still imported, but must be referenced subsequently by name.

**int max_models**

The maximum number of models to import.

*Effects:* Searches for filename, if no path specified with file looks in current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined).

*Result:* **br_uint_32**

Returns the number of successfully imported models.

## BrFmtNFFLoad()

*Description:* Import a model expressed in the Neutral File Format. The model is neither updated nor registered.

*Declaration:* **br_model* BrFmtNFFLoad(const char* name)**

*Arguments:* **const char * name**

Name of the file containing the model.

*Effects:* Searches for `filename`, if no path specified with file looks in current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined).

*Result:* **br_model ***

Returns a pointer to the imported model, or `NULL` if it could not be imported.

## BrModelSave()

*Description:* Save a model to a file.

*Declaration:* **br_uint_32 BrModelSave(const char* filename, const br_model* model)**

*Arguments:* **const char * filename**

Name of the file to save the model to.

**const br_model * model**

A pointer to a model.

*Effects:* Writes the model to a file*.

*Result:* **br_uint_32**

Returns `NULL` if the model could not be saved.

*See Also:* **BrWriteModeSet()** 64

* Any existing file of the same name is overwritten.

## BrModelSaveMany()

*Description:* Save a number of models to a file.

*Declaration:* **br_uint_32 BrModelSaveMany(const char* filename, const br_model* const* models, br_uint_16 num)**

*Arguments:* **const char * filename**

Name of the file to save the models to.

**const br_model * const * models**

A pointer to an array of pointers to models. If NULL, all registered models are saved (irrespective of num).

**br_uint_16 num**

Number of models to save.

*Effects:* Writes the models to a file*.

*Result:* **br_uint_32**

Returns the number of models saved successfully.

*See Also:* **BrWriteModeSet()** 64

* Any existing file of the same name is overwritten.

# br_model_custom_cbfn

## The Call-Back Function

This type defines a call-back function, primarily intended for the `custom` member of the **br_model**[232] structure. It enables an application to customise rendering for particular models, or to perform computations that require information only obtainable at the time a model is rendered.

### *The* ***typedef***

(See *model.h* for a precise declaration)

```
void br_model_custom_cbfn(br_actor*, br_model*, const br_material*, void*, br_uint_8,
                                                int, const br_matrix34*,
                                                const br_matrix4*)
```
*Custom model renderer*

### *Related Functions*

Functions dedicated for use within rendering call-backs: **Br[Zb|Zs]ModelRender()**[253|254], **BrOnScreenCheck()**[254], **BrOriginToScreenXY()**[255], **BrPointToScreenXY()**[255], **BrPointToScreenXYMany()**[256], **BrOriginToScreenXYZO()**[256], **BrPointToScreenXYZO()**[257], **BrPointToScreenXYZOMany()**[258].

### *Related Structures*

See **br_renderbounds_cbfn**[315] for a post-model rendering call-back. See **br_primitive_cbfn**[315] for a primitive insertion call-back used within the Z-Sort renderer.

## Specification

---

# CBFnModelCustom()

*Description:* An application defined call-back function that is called when a model (whose `custom` member defined as the address of this function) is about to be processed by the rendering engine. If this function does nothing, the model will not be rendered. The pass through equivalent would be for this function to call **Br[Zb|Zs]ModelRender()**[253|254].

*Declaration:*

```
void BR_CALLBACK CBFnModelCustom(br_actor* actor,
br_model* model, const br_material* material,
void* render_data, br_uint_8 style, int on_screen,
const br_matrix34* model_to_view,
const br_matrix4* model_to_screen)
```

*Arguments:* **br_actor * actor**

Pointer to model actor referencing the model referring to this call-back.

**br_model * model**

Pointer to model referring to this call-back.

**const br_material * material**

Pointer to actor's material if defined, or default material otherwise.

**void * render_data**

A pointer to the order table the primitives for this model would be inserted into, if the Z-Sort renderer is used. The value is `NULL` if no data is appropriate for the renderer, e.g. when using the Z-Buffer renderer.

**br_uint_8 style**

Actor's rendering style, or default. BRender will not supply `BR_RSTYLE_DEFAULT` or `BR_RSTYLE_NONE`.

**int on_screen**

On-screen flag (see **BrOnScreenCheck()** 254). The call-back will never be called by BRender with the flag value `OSC_REJECT`.

**const br_matrix34 * model_to_view**

A pointer to a matrix giving the model to view space transformation.

**const br_matrix4 * model_to_screen**

A pointer to a matrix giving the model to screen transformation.

*Preconditions:* BRender has completed initialisation. Rendering is in progress. The model's bounds intersect or are within the viewing volume.

*Effects:* Behaviour is up to the application. **Br[Zb|Zs]ModelRender()** 253|254 or any of the operations described for **br_model_custom_cbfn** 251 can be used.

*Remarks:* Any other BRender functions may be called from within this call-back with the following restrictions:

- Don't call any rendering functions, apart from **Br[Zb|Zs]ModelRender()** 253|254.
- Don't modify any light, clip-plane or camera actors.
- Don't access any output buffers until after rendering has completed.
- Don't change the environment actor.
- For best performance, avoid adding, updating or removing registry items – try to do these things before rendering.
- Do not modify the actor hierarchy

*Example:* Possible uses include:

- Selecting models with different levels of detail according to viewer distance
- Morphing models (**BrModelUpdate()** 241 required)
- Collision detection (not necessarily indicating the best method)
- Labelling
- Rendering liquids, gases, particulate, flames, smoke, etc.

*See Also:* **br_renderbounds_cbfn**$_{315}$, **br_primitive_cbfn**$_{315}$, **br_pick2d_cbfn**$_{272}$, **br_pick3d_cbfn**$_{275}$.

## Operations

The following functions are provided solely for use within ***CBFnModelCustom()***$_{251}$, ***CBFnRenderBounds()***$_{316}$, and ***CBFnPrimitive()***$_{316}$ functions. **Br[Zb|Zs]ModelRender()**$_{253|254}$ is the equivalent of the function that would have been called had the model not specified a call-back (but it shouldn't be specified as the call-back itself). **Br[Zb|Zs]ModelRender()**$_{253|254}$ should not be called within ***CBFnRenderBounds()***$_{316}$ or ***CBFnPrimitive()***$_{316}$.

# BrZbModelRender()

*Description:* Render a model actor as part of the current Z-Buffer rendering.

*Declaration:*
```
void BrZbModelRender(br_actor* actor, br_model* model,
const br_material* material, br_uint_8 style,
int on_screen, int use_custom)
```

*Arguments:*

**br_actor * actor**

The pointer to the model actor referencing the supplied model (must not be NULL).

**br_model * model**

The pointer to the model to be rendered (must not be NULL).

**const br_material * material**

A pointer to the material to use for rendering faces that don't specify a material (must not be NULL).

**br_uint_8 style**

The rendering style to use for rendering the model (any defined style may be specified, even BR_RSTYLE_NONE).

**int on_screen**

A flag specifying whether the model is either partially or completely on-screen (either OSC_PARTIAL or OSC_ACCEPT). If OSC_ACCEPT is specified, even off-screen faces will be rendered. See **BrOnScreenCheck()**$_{254}$.

**int use_custom**

If non-zero, invoke the specified model's custom call-back function. This is typically zero if the same model is specified.

## BrZsModelRender()

*Description:* Render a model actor as part of the current Z-Sort rendering.

*Declaration:* **void BrZsModelRender(br_actor* actor, br_model* model, const br_material* material, br_order_table* order_table, br_uint_8 style, int on_screen, int use_custom)**

*Arguments:* **br_actor * actor**

The pointer to the model actor referencing the supplied model (must not be NULL).

**br_model * model**

The pointer to the model to be rendered (must not be NULL).

**const br_material * material**

A pointer to the material to use for rendering faces that don't specify a material (must not be NULL).

**br_order_table * order_table**

A pointer to the order table the model's primitives should be inserted into (must not be NULL).

**br_uint_8 style**

The rendering style to use for rendering the model (any defined style may be specified, even BR_RSTYLE_NONE).

**int on_screen**

A flag specifying whether the model is either partially or completely on-screen (either OSC_PARTIAL or OSC_ACCEPT). If OSC_ACCEPT is specified, even off-screen faces will be rendered. See **BrOnScreenCheck()** 254.

**int use_custom**

If non-zero, invoke the specified model's custom call-back function. This is typically zero if the same model is specified.

## BrOnScreenCheck()

*Description:* Check a bounding box in the model space against the view volume and any clip-planes.

*Declaration:* **br_uint_8 BrOnScreenCheck(const br_bounds* bounds)**

*Arguments:* **const br_bounds * bounds**

A pointer to a **br_bounds** *108* structure giving the bounding box dimensions in the model's co-ordinate space.

*Result:* **br_uint_8**

Returns one of the following:

| Flag Symbol | Meaning |
|---|---|
| OSC_REJECT | The model is entirely outside the viewing volume |
| OSC_PARTIAL | The model is partially within the viewing volume |
| OSC_ACCEPT | The model is entirely within the viewing volume |

## BrOriginToScreenXY()

*Description:* Transform and project the origin in the model's co-ordinate space onto the screen.

*Declaration:* **br_uint_8 BrOriginToScreenXY(br_vector2* screen)**

*Arguments:* **br_vector2 * screen**

A pointer to the destination vector to receive the co-ordinates of the point in projected screen space (See Projected screen space, page 23). Note that the point is not necessarily on screen, i.e. inside the bounds of the output pixel map.

*Result:* **br_uint_8**

Returns zero if the point is in front of the eye (the viewing pyramid).

## BrPointToScreenXY()

*Description:* Transform and project a single point in the model's co-ordinate space onto the screen.

*Declaration:* **br_uint_8 BrPointToScreenXY(br_vector2* screen, const br_vector3* point)**

*Arguments:* **br_vector2 * screen**

A pointer to the destination vector to receive the co-ordinates of the point in projected screen space (See Projected screen space, page 23). Note that the point is not necessarily on screen, i.e. inside the bounds of the output pixel map.

**const br_vector3 * point**

A pointer to the source vector containing the co-ordinates of the point to project.

*Result:* **br_uint_8**

Returns zero if the point is in front of the eye (the viewing pyramid).

## BrPointToScreenXYMany()

*Description:* Transform and project a number of points in the model's co-ordinate system onto the screen.

*Declaration:* **void BrPointToScreenXYMany(br_vector2* screens, const br_vector3* points, br_uint_32 npoints)**

*Arguments:* **br_vector2 * screens**

A pointer to an array of destination vectors to receive the co-ordinates of each point in projected screen space (See Projected screen space, page 23). Note that the points are not necessarily on screen, i.e. inside the bounds of the output pixel map.

**const br_vector3 * points**

A pointer to an array of source vectors containing co-ordinates of points in model space.

**br_uint_32 npoints**

Number of points to process.

*Effects:* Equivalent to a call of **BrPointToScreenXY()**$_{255}$ for each screen and point vector.

## BrOriginToScreenXYZO()

*Description:* Transform and project the origin in the model's co-ordinate space onto the screen, generating x, y and z co-ordinates. If it is off-screen, it is not projected.

*Declaration:* **br_uint_32 BrOriginToScreenXYZO(br_vector3* screen)**

*Arguments:* **br_vector3 * screen**

A pointer to the destination vector to receive the co-ordinates of the point in projected screen space (See Projected screen space, page 23).

*Effects:* The origin is checked against the viewing volume. If within, the equivalent screen and depth buffer co-ordinates are calculated, otherwise the function has no effect.

*Result:* **br_uint_32**

If the co-ordinates have been placed in the destination vector the function returns zero, otherwise the origin is off-screen, in which case its out-code, made up from a combination of the flags in the following table is returned.

| Out-Code Flag | Flag Value | Set When Origin Is |
|---|---|---|
| OUTCODE_LEFT | 0x01 | Outside left plane |
| OUTCODE_RIGHT | 0x02 | Outside right plane |
| OUTCODE_TOP | 0x04 | Outside top plane |

| | | |
|---|---|---|
| OUTCODE_BOTTOM | 0x08 | Outside bottom plane |
| OUTCODE_HITHER | 0x10 | Outside hither plane |
| OUTCODE_YON | 0x20 | Outside yon plane |

## BrPointToScreenXYZO()

*Description:* Transform and project a point in the model's co-ordinate space onto the screen, generating x, y and z co-ordinates. If it is off-screen, it is not projected.

*Declaration:* **br_uint_32 BrPointToScreenXYZO(br_vector3* screen, const br_vector3* point)**

*Arguments:* **br_vector3 * screen**

A pointer to the destination vector to receive the co-ordinates of the point in projected screen space (See Projected screen space, page 23).

**const br_vector3 * point**

A pointer to the source vector.

*Effects:* The point is checked against the viewing volume. If within, the equivalent screen and depth buffer co-ordinates are calculated, otherwise the function has no effect.

*Result:* **br_uint_32**

If the co-ordinates have been placed in the destination vector the function returns zero, otherwise the point is off-screen, in which case its out-code, made up from a combination of the flags in the following table is returned.

| Out-Code Flag | Flag Value | Set When Point Is |
|---|---|---|
| OUTCODE_LEFT | 0x01 | Outside left plane |
| OUTCODE_RIGHT | 0x02 | Outside right plane |
| OUTCODE_TOP | 0x04 | Outside top plane |
| OUTCODE_BOTTOM | 0x08 | Outside bottom plane |
| OUTCODE_HITHER | 0x10 | Outside hither plane |
| OUTCODE_YON | 0x20 | Outside yon plane |

*See Also:* **BrOriginToScreenXYZO()** 256

## BrPointToScreenXYZOMany()

*Description:* Transform and project a number of points in the model's co-ordinate space onto the screen, generating a series x, y and z co-ordinates and out-codes. All those that are off-screen are not projected.

*Declaration:*
```
void BrPointToScreenXYZOMany(br_vector3* screens,
br_uint_32* outcodes, const br_vector3* points,
br_uint_32 npoints)
```

*Arguments:*

**br_vector3 * screens**

A pointer to a list of destination vectors, each of which will receive co-ordinates of each point in projected screen space (See Projected screen space, page 23).

**br_uint_32 * outcodes**

A pointer to an array of out-codes for each point. If the co-ordinates have been placed in the corresponding destination vector the out-code will be zero, otherwise the point is off-screen, in which case only its out-code (see **BrPointToScreenXYZO()** 257) is stored.

**const br_vector3 * points**

A pointer to an array of source vectors containing the points to be projected.

**br_uint32 npoints**

Number of points.

*Effects:* Equivalent to a call of **BrPointToScreenXYZO()** 257 for each point. Only co-ordinates of points in the viewing volume are written to corresponding elements of `screens`.

*See Also:* **BrOriginToScreenXYZO()** 256

# br_model_enum_cbfn

## The Call-Back Function

This type defines a function, supplied to **BrModelEnum()** 244, and to be called by it for a selection of models.

### *The* ***typedef***

(See *fwproto.h* for a precise declaration)

```
br_uint_32              br_model_enum_cbfn(br_model*, void*)Enumerator
```

## Specification

---

## CBFnModelEnum()

| | |
|---|---|
| *Description:* | An application defined call-back function accepting a model and an application supplied argument (as supplied to **BrModelEnum()** 244). |
| *Declaration:* | **br_uint_32 BR_CALLBACK CBFnModelEnum(br_model* model, void* arg)** |
| *Arguments:* | **br_model * model**<br>One of the models selected by **BrModelEnum()** 244.<br>**void * arg**<br>The argument supplied to **BrModelEnum()** 244. |
| *Preconditions:* | BRender has completed initialisation. |
| *Effects:* | Application defined. Avoid adding or removing models within this function. |
| *Result:* | **br_uint_32**<br>Any non-zero value will terminate the enumeration and be returned by **BrModelEnum()** 244. Return zero to continue the enumeration. |
| *See Also:* | **BrModelEnum()** 244, **BrModelFind()** 244. |

---

# br_model_find_cbfn

## The Call-Back Function

This type defines a function, registered with **BrModelFindHook()** 245, to be called when **BrModelFind()** 244 or **BrModelFindMany()** 245 fail to find any model.

### *The* ***typedef***

(See *fwproto.h* for a precise declaration)

**br_model*** **br_model_find_cbfn(const char*)** ***Find (when*** **BrModelFind()** ***fails)***

## Specification

---

## CBFnModelFind()

*Description:* An application defined call-back function used when **BrModelFind()** 244 or **BrModelFindMany()** 245 fail.

*Declaration:* **br_model* BR_CALLBACK CBFnModelFind(const char* name)**

*Arguments:* **const char * name**

The search pattern supplied to **BrModelFind()** 244 or **BrModelFindMany()** 245 that did not match any model.

*Preconditions:* BRender has completed initialisation. No model has an identifier that successfully matches the search pattern.

*Effects:* Application defined.

*Result:* **br_model ***

Either return an existing model that is deemed appropriate for the search pattern, or NULL if there isn't one. This value will be returned by **BrModelFind()** 244 or **BrModelFindMany()** 245.

*Remarks:* This could either be used to supply a default model or to create a model. If models were created on demand, then this function could search another list of available models (but not yet created) and see if the pattern matched any of them, if it did, one of them could be registered and returned. Note that there is no way to supply more than one model.

*See Also:* **BrModelFind()** 244, **BrModelFindMany()** 245, **BrModelFindHook()** 245, **BrModelFindFailedLoad()** 246.

---

# br_modelpick2d_cbfn

## The Call-Back Function

This type defines a call-back function, which is called by **BrModelPick2D()** 235. It is called for each face of a particular model that intersects a given ray. It's typically used to implement face picking functions for user interfaces.

### *The* ***typedef***

(See *fwproto.h* for a precise declaration)

```
int br_modelpick2d_cbfn(br_model*, const br_material*, const br_vector3*,
const br_vector3*, br_scalar, int, int, int, const br_vector3*, const br_vector2*,
void*)
```
***Face pick***

### *Related Functions*

The function that invokes this call-back is **BrModelPick2D()** 235.

For a function that calls a call-back for each model actor beneath a specific pixel, see **BrScenePick2D()** 86.

### *Related Structures*

See **br_pick2d_cbfn** 272 for a call-back from which this one is typically called.

See **br_model_custom_cbfn** 251 for details of functions (used from within rendering call-backs) that can convert model co-ordinates into screen co-ordinates, e.g. **BrPointToScreenXY()** 255.

## Specification

---

# CBFnModelPick2D()

*Description:* An application defined call-back function that is called by **BrModelPick2D()** 235. It is called for each face that intersects a specific section of a particular ray in a model's co-ordinate space.

*Declaration:*

```
int BR_CALLBACK CBFnModelPick2D(br_model* model,
const br_material* material, const br_vector3* ray_pos,
const br_vector3* ray_dir, br_scalar t, int face,
int edge, int vertex, const br_vector3* p,
const br_vector2* map, void* arg)
```

*Arguments:* **br_model * model**

Pointer to the model to pick from.

**const br_material * material**

Pointer to the default material attributed to the model.

**const br_vector3 * ray_pos**

Pointer to a 3D vector giving a starting position of the pick ray in the model's co-ordinate space.

**const br_vector3 * ray_dir**

Pointer to a 3D vector giving direction of ray from view-point through pixel in the model's co-ordinate space.

**br_scalar t**

The co-efficient giving the position of the intersection of the face with the pick ray. The position is supplied in `p`.

**int face**

The index of the face intersecting the ray. See `faces` of **br_model**$_{232}$.

**int edge**

The index giving the edge nearest the intersection point. This is the edge from `vertices[edge]` to `vertices[(edge+1)%3]`. See **br_face**$_{125}$.

**int vertex**

The index giving the vertex nearest the intersection point. This is the vertex `vertices[vertex]` in the **br_face**$_{125}$ structure.

**const br_vector3 * p**

The position of the point at which the ray intersects the face, obtained using the following formula.

$$P_{intersect} = P_{ray} + t_{intersect}\mathbf{d}_{ray}$$

**const br_vector2* map**

The texture co-ordinates of the intersection point on the face. Apply the (perspective correct) material's texture map transform to obtain the row and column indices into the texture map.

**void * arg**

The corresponding value specified in the call of **BrModelPick2D()**$_{235}$ that invoked this call-back.

*Preconditions:* The function is called from within **BrModelPick2D()**$_{235}$.

*Effects:* Behaviour is up to the application.

*Result:* **int**

Return zero to continue the search for intersecting faces, non-zero to terminate. A non-zero value will be returned by **BrModelPick2D()**$_{235}$.

*Example:* Possible uses include:

- Face, Edge and Vertex Selection

*See Also:* **br_model**$_{232}$, **br_pick2d_cbfn**$_{272}$, **br_pick3d_cbfn**$_{275}$.

# br_order_table

## The Structure

This is an order table used by the Z-Sort renderer. It contains a table of pointers to linked lists of rendering primitives. The Z depth range of the order table is divided into a number of equal intervals (buckets). Each primitive generated by the Z-Sort renderer is inserted into the appropriate list so that it is eventually rendered in order of depth. It is the application developer's task to determine a scheme so that the scene is crudely segmented into layers of background through to foreground (with an order table for each layer), and any mutually depth critical faces are sorted in order. It does not matter in which order non-overlapping faces are drawn with respect to each other, so the trick is to sort only overlapping faces that need sorting, i.e. concave models, or interpenetrating models.

There is a facility of nominating a primary order table such that all other secondary order tables are rendered in between its buckets. This provides a quick and easy way of progressing to the next step, after having a single order table, i.e. a primary order table for static features and individual order tables for models.

Order tables are assigned to actors, either by default, inheritance or explicitly (see **BrZsActorOrderTableSet()**[89]).

### *The* ***typedef***

(See *order.h* for precise declaration and ordering)

#### Parameters

| | | |
|---|---|---|
| **br_scalar** | **min_z** | ***Minimum z depth of range*** |
| **br_scalar** | **max_z** | ***Maximum z depth of range*** |
| **br_scalar** | **sort_z** | ***Depth at which order table should be sorted*** |
| **br_uint_16** | **size** | ***Number of buckets*** |
| **br_uint_16** | **type** | ***Sort type*** |
| **br_uint_32** | **flags** | ***Various flags*** |

#### Internal

| | | |
|---|---|---|
| **br_primitive **** | **table** | ***Order table array (of primitive linked-lists)*** |
| **br_order_table *** | **next** | ***Next order table*** |
| **br_uint_16** | **visits** | ***Number of times order table has been referenced so far*** |

### *Related Functions*

#### Scene Modelling

**BrZsActorOrderTableSet()**[89], **BrZsActorOrderTableGet()**[90].

#### Scene Rendering

**BrZsPrimitiveCallbackSet()**[39], **BrZsPrimitiveBucketSelect()**[318].

*Related Structures*

Scene Rendering

**br_primitive**[312], **br_primitive_cbfn**[315].

# Members

*Parameters*

## br_scalar min_z, max_z

Range of z depths over which to sort. Any z depths of vertices outside this range will be clamped to the limits. Note that the z depth values are a result of a linear mapping from camera co-ordinate space, of the range [-hither_z, -yon_z] to the range [-hither_z, +yon_z]. These members are usually set by using the BR_ORDER_TABLE_NEW_BOUNDS or BR_ORDER_TABLE_INIT_BOUNDS flags, but can be set explicitly using the **BrZsScreenZToDepth()**[33] functions. Note that **BrOriginToScreenXYZO()**[256], **BrPointToScreenXYZO()**[257] and **BrPointToScreenXYZOMany()**[258] return projected screen z ordinates, not depths.

The z range (max_z-min_z) must be greater than 0.001, otherwise it will be forced to this range.

## br_scalar sort_z

This member defines the value of z used to determine the position at which this order table is in the list of order tables. This value is typically set to min_z, and may be set explicitly or automatically at the same time as min_z and max_z. There are flags available which can be used to specify how sort_z is calculated.

Note that this member is not relevant to primary order tables.

## br_uint_16 size

This member defines the number of intervals (buckets) that the z range is divided into. It also defines the number of pointers pointed to by table.

An order table used for a single opaque convex model need only have one bucket, whereas a complex model actor with several child models (within the z range of the parent) might need several intervals.

## br_uint_16 type

How primitives are inserted into an order table, obviously depends upon their depth, but for primitives with more than one z value, such as lines, triangles and quads, how to arrive at a single z value is not a clear cut decision. Because it depends so much upon the scene and the application, it is left to the application to specify the way the sorting z should be obtained. The following table describes each type which can be specified in this member.

| Sort Type | Z for Bucket Determination |
|---|---|
| BR_SORT_FIRST_VERTEX | Arbitrary – the first vertex is as good as any |
| BR_SORT_MIN | The nearest vertex |
| BR_SORT_MAX | The furthest vertex |
| BR_SORT_AVERAGE | The average depth of vertices |

## br_uint_32 flags

At some point the z range of each order table must be set appropriately. The z range is not only used to determine into which buckets each primitive should be placed, but also in which order the order tables should be rendered. If the application knows in advance what this range can be then the z range can be set before rendering begins (see **BrZsScreenZToDepth()** 33), otherwise various flags can be used to get the z range to be calculated automatically. Note that an order table is used in descendant order, that is, actors nearer the root of the hierarchy will see the order table first. Once the hierarchy has been traversed, the order tables will be sorted according to the value of sort_z, which can be set according to the values of min_z and max_z. The order tables will be rendered furthest first (see also **BrZsOrderTablePrimaryEnable()** 268).

The flags are described in the following table.

| Flag | Behaviour |
|---|---|
| BR_ORDER_TABLE_LEAVE_BOUNDS | Z range is left unchanged (presumably preset) |
| BR_ORDER_TABLE_NEW_BOUNDS | Z range is recalculated from the bounds of each actor that uses it, just before each use. Note that the order table is sorted according to the last value of sort_z. |
| BR_ORDER_TABLE_INIT_BOUNDS | Z range is calculated from the bounds of the first actor that uses it (e.g. a BR_ACTOR_BOUNDS_CORRECT actor). |
| BR_ORDER_TABLE_CONVEX | Z range is ignored for bucket determination (primitives placed in first bucket, and not sorted), but other flags still apply for ordering the order table. |
| BR_ORDER_TABLE_SORT_NEAR | sort_z is set to min_z when the bounds are calculated |
| BR_ORDER_TABLE_SORT_CENTRE | sort_z is set to (min_z+max_z)/2 when the bounds are calculated |
| BR_ORDER_TABLE_SORT_FAR | sort_z is set to max_z when the bounds are calculated |

The first three flags are mutually exclusive, as are the last three. Any one of the first three may be combined with the fourth, and any one of the last three.

Bounds can be computed from the following actors: BR_ACTOR_MODEL, BR_ACTOR_BOUNDS and BR_ACTOR_BOUNDS_CORRECT.

Note that when an order table's bounds are calculated, they are always calculated from the full bounds of the model (or bounds data structure). Therefore, it is possible, where bounds intersect the boundaries of the viewing volume, for one of the order table bounds to lie outside the view space. All buckets are still rendered, but the buckets outside the view space will be empty (unless a custom primitive call-back has utilised them).

*Internal*

### br_primitive ** table

A pointer to a list of heads of linked lists of primitives. Each linked list is effectively each bucket of the bucket sort (per order table). This member is for internal use only. Each pointer will be NULL if the bucket is empty. Note that primitives are allocated from the primitive heap supplied at the start of rendering

### br_order_table * next

Pointer to the next order table (next lowest min_z). NULL at end of list. Each rendering, this list is constructed (before drawing), and then deconstructed (after drawing). This member should only be non-NULL during rendering, i.e. within a rendering call-back function such as **CBFnModelCustom()**[251]. This member is for internal use only.

### br_uint_16 visits

This member is used to determine when to clear and initialise the order table's z range (if required). This member is set to zero upon allocation and after rendering, and incremented each time it is referenced by an actor traversed during rendering. When zero, the order table is cleared (and if BR_ORDER_TABLE_INIT_BOUNDS is set, its z range calculated) when it is next referenced by an actor traversed during rendering. This member is for internal use only.

## Operations

---

## BrZsOrderTablePrimitiveInsert()

*Description:* Insert a rendering primitive into an order table.

*Declaration:*
```
void
BrZsOrderTablePrimitiveInsert(br_order_table* order_table
, br_primitive* primitive, br_uint_16 bucket)
```

*Arguments:* **br_order_table * order_table**

Non-NULL pointer to order table in which to insert primitive (supplied as argument to ***CBFnPrimitive()*** 316).

**br_primitive * primitive**

Non-NULL pointer to primitive to insert (supplied as argument to ***CBFnPrimitive()*** 316).

**br_uint_16 bucket**

Bucket of order table in which to insert primitive (between 0 and order_table->size-1). See **BrZsPrimitiveBucketSelect()** 318.

*Preconditions:* Between **BrZsSceneRenderBegin()** 36 & **BrZsSceneRenderEnd()** 38. Within a primitive call-back function.

*Effects:* The primitive is inserted into the order table in the specified bucket.

*Remarks:* This function is only provided to customise the ordering of primitives in a particular order table. It is not intended to permit automatic generation of primitives.

Note that order tables are marked for clearing after rendering and allocation. Upon entry to a primitive call-back, the order table will have been cleared, and may already contain primitives. The state of other order tables is undefined.

*See Also:* **BrZsModelRender()** 254.

---

# BrZsOrderTablePrimaryEnable()

*Description:* Enable the use of a primary order table, between whose buckets all other order tables are then rendered. By default, no primary order table is enabled.

*Declaration:*
```
void
BrZsOrderTablePrimaryEnable(br_order_table* order_table)
```

*Arguments:* **br_order_table * order_table**

A pointer to the order table to be used as the primary order table. If NULL is supplied, the default order table will be used.

*Preconditions:* Between **BrZsBegin()** 28 & **BrZsEnd()** 40.

*Effects:* The Z sort renderer will use the specified order table as a primary order table, making other order tables secondary. All secondary order tables are rendered sequentially, as appropriate, between the rendering of the buckets of the primary order table. Secondary order tables whose sort_z lies further away than max_z are rendered first (in order of their sort_z). Then the furthest bucket of the primary order table is rendered, followed by the secondary order tables whose sort_z lies within that bucket (in order of their sort_z). This continues for each bucket of the primary order table. Finally, the secondary order tables whose sort_z is in front of the nearest bucket of the primary order table are rendered (in order of their sort_z).

*Remarks:* Note that the primary order table facility only provides a relatively coarse way of merging the ordering of primitives (or buckets) in secondary order tables with those in the primary order table. Nevertheless, with a bit of care, it often gives satisfactory results for a wide range of scenes, with relatively straightforward allocation of order tables.

*See Also:* **BrZsOrderTablePrimaryDisable()** 269.

## BrZsOrderTablePrimaryDisable()

*Description:* Disable the use of a primary order table. Note that the primary order table is disabled by default.

*Declaration:* **void BrZsOrderTablePrimaryDisable()**

*Preconditions:* Between **BrZsBegin()** 28 & **BrZsEnd()** 40.

*Effects:* Order tables are rendered sequentially in order of their `sort_z`.

*Remarks:* The simpler Z sort algorithm that this function engages, generally requires a deal of care and effort to ensure correct rendering order, especially in complex scenes. The results generally produce a more reliable ordering than with a primary order table, as so much more care is needed.

*See Also:* **BrZsOrderTablePrimaryEnable()** 268.

### Copy/Assign

Structure assignment is not recommended. Copy members as appropriate.

### Access & Maintenance

Members should only be modified with care during rendering. Do not modify primitives via `table`, or the linked list of order tables via `next`. Order tables are automatically cleared at the start of each rendering, but can be cleared thereafter (if required) using **BrZsOrderTableClear()** 269.

## BrZsOrderTableClear()

*Description:* Clear an order table, re-initialising its table of pointers to primitives.

*Declaration:* **void BrZsClearOrderTable(br_order_table* order_table)**

*Arguments:* **br_order_table * order_table**

A non-`NULL` pointer to the order table to be cleared.

*Preconditions:* Between **BrZsBegin()** 28 & **BrZsEnd()** 40.

*Effects:* The series of void pointers pointed to by `table` are reset to `NULL`. Some private flags may be modified.

*Remarks:* Note that order tables are marked for clearing after rendering and allocation. Upon entry to a primitive call-back, the order table will have been cleared, and may already contain primitives. Given automatic clearing, the use of this function is not generally required.

*See Also:* **BrZsActorOrderTableGet()** 90.

## Referencing & Lifetime

Order tables can be referred to any number of times by an actor (through use of **BrZsActorOrderTableSet()** 89). Order tables must be valid throughout rendering, i.e. between **BrZsSceneRenderBegin()** 36 and **BrZsSceneRenderEnd()** 38. Note that primitive data is re-used from the primitive heap (supplied to **BrZsBegin()** 28) each rendering. Therefore, it is not possible to incorporate primitives from an order table generated in one scene into another scene.

There is a default order table which has 256 buckets and spans the depth of the view volume (its z range is set from the camera used for the rendering). It thus has just the flag `BR_ORDER_TABLE_LEAVE_BOUNDS` set. Do not attempt to modify the default order table - create your own.

## Initialisation

Allocation performs most initialisation. The only remaining initialisation required is to set the z range (`min_z` and `max_z`) and the z sort position (`sort_z`), which only needs to be done if these member will not be set by using flags such as `BR_ORDER_TABLE_NEW_BOUNDS` and `BR_ORDER_TABLE_INIT_BOUNDS`.

## Construction & Destruction

Order tables should be constructed using **BrZsOrderTableAllocate()** 270 and destroyed using **BrZsOrderTableFree()** 271.

# BrZsOrderTableAllocate()

*Description:* Allocate a new order table.

*Declaration:*
```
br_order_table* BrZsOrderTableAllocate(br_uint_16 size,
br_uint_32 flags, br_uint_16 type)
```

*Arguments:* **br_uint_16 size**

Number of buckets in order table (See br_uint_16 size, page 265).

**br_uint_32 flags**

Order table flags (See br_uint_32 flags, page 266).

**br_uint_16 type**

Order table type (See br_uint_16 type, page 266).

*Preconditions:* Between **BrZsBegin()** 28 & **BrZsEnd()** 40.

Memory can be allocated.

*Effects:* A **br_order_table** *264* data structure is allocated (using **BrResAllocate()** 48, from the BR_MEMORY_RENDER_DATA memory class) and initialised (See **br_order_table** *264*, Initialisation). The z range and sort z are initialised to zero.

*Result:* **br_order_table ***

A pointer to the new **br_order_table** *264* data structure.

*Remarks:* Ensure min_z, max_z and sort_z are set appropriately if they will not be initialised automatically, i.e. if the BR_ORDER_TABLE_LEAVE_BOUNDS flag is set.

*See Also:* **BrZsOrderTableFree()** 271

---

# BrZsOrderTableFree()

*Description:* Free an order table.

*Declaration:* **void BrZsOrderTableFree(br_order_table* order_table)**

*Arguments:* **br_order_table * order_table**

A non-NULL pointer to an order table previously allocated using **BrZsOrderTableAllocate()** 270.

*Preconditions:* Between **BrZsBegin()** 28 & **BrZsEnd()** 40. Not currently rendering.

*Effects:* Uses **BrResFree()** 51 to release storage.

*Remarks:* Ensure that no actor is currently using the order table before using this function. Do not attempt to free an active order table, such as is passed as an argument to a rendering call-back function, e.g. ***CBFnPrimitive()*** 316.

*See Also:* **BrZsOrderTableAllocate()** 270, **BrResFree()** 51.

---

# br_pick2d_cbfn

## The Call-Back Function

This type defines a call-back function, which is called by **BrScenePick2D()**[86]. It is called for each model actor in an actor hierarchy that is beneath a particular screen pixel (corresponding to a particular camera). It's typically used to implement picking functions for user interfaces.

### *The **typedef***

(See *fwproto.h* for a precise declaration)

```
int br_pick2d_cbfn(br_actor*, const br_model*, const br_material*, const br_vector3*,
                   const br_vector3*, br_scalar, br_scalar, void*)
```
***2D pick call-back***

### *Related Functions*

The function that invokes this call-back is **BrScenePick2D()**[86].

For a function that calls a call-back for each model actor whose bounds intersect a specific bounds, see **BrScenePick3D()**[84].

### *Related Structures*

See **br_pick3d_cbfn**[275] for a similar 3D call-back.

See **br_model_custom_cbfn**[251] for details of functions (used from within rendering call-backs) that can convert model co-ordinates into screen co-ordinates, e.g. **BrPointToScreenXY()**[255].

## Specification

---

# CBFnPick2D()

*Description:* An application defined call-back function that is called by **BrScenePick2D()**[86]. It is called for each model actor whose bounds intersect the ray passing from a camera through a particular screen pixel (see **BrScenePick2D()**[86]).

*Declaration:*
```
int BR_CALLBACK CBFnPick2D(br_actor* a,
const br_model* model, const br_material* material,
const br_vector3* ray_pos, const br_vector3* ray_dir,
br_scalar t_near, br_scalar t_far, void* arg)
```

*Arguments:* **br_actor * a**

Pointer to model actor whose model bounds intersect the pick ray.

**const br_model * model**

Pointer to the model attributed to the model actor whose bounds intersect the pick ray (may be an inherited model).

**const br_material * material**

Pointer to the default material attributed to the model actor that may be used by the model (may be an inherited material).

**const br_vector3 * ray_pos**

Pointer to a 3D vector giving a starting position of the pick ray in the model's co-ordinate space. This position has no special significance, i.e. it is not necessarily the position of the view-point.

**const br_vector3 * ray_dir**

Pointer to a 3D vector giving direction of ray from view-point through pixel in the model's co-ordinate space. The magnitude has no special significance, i.e. it is not necessarily the position of the model.

**br_scalar t_near**

The co-efficient giving the position of the entry point of the pick ray into the bounds of the intersecting model, in the model's co-ordinate space. To obtain this position refer to the following formula.

$$P_{entry} = P_{ray} + t_{near}\mathbf{d}_{ray}$$

**br_scalar t_far**

The co-efficient giving the position of the exit point of the pick ray out of the bounds of the intersecting model, in the model's co-ordinate space. To obtain this position refer to the following formula.

$$P_{exit} = P_{ray} + t_{far}\mathbf{d}_{ray}$$

**void * arg**

The corresponding value specified in the call of **BrScenePick2D()**[86] that invoked this call-back.

*Preconditions:* BRender has completed initialisation. A model's bounds intersect the pick ray. The order in which intersections are computed is undefined.

*Effects:* Behaviour is up to the application.

*Result:* **int**

Return zero continue the search for intersecting model actors, non-zero to terminate. A non-zero value will be returned by **BrScenePick2D()**[86].

*Remarks:* For borderline cases, the pick ray is defined to be such that if a model's rendering would appear in the pixel then the model's bounds will intersect the ray. The precise sub-pixel position of the ray is consistent, but undefined.

*Example:* Possible uses include:

- Selection
- Manipulation

*See Also:* **br_model_custom_cbfn**$_{251}$, **br_renderbounds_cbfn**$_{315}$, **br_pick3d_cbfn**$_{275}$.

---

# br_pick3d_cbfn

## The Call-Back Function

This type defines a call-back function, which is called by **BrScenePick3D()**$_{84}$. It is called for each model actor in an actor hierarchy whose bounds intersect a given bounds with respect to a particular actor's co-ordinate space.

### *The* ***typedef***

(See *fwproto.h* for a precise declaration)

```
int br_pick3d_cbfn(br_actor*, const br_model*, const br_material*, const br_matrix34*,
                                                const br_bounds*, void*)  3D pick
```

### *Related Functions*

The function that invokes this call-back is **BrScenePick3D()**$_{84}$.

For a function that calls a call-back for each model actor whose bounds intersect a ray from view-point through a particular screen pixel, see **BrScenePick3D()**$_{84}$.

### *Related Structures*

See **br_pick2d_cbfn**$_{272}$ for a similar 2D call-back.

See **br_model_custom_cbfn**$_{251}$ for details of functions (used from within rendering call-backs) that can convert model co-ordinates into screen co-ordinates, e.g. **BrPointToScreenXY()**$_{255}$.

## Specification

---

# CBFnPick3D()

*Description:* An application defined call-back function that is called by **BrScenePick3D()**$_{84}$. It is called for each model actor whose bounds intersect a given bounds with respect to a particular actor's co-ordinate space (see **BrScenePick3D()**$_{84}$).

*Declaration:*

```
int BR_CALLBACK CBFnPick3D(br_actor* a,
const br_model* model, const br_material* material,
const br_matrix34* transform, const br_bounds* bounds,
void* arg)
```

*Arguments:* **br_actor * a**

Pointer to model actor whose model bounds intersect the specified bounds.

**const br_model * model**

Pointer to model whose bounds intersect the pick ray.

**const br_material * material**

Pointer to the default material attributed to the model actor that may be used by the model (may be an inherited material).

**const br_matrix34 * transform**

A pointer to a matrix transforming the intersecting actor's model co-ordinates into the co-ordinate space of the reference actor (as supplied to **BrScenePick3D()**$_{84}$).

**const br_bounds * bounds**

A pointer to the original bounds (in the reference actor's co-ordinate space).

**void * arg**

The corresponding value specified in the call of **BrScenePick3D()**$_{84}$ that invoked this call-back.

*Preconditions:* BRender has completed initialisation. A model's bounds intersect the bounds in the reference actor's co-ordinate space.

*Effects:* Behaviour is up to the application.

*Example:* Possible uses include:

- Collision detection (does not necessarily indicate the best method)
- Volumetric selection

*See Also:* **br_model_custom_cbfn**$_{251}$, **br_renderbounds_cbfn**$_{315}$, **br_pick2d_cbfn**$_{272}$.

# br_pixelmap

## The Structure

BRender's pixel map structure, used for texture maps, shade tables, blend tables, colour buffers and Z-buffers. See Image Support. Texture maps, shade tables, and blend tables being required for rendering materials should be maintained within the registry as necessary.

### *The* ***typedef***

(See *pixelmap.h* for precise declaration and ordering)

#### Behaviour

| | | |
|---|---|---|
| **br_uint_8** | **type** | ***Individual pixel type*** |
| **br_uint_8** | **flags** | |

#### Size & Position

| | | |
|---|---|---|
| **br_int_16** | **origin_x, origin_y** | ***Local origin (centre), relative to base_x, base_y*** |
| **br_uint_16** | **width, height** | ***Size of pixel map (in pixels)*** |

#### Data

| | | |
|---|---|---|
| **void *** | **pixels** | ***A pointer to raw pixel data*** |
| **br_int_16** | **row_bytes** | ***The difference in bytes between pixels at the same column of adjacent rows*** |
| **br_uint_16** | **base_x, base_y** | ***Co-ordinates of the region of pixels in use*** |
| **br_pixelmap *** | **map** | ***A pointer to a colour map, used when pixel colours are indexed*** |

#### Supplementary

| | | |
|---|---|---|
| **char *** | **identifier** | ***An optional name for texture maps and tables*** |
| **void *** | **user** | ***User data (application dependent)*** |
| **void *** | **device** | ***A device pointer, used if the pixel map originated from a device*** |

### *Related Functions*

#### Scene Rendering

See **BrZbSceneRenderBegin()** 34, **BrZbSceneRender()** 34.

#### Image Support

See **BrPixelmapDoubleBuffer()** 44, **BrPixelmapDirtyRectangleCopy()** 42, **BrPixelmapDirtyRectangleFill()** 42, **BrScenePick2D()** 86.

### *Related Structures*

Scene Modelling

See **br_material**[154].

## Members

### *Behaviour*

### br_uint_8 type

This member defines the type of data stored for each pixel in the pixel map. The various types have values defined by the following symbols:

| Pixel Map Type | Pixel Map Behaviour |
|---|---|
| BR_PMT_INDEX_1 | 1 bit index into a colour map (2 colours) |
| BR_PMT_INDEX_2 | 2 bit index into a colour map (4 colours) |
| BR_PMT_INDEX_4 | 4 bit index into a colour map (16 colours) |
| BR_PMT_INDEX_8 | 8 bit index into a colour map (256 colours) |
| BR_PMT_RGB_555 | 16 bit 'true colour' RGB, 5 bits each colour |
| BR_PMT_RGB_565 | 16 bit 'true colour' RGB, 5 bits red and blue, 6 bits green |
| BR_PMT_RGB_888 | 24 bit 'true colour' RGB, 8 bits each colour |
| BR_PMT_RGBX_888 | 32 bit 'true colour' RGB, 8 bits each colour, 8 bits unused |
| BR_PMT_RGBA_8888 | 32 bit 'true colour' RGB, 8 bits each colour with an 8 bit alpha channel |
| BR_PMT_DEPTH_16 | The pixel map is used as a depth buffer with 16 bit precision |
| BR_PMT_DEPTH_32 | The pixel map is used as a depth buffer with 32 bit precision |

| Pixel Map Type | 32 Bit Pixel Value Encoding | First Four Bytes* of Left Hand Pixel |
|---|---|---|
| BR_PMT_INDEX_1 | 000000000000000000000000000000i | i....... ........ ........ ........ |
| BR_PMT_INDEX_2 | 000000000000000000000000000000ii | ii...... ........ ........ ........ |
| BR_PMT_INDEX_4 | 00000000000000000000000000000iiii | iiii.... ........ ........ ........ |
| BR_PMT_INDEX_8 | 0000000000000000000000000iiiiiiii | iiiiiiii ........ ........ ........ |
| BR_PMT_RGB_555 | 0000000000000000rrrrrgggggbbbbb | gggbbbbb 0rrrrrgg ........ ........ |
| BR_PMT_RGB_565 | 0000000000000000rrrrrgggggbbbbb | gggbbbbb rrrrrggg ........ ........ |
| BR_PMT_RGB_888 | 00000000rrrrrrrrgggggggbbbbbbbb | bbbbbbbb gggggggg rrrrrrrr ........ |
| BR_PMT_RGBX_888 | 00000000rrrrrrrrgggggggbbbbbbbb | bbbbbbbb gggggggg rrrrrrrr xxxxxxxx |
| BR_PMT_RGBA_8888 | aaaaaaaarrrrrrrrgggggggbbbbbbbb | bbbbbbbb gggggggg rrrrrrrr aaaaaaaa |
| BR_PMT_DEPTH_16 | dddddddddddddddd0000000000000000 | Undefined |
| BR_PMT_DEPTH_32 | dddddddddddddddddddddddddddddddd | Undefined |

* The left hand byte is the byte at pixels.

All values are written with the most significant bit to the left.

The Encoding column represents the 32 bit value to be supplied as `colour` to functions such as **`BrPixelmapPixelSet()`**282. The last column shows how the first pixel on a row will appear in the first four bytes indexed from `pixels`. The dots represent further pixels. The ordering of bytes pixel maps is independent of word byte order, except in the case of depth buffers, in which pixels are read and written as 32 bit values. This means that in a 16 bit depth buffer the least significant 16 bits are lost.

Note, with respect to pixel maps used as textures, that zero pixels (irrespective of any palette information) are not rendered, and so have the effect of transparency. This only applies to textures and not to pixel map operations such as **`BrPixelmapCopy()`**285.

## br_uint_8 flags

This is a read-only member, set upon allocation, that contains various flag values. One of the flags that may be useful is `BR_BMF_NO_ACCESS` which will be set if the pixel data is stored at `pixels`. If not set then `pixels` is invalid and there is no direct access to pixel data.

### *Size & Position*

## br_int_16 origin_x, origin_y

These members define the position of the co-ordinate origin of the pixel map relative to the base origin (given by `base_x`, `base_y`). Thus a point plotted at (0,0) will be plotted at column `origin_x` from `base_x`. The co-ordinate origin also effectively defines the centre of projection when used as a rendering destination.

## br_uint_16 width, height

These members contain the dimensions of the visible region of the pixel map.

### *Data*

## void * pixels

When pixel data is directly accessible (see `flags`), this member points to an area of memory containing the raw pixel data. It either points to the start of the memory occupied by the pixel map or the last `row_bytes` of it. However, it always points to the byte of the left hand pixel of the 'first' row. For instance, in monochrome pixel maps it will point to the byte whose most significant bit represents the left hand pixel of the first row. In true colour pixel maps it will point to the least significant byte of the colour of the left hand pixel of the first row, which will be the blue component in `BR_PMT_RGB_888` pixel maps and the alpha component in `BR_PMT_RGBA_8888` pixel maps.

### br_int_16 row_bytes

This member defines the physical row length of the pixel map in terms of the byte difference between pixels in the same column of adjacent rows. It will be negative if the pixel map memory is inverted.

### br_uint_16 base_x, base_y

These members define the top left of the start of pixel map data in terms of `base_y` as a number of `row_bytes`, and `base_x` as a smaller offset from this.

### br_pixelmap * map

For indexed pixel maps (of type `BR_PMT_INDEX_?`), this member points to a colour map. This is used to obtain the 'true colour' corresponding to a particular index.

*Supplementary*

### char * identifier

Pointer to unique, zero terminated, character string (or `NULL` if not required).

If the pixel map is loaded or imported, the `identifier` will have been set using **`BrResStrDup()`**[49].

### void * user

This member may be used by the application for its own purposes. It is initialised to `NULL` upon allocation (if allocated by BRender), and not accessed by BRender thereafter.

### void * device

Some platform specific functions can return a pixel map structure corresponding to an area of video memory or other hardware. This member enables pixel map functions to determine whether the pixel map is in memory or not. It should not be used by the application.

## Operations

---

# BrPixelmapFill()

*Description:* Fill a pixel map with a given value.

*Declaration:* **void BrPixelmapFill(br_pixelmap* dat, br_uint_32 colour)**

*Arguments:* **br_pixelmap * dat**

A pointer to the pixel map to be filled.

**br_uint_32 colour**

Value to set each pixel to.

---

# BrPixelmapRectangleFill()

*Description:* Fill a rectangular window in a pixel map with a given value.

*Declaration:* **void BrPixelmapRectangleFill(br_pixelmap* dst, br_int_16 x, br_int_16 y, br_uint_16 w, br_uint_16 h, br_uint_32 colour)**

*Arguments:* **br_pixelmap * dst**

A pointer to the destination pixel map.

**br_int_16 x,y**

Co-ordinates of the rectangle's top left corner.

**br_uint_16 w,h**

Rectangle width and height (in pixels).

**br_uint_32 colour**

Value to set each pixel to.

*Example:*

```
br_int_16 x,y;
br_uint_16 w,h;
br_pixelmap *offscreen;
...
BrPixelmapRectangleFill(offscreen,x,y,w,h,0);
```

---

# BrPixelmapLine()

*Description:* Draw a line in a pixel map between (x1,y1) and (x2,y2), clipping it to the edges of the pixel map if necessary.

*Declaration:* **void BrPixelmapLine(br_pixelmap* dst, br_int_16 x1, br_int_16 y1, br_int_16 x2, br_int_16 y2, br_uint_32 colour)**

*Arguments:* **br_pixelmap * dst**

A pointer to the destination pixel map.

**br_int_16 x1,y1,x2,y2**

Co-ordinates of the line's endpoints.

**br_uint_32 colour**

Value to set each pixel in the line to.

---

## BrPixelmapPixelSet()

*Description:* Set a pixel to a given value.

*Declaration:* **void BrPixelmapPixelSet(br_pixelmap* dst, br_int_16 x, br_int_16 y, br_uint_32 colour)**

*Arguments:* **br_pixelmap * dst**

A pointer to the destination pixel map.

**br_int_16 x,y**

Pixel co-ordinates.

**br_uint_32 colour**

Pixel value.

---

## BrPixelmapPixelGet()

*Description:* Get the value of a particular pixel

*Declaration:* **br_uint_32 BrPixelmapPixelGet(const br_pixelmap* src, br_int_16 x, br_int_16 y)**

*Arguments:* **const br_pixelmap * src**

A pointer to the source pixel map from which to read the pixel.

**br_int_16 x,y**

Pixel co-ordinates.

*Result:* **br_uint_32 colour**

Pixel value. If the point is off-screen, zero will be returned.

*Remarks:* Some device oriented pixel maps may not support read operations.

## BrPixelmapText()

*Description:* Write a string into a pixel map in a given font.

*Declaration:* **void BrPixelmapText(br_pixelmap* dst, br_int_16 x, br_int_16 y, br_uint_32 colour, const br_font* font, const char* text)**

*Arguments:* **br_pixelmap * dst**

A pointer to the destination pixel map.

**br_int_16 x,y**

Co-ordinates of text's top left corner.

**br_uint_32 colour**

Value to set each text pixel to.

**const br_font * font**

A pointer to a BRender font, or `NULL` for the default font.

The following pointers are available:

`BrFontFixed3x5`

`BrFontProp4x6`

`BrFontProp7x9`

See *`brfont.h`* for precise declaration.

**const char * text**

A string.

*Example:*

```
br_uint_32 colour;
br_pixelmap *pmap;
...
BrPixelmapText(pmap,0,0,colour,BrFontProp7x9,"Example text...");
```

## BrPixelmapTextF()

*Description:* Write a '`printf`' formatted string into a pixel map in a given font. The function will accept format strings and arguments just as for the standard `printf()` function.

*Declaration:* **void BrPixelmapTextF(br_pixelmap* dst, br_int_16 x, br_int_16 y, br_uint_32 colour, const br_font* font, const char* fmt, ...)**

*Arguments:* **br_pixelmap * dst**

A pointer to the destination pixel map.

**br_int_16 x, y**

Co-ordinates of text's top left corner.

**br_uint_32 colour**

Value to set each text pixel to.

**const br_font * font**

A pointer to a BRender font, or `NULL` for the default font.

The following pointers are available:

`BrFontFixed3x5`

`BrFontProp4x6`

`BrFontProp7x9`

See *`brfont.h`* for precise declaration.

**const char * fmt**

A format string (as for `printf()`).

*Example:*

```
br_uint_32 colour;
br_pixelmap *pmap;
...
BrPixelmapTextF
(  pmap,0,0,255,NULL,"Frames/Sec = %16g Polys/Sec = %16g"
,  TIMING_FRAMES/((end_time-start_time)/(double)CLOCK_RATE)
,  total_faces/((end_time-start_time)/(double)CLOCK_RATE)
);
```

## BrPixelmapTextWidth()

*Description:* Find the width of a string for a given font and pixel map.

*Declaration:* **br_uint_16 BrPixelmapTextWidth(const br_pixelmap* dst, const br_font* font, const char* text)**

*Arguments:* **const br_pixelmap * dst**

A pointer to a pixel map.

**const br_font * font**

A pointer to a BRender font, or `NULL` for the default font.

**const char * text**

A string.

*Result:* **br_uint_16**

Returns the string width in pixels. If the `text` argument is `NULL`, the width of one character is returned.

*See Also:* **BrPixelmapText()** 283

## BrPixelmapTextHeight()

*Description:* Find the height of a font for a given pixel map.

*Declaration:* **br_uint_16 BrPixelmapTextHeight(const br_pixelmap* dst, const br_font* font)**

*Arguments:* **const br_pixelmap * dst**

A pointer to a pixel map.

**const br_font * font**

A pointer to a BRender font, or `NULL` for the default font.

*Result:* Returns the font height in pixels.

*See Also:* **BrPixelmapText()** 283

# Copy/Assign

The **br_pixelmap** 277 structure should not be copied by structure assignment. Copies should only be made via construction. However, the pixels may be copied, in whole or in part using the following functions.

## BrPixelmapCopy()

*Description:* Copy the data in one pixel map to another. The source and destination pixel maps must have the same type and dimensions.

*Declaration:* **void BrPixelmapCopy(br_pixelmap* dst, const br_pixelmap* src)**

*Arguments:* **br_pixelmap * dst**

A pointer to the destination pixel map.

**const br_pixelmap * src**

A pointer to the source pixel map.

*Example:*

```
br_pixelmap *offscreen, *backdrop;
...
BrPixelmapCopy(offscreen, backdrop);
```

## BrPixelmapRectangleCopy()

*Description:* Copy a rectangular window from one pixel map to another.

*Declaration:*
```
void BrPixelmapRectangleCopy (br_pixelmap* dst,
br_int_16 dx, br_int_16 dy, const br_pixelmap* src,
br_int_16 sx, br_int_16 sy, br_uint_16 w, br_uint_16 h)
```

*Arguments:*

**br_pixelmap * dst**

A pointer to the destination pixel map.

**br_int_16 dx,dy**

Co-ordinates of the destination rectangle's top left corner.

**const br_pixelmap * src**

A pointer to the source pixel map.

**br_int_16 sx,sy**

Co-ordinates of the source rectangle's top left corner.

**br_uint_16 w,h**

Rectangle width and height (in pixels).

## Access & Maintenance

Colour or texture maps and shade tables must be added to the registry if they are involved in rendering a scene. They should not be modified during rendering.

Colour maps and shade tables that have been added to the registry may be accessed by BRender during rendering.

If any changes are made to colour maps or shade tables involved in rendering, they must be updated before the next rendering in which they are involved.

Note that while texture maps and shade tables are pixel maps, the reverse does not apply. A pixel map is only a texture map by virtue of being added to the registry as such. Similarly with a shade table. Most importantly, a pixel map cannot be both! A texture map must be removed from the registry for it to become just a pixel map, and only then could it be added as a shade table. The same applies with roles interchanged. Note that this just applies to the instance of the **br_pixelmap**$_{277}$ data structure, not to the memory that holds the pixel data. Therefore it is quite valid to have a texture map and shade table sharing the same pixel memory (not that it is easy to envisage a useful effect of doing so).

## BrMapAdd()

*Description:* Add a texture map to the registry, updating it as necessary. All texture maps must be added to the registry before they are subsequently involved in rendering.

*Declaration:* **br_pixelmap* BrMapAdd(br_pixelmap* pixelmap)**

*Arguments:* **br_pixelmap * pixelmap**

A pointer to a texture map.

*Result:* **br_pixelmap ***

Returns a pointer to the added texture map, else NULL if unsuccessful.

*See Also:* **BrMapUpdate()** 287, **BrMapAddMany()** 287, **BrPixelmapLoad()** 301, **BrMapFind()** 295, **BrMapRemove()** 288.

---

# BrMapAddMany()

*Description:* Add a number of texture maps to the registry, updating them as necessary.

*Declaration:* **br_uint_32 BrMapAddMany(br_pixelmap* const* pixelmaps, int n)**

*Arguments:* **br_pixelmap * const * pixelmaps**

A pointer to an array of pointers to texture maps.

**int n**

Number of texture maps to add to the registry.

*Result:* **br_uint_32**

Returns the number of texture maps added successfully.

*See Also:* **BrMapUpdate()** 287, **BrMapAdd()** 287, **BrMapRemove()** 288, **BrMapRemoveMany()** 288

---

# BrMapUpdate()

*Description:* Update a texture map.

*Declaration:* **void BrMapUpdate(br_pixelmap* pixelmap, br_uint_16 flags)**

*Arguments:* **br_pixelmap * pixelmap**

A pointer to a texture map.

**br_uint_16 flags**

Texture map update flags. In general, BR_MAPU_ALL should be used.

*See Also:* **BrMapAdd()** 287.

---

# BrMapRemove()

*Description:* Remove a texture map from the registry.

*Declaration:* **br_pixelmap* BrMapRemove(br_pixelmap* pixelmap)**

*Arguments:* **br_pixelmap * pixelmap**

A pointer to a texture map.

*Result:* **br_pixelmap ***

Returns a pointer to the item removed.

*See Also:* **BrMapAdd()** 287

## BrMapRemoveMany()

*Description:* Remove a number of texture maps from the registry.

*Declaration:* **br_uint_32 BrMapRemoveMany(br_pixelmap* const* pixelmaps, int n)**

*Arguments:* **br_pixelmap * const * pixelmaps**

A pointer to an array of pointers to texture maps.

**int n**

Number of texture maps to remove from the registry.

*Result:* **br_uint_32**

Returns the number of texture maps removed successfully.

*See Also:* **BrMapAddMany()** 287

## BrTableAdd()

*Description:* Add a shade table to the registry, updating it as necessary. All shade tables must be added to the registry before they are used subsequently.

*Declaration:* **br_pixelmap* BrTableAdd(br_pixelmap* pixelmap)**

*Arguments:* **br_pixelmap * pixelmap**

A pointer to a shade table.

*Result:* **br_pixelmap ***

Returns a pointer to the added shade table, else NULL if unsuccessful.

*See Also:* **BrTableUpdate()** 289, **BrTableAddMany()** 289, **BrPixelmapLoad()** 301, **BrTableFind()** 298, **BrTableRemove()** 289

## BrTableAddMany()

*Description:* Add a number of shade tables to the registry, updating them as necessary.

*Declaration:* **br_uint_32 BrTableAddMany(br_pixelmap* const* pixelmaps, int n)**

*Arguments:* **br_pixelmap * const * pixelmaps**

A pointer to an array of pointers to shade tables.

**int n**

Number of shade tables to add to the registry.

*Result:* **br_uint_32**

Returns the number of shade tables added successfully.

*See Also:* **BrTableUpdate()** 289, **BrTableAdd()** 288, **BrTableRemove()** 289, **BrTableRemoveMany()** 290

---

## BrTableUpdate()

*Description:* Update a shade table.

*Declaration:* **void BrTableUpdate(br_pixelmap* pixelmap, br_uint_16 flags)**

*Arguments:* **br_pixelmap * pixelmap**

A pointer to a shade table.

**br_uint_16 flags**

Shade table update flags. In general, BR_TABU_ALL should be used.

*See Also:* **BrTableAdd()** 288.

---

## BrTableRemove()

*Description:* Remove a shade table from the registry.

*Declaration:* **br_pixelmap* BrTableRemove(br_pixelmap* pixelmap)**

*Arguments:* **br_pixelmap * pixelmap**

A pointer to a shade table.

*Result:* **br_pixelmap ***

Returns a pointer to the shade table removed.

*See Also:* **BrTableAdd()** 288

---

## BrTableRemoveMany()

*Description:* Remove a number of shade tables from the registry.

*Declaration:* **br_uint_32 BrTableRemoveMany(br_pixelmap* const* pixelmaps, int n)**

*Arguments:* **br_pixelmap * const * pixelmaps**

A pointer to an array of pointers to shade tables.

**int n**

Number of shade tables to remove from the registry.

*Result:* **br_uint_32**

Returns the number of shade tables removed successfully.

*See Also:* **BrTableAddMany()** 289

---

## Referencing & Lifetime

Pixel maps may be freely and multiply referenced. Once added to the registry as a texture map or shade table, there are certain restrictions, but they may still be multiply referenced. Maps and tables must have been added to the registry if they will be involved in rendering. Texture maps and shade tables must be maintained while they are in the registry or being referenced.

## Initialisation

The structure is initialised by the various construction functions. Pixel maps should not be created or initialised by any other means.

## Construction & Destruction

Apart from platform specific functions, pixel maps should only be constructed by the following BRender functions. Destruction should naturally be performed by the corresponding 'free' function, usually **BrPixelmapFree()** 293. Note that texture maps and shade tables should be removed from the registry before destruction.

---

# BrPixelmapAllocate()

*Description:* Allocate a new pixel map.

*Declaration:* **br_pixelmap* BrPixelmapAllocate(br_uint_8 type, br_uint_16 w, br_uint_16 h, void* pixels, int flags)**

*Arguments:* **br_uint_8 type**

Pixel map type.

**br_uint_16 w**

Width in pixels.

**br_uint_16 h**

Height in pixels.

**void * pixels**

A pointer to an existing block of memory. If NULL, the pixel memory is allocated automatically using **BrResAllocate()**[48].

The calculation obtaining the minimum size required for the block of memory is non-obvious. It is row_bytes*h., and row_bytes is not simply w*bits-per-pixel/8. To determine row_bytes, it is probably simplest to call this function with a dummy, non-NULL pointer (address of an automatic, say), and record the value of row_bytes returned in the **br_pixelmap**[277] (free it immediately afterwards). The memory pointed to by pixels is not accessed by this function.

**int flags**

Supply either BR_PMAF_NORMAL or BR_PMAF_INVERTED. If the latter, the function will automatically set the pixels member of the returned pixel map (whether supplied or not) to be at the start of the final row of pixel map memory (row_bytes is negative).

| Flags | Meaning |
|---|---|
| BR_PMAF_NORMAL | Pixel map ordinate x increases, and y decreases as addresses within the memory block ascend. |
| BR_PMAF_INVERTED | Pixel map ordinate x increases, and y increases as addresses within the memory block ascend. |

*Result:* **br_pixelmap ***

Returns a pointer to the new pixel map, or NULL if unsuccessful.

# BrPixelmapAllocateSub()

*Description:* Allocate a pixel map as part of an existing pixel map. The new pixel map is clipped to the existing pixel map.

*Declaration:* **br_pixelmap* BrPixelmapAllocateSub(br_pixelmap* pm, br_uint_16 x, br_uint_16 y, br_uint_16 w, br_uint_16 h)**

*Arguments:* **br_pixelmap * pm**

A pointer to an existing pixel map.

**br_uint_16 x,y**

Co-ordinates of the top left of the new pixel map in the existing pixel map.

**br_uint_16 w,h**

Width and height of the new pixel map.

*Result:* **br_pixelmap ***

Returns a pointer to the new pixel map, or NULL if unsuccessful.

*Remarks:* The effective origin is set that of the existing pixel map.

## BrPixelmapMatch()

*Description:* Given a pixel map, allocate either a depth buffer or an off-screen colour buffer with the same dimensions.

*Declaration:* **br_pixelmap* BrPixelmapMatch(const br_pixelmap* src, int match_type)**

*Arguments:* **const br_pixelmap * src**

A pointer to the source pixel map.

**int match_type**

The type of matching pixel map required.

| Match Type | Match Method |
|---|---|
| BR_PMMATCH_OFFSCREEN | Create a pixel map of the same type and dimensions. |
| BR_PMMATCH_DEPTH_16 | Create an appropriate 16 bit depth buffer. |

*Result:* **br_pixelmap ***

Returns a pointer to the matching pixel map.

## BrPixelmapClone()

*Description:* Create a pixel map of the same type and dimensions and copy the pixel data.

*Declaration:* **br_pixelmap* BrPixelmapClone(const br_pixelmap* src)**

*Arguments:* **const br_pixelmap * src**

A pointer to the source pixel map.

*Result:* **br_pixelmap ***

Returns a pointer to the new pixel map.

*Example:*

```
br_pixelmap *image, *working_copy;
...
image = BrPixelmapLoad("backdrop.pix");
working_copy = BrPixelmapClone(image);
```

## BrPixelmapFree()

*Description:* Deallocate a pixel map and any associated memory.

*Declaration:* **void BrPixelmapFree(br_pixelmap* pmap)**

*Arguments:* **br_pixelmap * pmap**

A pointer to a pixel map.

---

## Supplementary

---

# BrPixelmapPixelSize()

*Description:* Find the pixel size for a given pixel map.

*Declaration:* **br_uint_16 BrPixelmapPixelSize(const br_pixelmap* pm)**

*Arguments:* **const br_pixelmap * pm**

A pointer to a pixel map.

*Result:* **br_uint_16**

Returns the size of each pixel, in bits.

---

# BrPixelmapChannels()

*Description:* Find the channels available for a given pixel map.

*Declaration:* **br_uint_16 BrPixelmapChannels(const br_pixelmap* pm)**

*Arguments:* **const br_pixelmap * pm**

A pointer to a pixel map.

*Result:* **br_uint_16**

Returns a mask giving the available channels, being a combination of the following bit value symbols:

BR_PMCHAN_INDEX

BR_PMCHAN_RGB

BR_PMCHAN_DEPTH

BR_PMCHAN_ALPHA

BR_PMCHAN_YUV

---

# BrMapCount()

*Description:* Count the number of registered texture maps whose names match a given search pattern. The search pattern can include the standard wild cards '*' and '?'.

*Declaration:* **br_uint_32 BrMapCount(const char* pattern)**

*Arguments:* **const char * pattern**

Search pattern.

*Result:* **br_uint_32**

Returns the number of texture maps matching the search pattern.

*See Also:* **BrMapEnum()** 294, **BrMapFind()** 295

---

# BrMapEnum()

*Description:* Calls a call-back function for every texture map matching a given search pattern. The call-back is passed a pointer to each matching item, and its second argument is an optional pointer supplied by the user. The search pattern can include the standard wild cards '*' and '?'. The call-back itself returns a **br_uint_32**$_{358}$ value. The enumeration will halt at any stage if the return value is non-zero.

*Declaration:* **br_uint_32 BrMapEnum(const char* pattern, br_map_enum_cbfn* callback, void* arg)**

*Arguments:* **const char * pattern**

Search pattern.

**br_map_enum_cbfn * callback**

A pointer to a call-back function.

**void * arg**

An optional argument to pass to the call-back function.

*Result:* **br_uint_32**

Returns the first non-zero call-back return value, or zero if all matching texture maps are enumerated.

*Example:*

```
br_uint_32 BR_CALLBACK test_callback(br_pixelmap* map, void* arg)
{  br_uint_32 count;
...
   return(count);
}
...
{  br_uint_32 enum;
...
   enum = BrMapEnum("map",&test_callback,NULL);
}
```

## BrMapFind()

*Description:* Find a texture map in the registry by name. A call-back function can be setup to be called if the search is unsuccessful. The search pattern can include the standard wild cards '*' and '?'.

*Declaration:* **br_pixelmap* BrMapFind(const char* pattern)**

*Arguments:* **const char * pattern**

Search pattern.

*Result:* **br_pixelmap ***

Returns a pointer to the texture map if found, otherwise NULL. If a call-back exists and is called, the call-back's return value is returned.

*See Also:* **BrMapFindHook()** 295, **BrMapFindMany()** 295

## BrMapFindMany()

*Description:* Find a number of texture maps in the registry by name. The search pattern can include the standard wild cards '*' and '?'.

*Declaration:* **br_uint_32 BrMapFindMany(const char* pattern, br_pixelmap** pixelmaps, int max)**

*Arguments:* **const char * pattern**

Search pattern.

**br_pixelmap ** pixelmaps**

A pointer to an array of pointers to texture maps.

**int max**

Maximum number of texture maps to find.

*Result:* **br_uint_32**

Returns the number of texture maps found. The pointer array is filled with pointers to the found texture maps.

*See Also:* **BrMapFind()** 295, **BrMapFindHook()** 295

## BrMapFindHook()

*Description:* Functions to set up a call-back.

*Declaration:* **br_map_find_cbfn* BrMapFindHook(br_map_find_cbfn* hook)**

*Arguments:* **br_map_find_cbfn * hook**

A pointer to a call-back function.

*Effects:* If **BrMapFind()** 295 is unsuccessful and a call-back has been set up, the call-back is passed the search pattern as its only argument. The call-back should then return a pointer to a substitute or default texture map.

For example, a call-back could be set up to return a default texture map if the desired texture map cannot be found in the registry.

The function **BrMapFindFailedLoad()** 296 is provided and will probably be sufficient in many cases.

*Result:* **br_map_find_cbfn ***

Returns a pointer to the old call-back function.

*Example:*

```
br_map BR_CALLBACK * test_callback(char* pattern)
{  br_map* default_map;
...
   return(default_map);
}
...
{  br_map* map;
...
   BrMapFindHook(&test_callback);
   map = BrMapFind("non_existent_map");
}
```

*See Also:* **BrMapFindFailedLoad()** 296

---

# BrMapFindFailedLoad()

*Description:* This function is provided as a suitable function to supply to **BrMapFindHook()** 295.

*Declaration:* **br_pixelmap* BrMapFindFailedLoad(const char* name)**

*Arguments:* **const char * name**

The name supplied to **BrMapFind()** 295.

*Effects:* Attempts to load the texture map from the filing system using `name` as the filename. Searches in current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined). If successful, sets this name as the identifier of the loaded texture map and adds the texture map to the registry.

*Result:* **br_pixelmap ***

Returns a pointer to the texture map, if found, else `NULL`.

*Example:*

```
BrMapFindHook(BrMapFindFailedLoad);
```

## BrTableCount()

*Description:* Count the number of registered shade tables whose names match a given search pattern. The search pattern can include the standard wild cards '*' and '?'.

*Declaration:* **br_uint_32 BrTableCount(const char* pattern)**

*Arguments:* **const char * pattern**

Search pattern.

*Result:* **br_uint_32**

Returns the number of shade tables matching the search pattern.

*See Also:* **BrTableEnum()** 297, **BrTableFind()** 298

## BrTableEnum()

*Description:* Calls a call-back function for every shade table matching a given search pattern. The call-back is passed a pointer to each matching shade table, and its second argument is an optional pointer supplied by the user. The search pattern can include the standard wild cards '*' and '?'. The call-back itself returns a **br_uint_32** *358* value. The enumeration will halt at any stage if the return value is non-zero.

*Declaration:* **br_uint_32 BrTableEnum(const char* pattern, br_table_enum_cbfn* callback, void* arg)**

*Arguments:* **const char * pattern**

Search pattern.

**br_table_enum_cbfn * callback**

A pointer to a call-back function.

**void * arg**

An optional argument to pass to the call-back function.

*Result:* **br_uint_32**

Returns the first non-zero call-back return value, or zero if all matching shade tables are enumerated.

*Example:*

```
br_uint_32 BR_CALLBACK test_callback(br_pixelmap* table, void* arg)
{  br_uint_32 count;
...
   return(count);
}
...
{  br_uint_32 enum;
...
   enum = BrTableEnum("Table",&test_callback,NULL);
}
```

## BrTableFind()

*Description:* Find a shade table in the registry by name. A call-back function can be setup to be called if the search is unsuccessful. The search pattern can include the standard wild cards '*' and '?'.

*Declaration:* **br_pixelmap* BrTableFind(const char* pattern)**

*Arguments:* **const char * pattern**

Search pattern.

*Result:* **br_pixelmap ***

Returns a pointer to the shade table if found, otherwise NULL. If a call-back exists and is called, the call-back's return value is returned.

*See Also:* **BrTableFindHook()** 299, **BrTableFindMany()** 298

## BrTableFindMany()

*Description:* Find a number of shade tables in the registry by name. The search pattern can include the standard wild cards '*' and '?'.

*Declaration:* **br_uint_32 BrTableFindMany(const char* pattern, br_pixelmap** pixelmaps, int max)**

*Arguments:* **const char * pattern**

Search pattern.

**br_pixelmap ** pixelmaps**

A pointer to an array of pointers to shade tables.

**int max**

Maximum number of shade tables to find.

*Result:* **br_uint_32**

Returns the number of shade tables found. The pointer array is filled with pointers to the found shade tables.

*See Also:* **BrTableFind()** 298, **BrTableFindHook()** 299

## BrTableFindHook()

*Description:* Functions to set up a call-back. If **BrTableFind()** 298 is unsuccessful and a call-back has been set up, the call-back it is passed the search pattern as its only argument. The call-back should then return a pointer to a substitute or default shade table.

For example, a call-back could be set up to return a default shade table if the desired shade table cannot be found in the registry.

The function **BrTableFindFailedLoad()** 299 is provided and will probably be sufficient in many cases.

*Declaration:*
```
br_table_find_cbfn*
BrTableFindHook(br_table_find_cbfn* hook)
```

*Arguments:* **br_table_find_cbfn * hook**

A pointer to a call-back function.

*Result:* **br_table_find_cbfn ***

Returns a pointer to the old call-back function.

*Example:*
```
br_table BR_CALLBACK * test_callback(char* pattern)
{  br_table* default_table;
...
   return(default_table);
}
...
{  br_table* table;
...
   BrTableFindHook(&test_callback);
   table = BrTableFind("non_existent_table");
}
```

*See Also:* **BrTableFindFailedLoad()** 299

## BrTableFindFailedLoad()

*Description:* This function is provided as a suitable function to supply to **BrTableFindHook()** 299.

*Declaration:*
```
br_pixelmap* BrTableFindFailedLoad(const char* name)
```

*Arguments:* **const char * name**

The name supplied to **BrTableFind()** 298.

*Effects:* Attempts to load the shade table from the filing system using `name` as the filename. Searches in current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined). If successful, sets this name as the identifier of the loaded shade table and adds the shade table to the registry.

*Result:* **`br_pixelmap *`**

Returns a pointer to the shade table, if found, else `NULL`.

*Example:*

```
BrTableFindHook(BrTableFindFailedLoad);
```

---

Import & Export

## BrPixelmapFileCount()

*Description:* Locate a given file and count the number of pixel maps in it.

*Declaration:* **br_uint_32 BrPixelmapFileCount(const char* filename, br_uint_16* num)**

*Arguments:* **const char * filename**

Name of the file containing the pixel maps to count.

**br_uint_16 * num**

Pointer to the variable in which to store the number of pixel maps counted in the file. If NULL, the file will still be located and appropriate success returned, but no count will be made.

*Effects:* Searches for filename, if no path specified with file looks in current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined). If a file is found, will count the number of pixel maps stored in it.

*Result:* **br_uint_32**

Returns zero if the file was found (even if it is not a pixel map file), non-zero otherwise.

## BrPixelmapLoad()

*Description:* Load a pixel map. Note that they are not added to the registry.

*Declaration:* **br_pixelmap* BrPixelmapLoad(const char* filename)**

*Arguments:* **const char * filename**

Name of the file containing the pixel map to load.

*Effects:* Searches for filename, if no path specified with file looks in current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined).

*Result:* **br_pixelmap ***

Returns a pointer to the loaded pixel map, or NULL if unsuccessful.

*See Also:* **BrPixelmapLoadMany()** 301, **BrPixelmapSave()** 304, **BrMapAdd()** 287, **BrTableAdd()** 288.

## BrPixelmapLoadMany()

*Description:* Load a number of pixel maps. Note that they are not added to the registry.

*Declaration:* **br_uint_32 BrPixelmapLoadMany(const char* filename, br_pixelmap** pixelmaps, br_uint_16 num)**

*Arguments:* **const char * filename**

Name of the file containing the pixel maps to load.

**br_pixelmap ** pixelmaps**

A non-NULL pointer to an array of pointers to pixel maps.

**br_uint_16 num**

Maximum number of pixel maps to load.

*Effects:* Searches for filename, if no path specified with file looks in current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined).

*Result:* **br_uint_32**

Returns the number of pixel maps loaded successfully. The pointer array is filled with pointers to the loaded pixel maps.

*See Also:* See **BrPixelmapFileCount()** 247 to determine the number of pixel maps in a file.

---

# BrFmtBMPLoad()

*Description:* Load a pixel map in the BMP format.

*Declaration:* **br_pixelmap* BrFmtBMPLoad(const char* name, br_uint_32 flags)**

*Arguments:* **const char * name**

Name of the file containing the pixel map.

**br_uint_32 flags**

Either BR_PMT_RGBX_888 or BR_PMT_RGBA_8888, when the source pixel map uses 32 bits per pixel. Zero otherwise.

*Effects:* Searches for filename, if no path specified with file looks in current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined).

*Result:* **br_pixelmap ***

Returns a pointer to the loaded pixel map.

---

# BrFmtGIFLoad()

*Description:* Load a pixel map in the GIF format.

*Declaration:* **br_pixelmap* BrFmtGIFLoad(const char* name, br_uint_32 flags)**

*Arguments:* **const char * name**

Name of the file containing the pixel map.

**br_uint_32 flags**

Either BR_PMT_RGBX_888 or BR_PMT_RGBA_8888, when the source pixel map uses 32 bits per pixel. Zero otherwise.

*Effects:* Searches for filename, if no path specified with file looks in current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined).

*Result:* **br_pixelmap ***

Returns a pointer to the loaded pixel map.

---

## BrFmtIFFLoad()

*Description:* Load a pixel map in the IFF format.

*Declaration:* **br_pixelmap* BrFmtIFFLoad(const char* name, br_uint_32 flags)**

*Arguments:* **const char * name**

Name of the file containing the pixel map.

**br_uint_32 flags**

Either BR_PMT_RGBX_888 or BR_PMT_RGBA_8888, when the source pixel map uses 32 bits per pixel. Zero otherwise.

*Effects:* Searches for filename, if no path specified with file looks in current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined).

*Result:* **br_pixelmap ***

Returns a pointer to the loaded pixel map.

---

## BrFmtTGALoad()

*Description:* Load a pixel map in the TGA format.

*Declaration:* **br_pixelmap* BrFmtTGALoad(const char* name, br_uint_32 flags)**

*Arguments:* **const char * name**

Name of the file containing the pixel map.

**br_uint_32 flags**

Either BR_PMT_RGBX_888 or BR_PMT_RGBA_8888, when the source pixel map uses 32 bits per pixel. Zero otherwise.

*Effects:* Searches for `filename`, if no path specified with file looks in current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined).

*Result:* **`br_pixelmap *`**

Returns a pointer to the loaded pixel map.

## BrPixelmapSave()

*Description:* Save a pixel map to a file.

*Declaration:*
```
br_uint_32 BrPixelmapSave(const char* filename,
const br_pixelmap* pixelmap)
```

*Arguments:* **`const char * filename`**

Name of the file to save the pixel map to.

**`const br_pixelmap * pixelmap`**

A pointer to a pixel map.

*Effects:* Writes the pixel map to a file*.

*Result:* **`br_uint_32`**

Returns `NULL` if the pixel map could not be saved.

## BrPixelmapSaveMany()

*Description:* Save a number of pixel maps to a file.

*Declaration:*
```
br_uint_32 BrPixelmapSaveMany(const char* filename,
const br_pixelmap* const* pixelmaps, br_uint_16 num)
```

*Arguments:* **`const char * filename`**

Name of the file to save the pixel maps to.

**`const br_pixelmap * const * pixelmaps`**

A pointer to an array of pointers to pixel maps. If `NULL`, all registered texture maps and shade tables are saved (irrespective of `num`).

**`br_uint_16 num`**

Number of pixel maps to save.

*Effects:* Writes the pixel maps to a file†.

*Result:* **`br_uint_32`**

Returns the number of pixel maps saved successfully.

---

* Any existing file of the same name is overwritten.

† Any existing file of the same name is overwritten.

# br_pool

## The Structure

This structure holds details of a memory block pool (an optimised memory allocation scheme on a per data structure basis). This is ideal for cases where a particular data structure has intensive dynamic storage, e.g. matrices.

Note that currently, there is no automatic contraction of Pools, nor are there functions to achieve this (with the current implementation it would have a relatively high processing overhead). Therefore, it may be wise to destroy Pools when possible, if they have irregular or sporadic use, or memory is at a premium.

### *The **typedef***

(See *pool.h* for precise declaration and ordering)

Primary

| | | |
|---|---|---|
| **br_pool_block *** | **free** | ***Linked list of unused blocks*** |
| **br_uint_32** | **block_size** | ***Size of each item*** |
| **br_uint_32** | **chunk_size** | ***Number of blocks to increase pool by*** |
| **int** | **mem_type** | ***Memory class of blocks*** |

### *Related Functions*

Memory Management

**BrAllocatorSet()**[103].

## Members

### *Primary*

#### br_pool_block * free

This is a pointer to the next available block of memory (of size `block_size`). When freed, blocks are cast as **br_pool_block**[310] and inserted at the head of the list to keep track of them. An allocation simply removes the item at the head from the list. It may then be initialised as required.

#### br_uint_32 block_size

This is the size of each item, or memory block. Its value must be greater than or equal to `sizeof(br_pool_block)`, if not, memory corruption will ensue. This is typically 4 bytes, but given that pools are intended for larger structures, this is not a significant restriction. See **BrPoolAllocate()**[308].

### br_uint_32 chunk_size

When all the blocks in a pool are allocated (or the pool is empty), the value of this member determines the number of blocks that will be added to the pool to enable further block allocations.

### uint mem_type

This is the memory class from which the blocks should be allocated. Any number of pools may exist for a given memory class.

## Operations

---

## BrPoolBlockAllocate()

*Description:* Allocate a block from a pool. If the pool is full, it will expand as necessary.

*Declaration:* **void* BrPoolBlockAllocate(br_pool* pool)**

*Arguments:* **br_pool * pool**

A pointer to the relevant pool.

*Result:* **void ***

Returns a pointer to the allocated block, or NULL if unsuccessful.

---

## BrPoolBlockFree()

*Description:* Deallocate a block from a pool.

*Declaration:* **void BrPoolBlockFree(br_pool* pool, void* b)**

*Arguments:* **br_pool * pool**

A pointer to the relevant pool.

**void * b**

A pointer to the block to deallocate.

---

## BrPoolEmpty()

*Description:* Mark all blocks in a pool as unused.

*Declaration:* **void BrPoolEmpty(br_pool* pool)**

*Arguments:* **br_pool * pool**

A pointer to the pool to be emptied.

---

## Copy/Assign

Do not copy.

## Access & Maintenance

No members should be modified. The structure will be maintained until the pool is freed using **`BrPoolFree()`**[309]. No additional maintenance required.

## Referencing & Lifetime

Do not reference a copy. Ensure that references to a pool descriptor do not outlast the pool.

## Initialisation

Initialisation is performed by **`BrPoolAllocate()`**[308].

Effectively this sets `free` to `NULL`, `block_size` to the specified value (aligned appropriately), `chunk_size` to the specified value, and `mem_type` to the specified value.

## Construction & Destruction

Memory pool structures should be constructed by **`BrPoolAllocate()`**[308], as this will allocate the pool descriptors from the pool resource class. Some Pool functions depend on this.

---

# `BrPoolAllocate()`

*Description:* Create a new block pool.

*Declaration:* **`br_pool* BrPoolAllocate(int block_size, int chunk_size, br_uint_8 mem_type)`**

*Arguments:* **`int block_size`**

The size of each block, in bytes. Minimum value is 1, though note that this size will be rounded up to the next multiple of a suitable alignment factor, typically 8 (`BR_POOL_ALIGN+1`). Pools are not intended for simple data structures.

**`int chunk_size`**

The number of blocks to allocate each time the pool becomes full. The value should be between 1 and say, 10% of the maximum number of blocks allocated at any one time. Naturally, there is a compromise between saving processing overhead for each memory allocation (only when the pool grows), and saving memory overhead in terms of average unused blocks.

**`br_uint_8 mem_type`**

Memory type. More than one pool may be created for a particular memory class.

*Result:* **br_pool ***

Returns a pointer to the new pool, or NULL if it could not be created.

## BrPoolFree()

*Description:* Deallocate an entire pool, thereby losing any blocks it may contain.

*Declaration:* **void BrPoolFree(br_pool* pool)**

*Arguments:* **br_pool * pool**

A pointer to the pool to be deallocated.

## Supplementary

Given that Pools are registered in the "POOL" resource class, the resource class supplementary functions are available.

# br_pool_block

## The Structure

This structure is simply used as a means of 'spiking' freed pool blocks until they can be reused.

(See *pool.h* for precise declaration and ordering)

### *The* **typedef**

Primary

| | | |
|---|---|---|
| **br_pool_block *** | **next** | ***Pointer to next free block*** |

### *Related Functions*

Memory Management

**BrPoolBlockAllocate()**[307], **BrPoolBlockFree()**[307].

## Members

### *Primary*

#### br_pool_block * next

A pointer to the next free pool block. This is really an area of memory with size as specified in the pool descriptor (**br_pool**[306]).

## Copy/Assign

Don't copy or reference.

## Access & Maintenance

Should not be accessed by the applications programmer.

## Referencing & Lifetime

Should not be referenced by the applications programmer.

## Initialisation

When a pool block is spiked onto the free linked list, it is cast to this structure, the `next` member is set to point to `free`, and `free` is then set to point to `this`.

## Construction & Destruction

The structure is only used for casting purposes.

# br_primitive

## The Structure

The primitive is an item of rendering data produced in the second phase of rendering. It is typically a lit face of a model, transformed into screen co-ordinates. It contains sufficient information (some private) to be depth sorted and rendered to the destination pixel map. The application programmer will only be handling primitives as far as is necessary to determine whether the primitive should appear and in what order it should be rendered. Primitives are only generated by BRender.

### *The* ***typedef***

(See *order.h* for a precise declaration)

#### Properties

| | | |
|---|---|---|
| **br_uint_16** | **type** | ***Primitive type*** |
| **br_material *** | **material** | ***Material from which primitive was generated*** |

#### Organisation

| | | |
|---|---|---|
| **br_primitive *** | **next** | ***Next primitive (when in bucket)*** |

### *Related Functions*

For details of how to specify that a primitive call-back function should be called during rendering see **BrZsPrimitiveCallbackSet()** 39. To determine into which bucket a primitive would be inserted see **BrZsPrimitiveBucketSelect()** 318. To insert a primitive into an order table see **BrZsOrderTablePrimitiveInsert()** 267.

### *Related Structures*

See **br_order_table** *264* for the container into which primitives are inserted. See **br_primitive_cbfn** *315* for details of how to customise primitive insertion into order tables.

## Members

### *Properties*

#### br_uint_16 type

There are currently three types of primitive: point, line and triangle. This member indicates the type of this primitive. It should not be modified. The following table lists the symbols defining the value of each type.

| Primitive Type | Description |
|---|---|
| BR_PRIMITIVE_POINT | The primitive is a point defined by a single vertex |

| | |
|---|---|
| BR_PRIMITIVE_LINE | The primitive is a line defined by two vertices |
| BR_PRIMITIVE_TRIANGLE | The primitive is a triangle defined by three vertices |

### br_material * material

The material assigned to the face of the model from which the primitive was generated (explicitly or by inheritance).

*Organisation*

### br_primitive * next

Each bucket within an order table is a linked list of primitives. The primitive contains the links in the form of this pointer. When a primitive is inserted into a bucket, it is inserted at the head (the first bucket in the list). Upon rendering the buckets are rendered from the head onwards. This member should not be modified.

## Copy/Assign

Do not copy primitives.

## Access & Maintenance

Primitives are only generated by BRender, and are primarily for internal use. The applications programmer is not expected to traverse or maintain primitives. Some members may be usefully read, but modification is not encouraged for members other than material.

## Referencing & Lifetime

Primitives are allocated from the primitive heap as supplied to **BrZsBegin()**[28]. They are valid until the next call of **BrZsSceneRenderBegin()**[36] when the primitive heap will be overwritten with new primitives - unless, of course a different one is set up.

## Construction & Destruction

Primitives can only be constructed through the rendering process. They are never destroyed, but are overwritten.

br_primitive

# br_primitive_cbfn

## The Call-Back Function

This type defines a call-back function, which can be specified by using the function **BrZsPrimitiveCallbackSet()** 39. It is called for each primitive generated from a model actor in the actor hierarchies supplied to **BrZsSceneRender()** 34 and **BrZsSceneRenderAdd()** 37 (it may also be called by **BrZsModelRender()** 254). It enables an application to perform customised insertion of primitives into order tables, or to perform computations that require information only obtainable just after a primitive has been generated by the rendering engine.

### *The **typedef***

(See *zsproto.h* for a precise declaration)

```
void br_primitive_cbfn(br_primitive*, br_actor*, const br_model*, const br_material*,
                                                  br_order_table*, const br_scalar*)
```

***Primitive call-back***

### *Related Functions*

For details of how to specify that a primitive call-back function should be called during rendering see **BrZsPrimitiveCallbackSet()** 39.

Functions dedicated for use within primitive call-backs: **BrZsOrderTablePrimitiveInsert()** 267, **BrZsPrimitiveBucketSelect()** 318, **BrOnScreenCheck()** 254, **BrOriginToScreenXY()** 255, **BrPointToScreenXY()** 255, **BrPointToScreenXYMany()** 256, **BrOriginToScreenXYZO()** 256, **BrPointToScreenXYZO()** 257, **BrPointToScreenXYZOMany()** 258.

### *Related Structures*

See **br_primitive**312 for details of the primitive data structure. See **br_model_custom_cbfn**251 for a substitute custom model rendering call-back. See **br_order_table**264 for details of inserting primitives into order tables.

## Specification

---

# CBFnPrimitive()

*Description:* An application defined call-back function that is called at some point during rendering – at precisely what point is undefined, and whether its children have been processed is also undefined. The pass through equivalent for this call-back is described in the example. The function may call any non-rendering functions available to a model's custom call-back function – see **br_model_custom_cbfn**251.

*Declaration:* **void BR_CALLBACK CBFnPrimitive(br_primitive* primitive, br_actor* actor, const br_model* model, const br_material* material, br_order_table* order_table, const br_scalar* z)**

*Arguments:* **br_primitive * primitive**

Primitive to be inserted in `order_table`.

**br_actor * actor**

Pointer to model actor referencing the model from which the primitive was generated.

**const br_model * model**

Pointer to the model from which the primitive was generated.

**const br_material * material**

Pointer to actor's material if defined, or default material otherwise.

**br_order_table * order_table**

Order table into which the primitive was about to be inserted.

**const br_scalar * z**

Non-`NULL` pointer to one or more depth values of the primitive's vertices ([`-hither_z`, `-yon_z`] in camera co-ordinate space (linearly) mapped to [`-hither_z`,`+yon_z`]). The number of values pointer to can be determined from the `type` member of `primitive`.

*Preconditions:* BRender has completed initialisation. Rendering is in progress (by the Z-Sort renderer). The rendering engine has generated a primitive that faces the viewer.

*Effects:* Behaviour is up to the application. Any of the operations described for **br_model_custom_cbfn**$_{251}$ can be used apart from **Br[Zb|Zs]ModelRender()**$_{253|254}$.

*Remarks:* Any other BRender functions may be called from within this call-back with the following restrictions:

- Don't call any rendering functions, e.g. **BrZsSceneRenderAdd()**$_{37}$.
- Don't modify any light, clip-plane or camera actors.
- Do not access output buffer until rendering has completed.
- Don't change the environment actor.
- For best performance, avoid adding, updating or removing registry items – try to do these things before rendering.
- Do not modify the actor hierarchy.

*Example:* The following code demonstrates a primitive call-back function that performs the equivalent of BRender's normal primitive insertion, and so can be used for collection of statistics, say.

```
void BR_CALLBACK CBFnPrimitive(br_primitive* primitive, br_actor*
   actor, const br_model* model, const br_material* material,
   br_order_table* order_table, const br_scalar* z)
{
```

```
    BrZsOrderTablePrimitiveInsert
    (  order_table
    ,  primitive
    ,  BrZsPrimitiveBucketSelect
       (  z
          primitive->type
          order_table->min_z
          order_table->max_z
          order_table->size
          order_table->type
       )
    )
    ...
    /* Collect statistics here */
    ...
}
```

*See Also:* **br_primitive**[312], **br_model_custom_cbfn**[251], **br_pick2d_cbfn**[272], **br_pick3d_cbfn**[275], **BrZsOrderTablePrimitiveInsert()**[267], **BrZsPrimitiveBucketSelect()**[318].

## Operations

The following functions are provided solely for appropriate use within ***CBFnModelCustom()***[251], ***CBFnRenderBounds()***[316] and ***CBFnPrimitive()***[316] functions (see **br_model_custom_cbfn**[251]).

**BrZsOrderTablePrimitiveInsert()**[267]
BrZsPrimitiveBucketSelect()[318]
**BrOnScreenCheck()**[254]
**BrOriginToScreenXY()**[255]
**BrPointToScreenXY()**[255]
**BrPointToScreenXYMany()**[256]
**BrOriginToScreenXYZO()**[256]
**BrPointToScreenXYZO()**[257]
**BrPointToScreenXYZOMany()**[258]

# BrZsPrimitiveBucketSelect()

*Description:* Determine into which bucket a primitive would be placed given a particular sorting method.

*Declaration:* **br_uint_16 BrZsPrimitiveBucketSelect(const br_scalar* z, br_uint_16 pr_type, br_scalar min_z, br_scalar max_z, br_uint_16 ot_size, br_uint_16 ot_type)**

*Arguments:* **const br_scalar * z**

A non-NULL pointer to the z values corresponding to each vertex. This should point to as many values as are indicated by the primitive type pr_type.

**br_int_16 pr_type**

The primitive type (see **br_primitive**$_{312}$).

**br_scalar min_z, max_z**

Depth range of buckets (use **BrZsScreenZToDepth()**$_{33}$ if wishing to work with screen ordinates).

**br_uint_16 ot_size**

Number of buckets (see **br_order_table**$_{264}$).

**br_uint_16 ot_type**

How the bucket should be determined (see **br_order_table**$_{264}$).

*Result:* **br_uint_16**

The index of the bucket as would be passed to **BrZsOrderTablePrimitiveInsert()**$_{267}$.

*Remarks:* Although the calculation performed by this function is trivial, it is provided to allow applications to ensure that the same bucket selection occurs in primitive call-back functions as would occur otherwise.

# br_quat

## The Structure

The quaternion is an extension of the complex number, with the addition of two further imaginary components. BRender restricts itself to using unit quaternions, which have the useful property of being able to easily represent a 3D transform consisting of a rotation about an arbitrary vector.

The x, y and z components of the quaternion hold the elements of this vector scaled to a length equal to the sine of half the angle of rotation. The w component holds the cosine of half the angle. The magnitude of a unit quaternion is of course 1 ($w^2+x^2+y^2+z^2=1$).

The quaternion is written thus:

$$w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$$

NB Do not confuse the quaternion with 3D vectors or homogenous co-ordinates, i.e. **br_quat**[320] and **br_vector4**[376] have no relationship.

### *The* ***typedef***

(See *quat.h* for precise declaration and ordering)

| | | |
|---|---|---|
| **br_scalar** | **w** | ***Real component*** |
| **br_scalar** | **x** | ***'i' component (vector, x axis component)*** |
| **br_scalar** | **y** | ***'j' component (vector, y axis component)*** |
| **br_scalar** | **z** | ***'k' component (vector, z axis component)*** |

### *Related Functions*

Maths

See **BrEulerToQuat()**[123], **BrMatrix34ToQuat()**[207], **BrMatrix4ToQuat()**[225].

### *Related Structures*

See **br_transform**[349].

## Members

### br_scalar w

Real component of quaternion. Represents the cosine of half the rotation about the (i,j,k) vector component of the quaternion.

### br_scalar x

First imaginary component of quaternion. Represents the x axis component of the vector about which the rotation occurs.

### br_scalar y

Second imaginary component of quaternion. Represents the y axis component of the vector about which the rotation occurs.

### br_scalar z

Third imaginary component of quaternion. Represents the z axis component of the vector about which the rotation occurs.

## Arithmetic

---

# BrQuatMul()

*Description:* Multiply two quaternions.

*Declaration:* **br_quat* BrQuatMul(br_quat* q, const br_quat* l, const br_quat* r)**

*Arguments:* **br_quat * q**

A pointer to the destination quaternion (may be same as either source).

**const br_quat * l**

A pointer to the left hand source quaternion.

**const br_quat * r**

A pointer to the right hand source quaternion.

*Effects:* The resultant quaternion is computed as follows:

$$(w_l + x_l\mathbf{i} + y_l\mathbf{j} + z_l\mathbf{k})(w_r + x_r\mathbf{i} + y_r\mathbf{j} + z_r\mathbf{k}) \equiv$$

$$w_l w_r - x_l x_r - y_l y_r - z_l z_r$$

$$+(w_l x_r + x_l w_r + y_l z_r - z_l y_r)\mathbf{i}$$

$$+(w_l y_r + y_l w_r + z_l x_r - x_l z_r)\mathbf{j}$$

$$+(w_l z_r + z_l w_r + x_l y_r - y_l x_r)\mathbf{k}$$

*Result:* **br_quat ***

The destination quaternion pointer is returned as supplied for convenience.

*Remarks:* Quaternion multiplication has the effect of concatenating the individual transformations that each represents. It is not commutative.

## BrQuatInvert()

*Description:* Obtain the inverse of a unit quaternion. This is effectively the rotation reversed (about the same vector component).

*Declaration:* **br_quat* BrQuatInvert(br_quat* q, const br_quat* qq)**

*Arguments:* **br_quat * q**

A pointer to the destination quaternion (can be same as source).

**const br_quat * qq**

A pointer to the source quaternion.

*Effects:* The resultant quaternion is computed as follows:

$$(w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k})^{-1} \equiv w - x\mathbf{i} - y\mathbf{j} - z\mathbf{k}$$

*Result:* **br_quat ***

The destination quaternion pointer is returned as supplied for convenience.

## BrQuatSlerp()

*Description:* The 'Slerp' operation (spherical, linear interpolation) interpolates linearly between two quaternions along the most direct path between the two orientations.

*Declaration:* **br_quat* BrQuatSlerp(br_quat* q, const br_quat* l, const br_quat* r, br_scalar t, br_int_16 spins)**

*Arguments:* **br_quat * q**

A pointer to the destination quaternion (can be same as either source).

**const br_quat * l**

A pointer to the starting quaternion (equivalent to result with t=0).

**const br_quat * l**

A pointer to the finishing quaternion (equivalent to result with t=1).

**br_scalar t**

Interpolation parameter in the range [0,1].

**br_int_16**

Number of extra spins in the interpolated path.

A special value of zero causes interpolation along the shortest path.

A special value of -1 causes interpolation along the longest path.

*Effects:*

Computes the quaternion between `l` and `r` corresponding to position `t` along the

interpolated path between them.

*Result:* **br_quat ***

The destination quaternion pointer is returned as supplied for convenience.

---

## Conversion

### *From Eulers and Matrices*

See **BrEulerToQuat()** 123, **BrMatrix34ToQuat()** 207, **BrMatrix4ToQuat()** 225.

### *To Eulers and Matrices*

See **BrQuatToEuler()** 323, **BrQuatToMatrix34()** 323, and **BrQuatToMatrix4()** 324 as described below.

Also see **BrTransformToTransform()** 353.

---

# BrQuatToEuler()

*Description:* Convert a unit quaternion to an Euler angle set, that would have the same transformational effect.

*Declaration:* **br_euler* BrQuatToEuler(br_euler* euler, const br_quat* q)**

*Arguments:* **br_euler * euler**

A pointer to the destination Euler angle set to receive the conversion.

**const br_quat * q**

A pointer to the source unit quaternion.

*Result:* **br_euler ***

Returns `euler` for convenience.

---

# BrQuatToMatrix34()

*Description:* Convert a unit quaternion to a 3D affine matrix, that would have the same transformational effect.

*Declaration:* **br_matrix34* BrQuatToMatrix34(br_matrix34* mat, const br_quat* q)**

*Arguments:* **br_matrix34 * mat**

A pointer to the destination matrix to receive the conversion.

**const br_quat * q**

A pointer to the source unit quaternion.

*Result:* **br_matrix34 ***

Returns `mat` for convenience.

*Remarks:* The resulting matrix at `mat` is equivalent to the following:

$$w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k} \equiv \begin{pmatrix} 1-4(y^2+z^2) & 4(xy+wz) & 4(xz-wy) & 0 \\ 4(xy-wz) & 1-4(x^2+z^2) & 4(yz+wx) & 0 \\ 4(xz+wy) & 4(yz-wx) & 1-4(x^2+y^2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

## BrQuatToMatrix4()

*Description:* Convert a unit quaternion to a 3D affine matrix, that would have the same transformational effect.

*Declaration:* **br_matrix4* BrQuatToMatrix4(br_matrix4* mat, const br_quat* q)**

*Arguments:* **br_matrix4 * mat**

A pointer to the destination matrix to receive the conversion.

**const br_quat * q**

A pointer to the source unit quaternion.

*Result:* **br_matrix4 ***

Returns `mat` for convenience.

*See Also:* **BrQuatToMatrix34()** 323.

## Copy/Assign

Use structure assignment to copy quaternions.

## Access & Maintenance

Members may be freely accessed. Maintenance is only required for unit quaternions that have been modified.

## BrQuatNormalise()

*Description:* Normalise a quaternion.

*Declaration:* **br_quat* BrQuatNormalise(br_quat* q, const br_quat* qq)**

*Arguments:* **br_quat * q**

A pointer to the destination quaternion (may be same as source).

**const br_quat * qq**

A pointer to the source quaternion.

*Effects:* The destination is each element of the source divided by its magnitude.

*Remarks:* BRender assumes that only unit quaternions are used, and due to this assumption being embodied in certain calculations, if non-unit quaternions are used it is likely that unwanted side effects such as scaling will be introduced. This function ensures that the quaternion is a unit one, and is intended to be applied to unit quaternions that have resulted from a large number of intermediate calculations and may have drifted (due to precision limitations) from unit magnitude.

# Initialisation

The following macro may be used as a static initialiser. Note the order in which the components are specified.

**BR_QUAT(x,y,z,w)**

Macro expands to
`{BR_SCALAR(x),BR_SCALAR(y),BR_SCALAR(z),BR_SCALAR(w)}`.

All other initialisation should be member-wise or by using structure assignment.

# br_renderbounds_cbfn

## The Call-Back Function

This type defines a call-back function, which can be specified by using the function **Br[Zb|Zs]RenderBoundsCallbackSet()** 38|39. It is called for each rendered model actor in the actor hierarchies supplied to **Br[Zb|Zs]SceneRender()** 34|34 and **Br[Zb|Zs]SceneRenderAdd()** 36|37 (it may also be called by **Br[Zb|Zs]ModelRender()** 253|254). It enables an application to perform extra rendering for particular actors, or to perform computations that require information only obtainable just after a model actor has been processed by the rendering engine.

### *The **typedef***

(See *zbproto.h* for a precise declaration)

```
void br_renderbounds_cbfn(br_actor*, const br_model*, const br_material*, void*,
                          br_uint_8, const br_matrix4*, const br_int_32 [4])
```
***Render bounds call-back***

### *Related Functions*

For details of how to specify that a render bounds call-back function should be called during rendering see **Br[Zb|Zs]RenderBoundsCallbackSet()** 38|39.

Functions dedicated for use within rendering call-backs: **Br[Zb|Zs]ModelRender()** 253|254, **BrOnScreenCheck()** 254, **BrOriginToScreenXY()** 255, **BrPointToScreenXY()** 255, **BrPointToScreenXYMany()** 256, **BrOriginToScreenXYZO()** 256, **BrPointToScreenXYZO()** 257, **BrPointToScreenXYZOMany()** 258.

### *Related Structures*

See **br_model_custom_cbfn** 251 for a substitute model rendering call-back.

## Specification

---

## CBFnRenderBounds()

*Description:* An application defined call-back function that is called for each rendered model, at some point during rendering – at precisely what point is undefined, except that its parents have been processed (though not necessarily rendered). The pass through equivalent for this call-back is to do nothing. The function may call any non-rendering functions available to a model's custom call-back function – see **br_model_custom_cbfn** 251.

*Declaration:*

```
void BR_CALLBACK CBFnRenderBounds(br_actor* actor,
const br_model* model, const br_material* material,
void* render_data, br_uint_8 style,
const br_matrix4* model_to_screen,
const br_int_32 bounds[4])
```

*Arguments:*

**br_actor * actor**

Pointer to model actor referencing the model.

**const br_model * model**

Pointer to a model that has affected the rendering.

**const br_material * material**

Pointer to actor's material if defined, or default material otherwise.

**void * render_data**

If the function is called during a rendering performed by the Z-Sort renderer this will point to the order table into which the model actor's primitives have been inserted.

**br_uint_8 style**

Actor's rendering style, or default. BRender will not supply BR_RSTYLE_BOUNDING_- or BR_RSTYLE_NONE.

**const br_matrix4 * model_to_screen**

A pointer to a matrix giving the model to screen transformation.

**const br_int_32 bounds[4]**

An array containing minimum and maximum screen ordinates of pixels that would be modified in the process of rendering the model. Use the following symbols to obtain each ordinate:

| Index Symbol | Ordinate Obtained |
|---|---|
| BR_BOUNDS_MIN_X | Left-most column in which pixels are modified |
| BR_BOUNDS_MAX_X | Right-most column in which pixels are modified |
| BR_BOUNDS_MIN_Y | Top-most row in which pixels are modified |
| BR_BOUNDS_MAX_Y | Bottom-most row in which pixels are modified |

*Preconditions:* BRender has completed initialisation. Rendering is in progress. The rendering engine has determined that the model will affect pixels in the output buffers.

*Effects:* Behaviour is up to the application. Any of the operations described for **br_model_custom_cbfn**$_{251}$ can be used except **Br[Zb|Zs]ModelRender()**$_{253|254}$.

*Remarks:* This function may also be called as a result of calling **Br[Zb|Zs]ModelRender()**$_{253|254}$, so if you are highlighting a model's edges say, be careful that you don't accidentally over recurse.

Any other BRender functions may be called from within this call-back with the following restrictions:

- Don't call any rendering functions.
- Don't modify any light, clip-plane or camera actors.
- Do not access output buffers until rendering has completed.
- Don't change the environment actor.
- For best performance, avoid adding, updating or removing registry items – try to do these things before rendering.
- Do not modify the actor hierarchy.

*Example:* Possible uses include:

- Dirty Rectangle Tracking (tracking modified areas of the screen pixel map).
- Monitoring the set of model actors currently on screen
- Labelling, selection, highlighting
- Collision detection (not necessarily indicating the best method)

Note that a suitable call to clear the area of a pixel map for the purpose of clearing dirty rectangles only, is as follows:

```
BrPixelmapDirtyRectangleFill
(  pixel_map
,  bounds[BR_BOUNDS_MIN_X]
,  bounds[BR_BOUNDS_MIN_Y]
,  bounds[BR_BOUNDS_MAX_X]-bounds[BR_BOUNDS_MIN_X]+1L
,  bounds[BR_BOUNDS_MAX_Y]-bounds[BR_BOUNDS_MIN_Y]+1L
,  0
)
```

*See Also:* **br_model_custom_cbfn**$_{251}$, **br_primitive_cbfn**$_{315}$, **br_pick2d_cbfn**$_{272}$, **br_pick3d_cbfn**$_{275}$.

---

# Operations

The following functions are provided solely for use within ***CBFnModelCustom()***$_{251}$, ***CBFnPrimitive()***$_{316}$ and ***CBFnRenderBounds()***$_{316}$ functions (see **br_model_custom_cbfn**$_{251}$).

**BrOnScreenCheck()**$_{254}$
**BrOriginToScreenXY()**$_{255}$
**BrPointToScreenXY()**$_{255}$
**BrPointToScreenXYMany()**$_{256}$
**BrOriginToScreenXYZO()**$_{256}$
**BrPointToScreenXYZO()**$_{257}$

br_renderbounds_cbfn

**BrPointToScreenXYZOMany()** 258

# br_resclass_enum_cbfn

## The Call-Back Function

This type defines a function, supplied to **BrResClassEnum()** 340, and to be called by it for a selection of resource classes.

(See *fwproto.h* for a precise declaration)

*The* ***typedef***

**br_uint_32** **br_resclass_enum_cbfn(br_resource_class *,void *)***Enumerator*

## Specification

---

## CBFnResClassEnum()

*Description:* An application defined call-back function accepting a resource class and an application supplied argument (as supplied to **BrResClassEnum()** 340).

*Declaration:* **br_uint_32 BR_CALLBACK CBFnResClassEnum(br_resource_class* item, void* arg)**

*Arguments:* **br_resource_class * item**

One of the resource classes selected by **BrResClassEnum()** 340.

**void * arg**

The argument supplied to **BrResClassEnum()** 340.

*Preconditions:* BRender has completed initialisation.

*Effects:* Application defined. Avoid adding or removing resource classes within this function.

*Result:* **br_uint_32**

Any non-zero value will terminate the enumeration and be returned by **BrResClassEnum()** 340. Return zero to continue the enumeration.

*See Also:* **BrResClassEnum()** 340, **BrResChildEnum()** 50.

---

# br_resclass_find_cbfn

## The Call-Back Function

This type defines a function, registered with **BrResClassFindHook()** 339, to be called when **BrResClassFind()** 338 or **BrResClassFindMany()** 338 fail to find any resource class.

(See *fwproto.h* for a precise declaration)

*The* ***typedef***

**br_resource_class *** **br_resclass_find_cbfn(const char *)** ***Find (when*** **BrResClassFind()** ***fails)***

## Specification

---

## CBFnResClassFind()

*Description:* An application defined call-back function used when **BrResClassFind()** 338 or **BrResClassFindMany()** 338 fail.

*Declaration:*
```
br_resource_class* BR_CALLBACK
CBFnResClassFind(const char* name)
```

*Arguments:* **const char * name**

The search pattern supplied to **BrResClassFind()** 338 or **BrResClassFindMany()** 338 that did not match any resource class.

*Preconditions:* BRender has completed initialisation. No resource class has an identifier that successfully matches the search pattern.

*Effects:* Application defined.

*Result:* **br_resource_class ***

Either return an existing resource class that is deemed appropriate for the search pattern, or NULL if there isn't one. This value will be returned by **BrResClassFind()** 338 or **BrResClassFindMany()** 338.

*Remarks:* This could either be used to supply a default resource or to create a resource class. If some resource classes were created on demand, then this function could search another list of available resource classes (but not yet created) and see if the pattern matched any of them, if it did, one of them could be registered and returned.Note that there is no way to supply more than one resource class.

*See Also:* **BrResClassFind()** 338, **BrResClassFindMany()** 338, **BrResClassFindHook()** 339.

---

# br_resenum_cbfn

## The Call-Back Function

This type defines a function, supplied to **BrResChildEnum()**$_{50}$, and to be called by it for each child resource block attached to a specific resource block.

(See *fwproto.h* for a precise declaration)

### *The* ***typedef***

**br_uint_32** **br_resenum_cbfn(void *,void *)** ***Enumerator***

## Specification

---

## CBFnResEnum()

*Description:* An application defined call-back function accepting a resource block and an application supplied argument (as supplied to **BrResChildEnum()**$_{50}$).

*Declaration:* **br_uint_32 BR_CALLBACK CBFnResEnum(void* vres, void* arg)**

*Arguments:* **void * vres**

One of the resource blocks attached as a child of the resource block supplied to **BrResChildEnum()**$_{50}$.

**void * arg**

The argument as supplied to **BrResChildEnum()**$_{50}$.

*Preconditions:* BRender has completed initialisation.

*Effects:* Application defined.

If blocks are attached to the parent during the enumeration, they may not be included in the enumeration. Only the current, supplied child resource block should be detached or freed (if desired).

*Result:* **br_uint_32**

Any non-zero value will terminate the enumeration and be returned by **BrResChildEnum()**$_{50}$. Return zero to continue the enumeration.

*See Also:* **BrResChildEnum()**$_{50}$, **BrResClassEnum()**$_{340}$.

---

# br_resource_class

## The Structure

This structure is used for application defined resource classes.

Once the structure is initialised, it is registered using **BrResClassAdd()** 335, thereafter allowing application defined memory or resource classes to be used.

### *The* ***typedef***

(See *brmem.h* for precise declaration and ordering)

Primary

| | | |
|---|---|---|
| **br_uint_8** | **res_class** | ***Resource memory class*** |
| **br_resourcefree_cbfn*** | **free_cb** | ***Resource destructor*** |

Supplementary

| | | |
|---|---|---|
| **char *** | **identifier** | ***Resource class name*** |

### *Related Functions*

Resource Management

See **BrResAllocate()** 48, **BrResFree()** 51, **BrResAdd()** 49, **BrResRemove()** 51, **BrResStrDup()** 49, **BrResSize()** 50, **BrResClass()** 49, **BrResChildEnum()** 50.

Memory Management

See **BrMemAllocate()** 55, **BrMemCalloc()** 55, **BrMemInquire()** 55, **BrMemFree()** 56, **BrMemStrDup()** 56.

### *Related Structures*

Memory Management

See **br_allocator** *102*.

## Members

### *Primary*

#### br_uint_8 res_class

The memory class of the new resource class. See Memory Classes for a description of possible values. Only values between BR_MEMORY_APPLICATION+1 and BR_MEMORY_MAX-1 (inclusive) should be used.

#### br_resourcefree_cbfn * free_cb

This is a pointer to the destructor function (See **br_resourcefree_cbfn**$_{341}$) it may be NULL if not required. Note that this function is not used to free the memory of the resource. It is simply an opportunity for the application to perform any other housekeeping functions indicated by the destruction of the resource. The resource is destroyed after the destructor returns.

For example, a user defined resource may be of structures containing pointers to reference counted (non-resource) items. In such a case, the referenced item will need to be dereferenced.

### *Supplementary*

#### char * identifier

Pointer to unique, zero terminated, character string (or NULL if not required). A string constant is recommended. The alternative is to use something like:

```
my_res_class=BrResAllocate(NULL,sizeof(br_resource_class),
                                BR_MEMORY_RESOURCE_CLASS);
my_res_class->identifier=BrResStrDup(my_res_class,"MyClass");
```

## Operations

---

# BrResClassAdd()

*Description:* Create a new resource class.

*Declaration:* **br_resource_class* BrResClassAdd(br_resource_class* rclass)**

*Arguments:* **br_resource_class * rclass**

A non-NULL pointer to a resource class. The res_class member of rclass must be set to a valid, unused class ID between BR_MEMORY_APPLICATION+1 and BR_MEMORY_MAX-1 (inclusive).

*Preconditions:* Between **BrBegin()**$_{10}$ & **BrEnd()**$_{11}$.

*Result:* **br_resource_class ***

Returns a pointer to the resource class or NULL if unsuccessful.

*Remarks:* Remember, that rclass must point to a resource class structure that will remain valid until the class is removed.

---

# BrResClassAddMany()

*Description:* Create several new resource classes.

*Declaration:*
```
br_uint_32
BrResClassAddMany(br_resource_class* const* items, int n)
```

*Arguments:* **br_resource_class * const * items**

A non-NULL pointer to a series of pointers to resource classes. The res_class member of each resource class structure pointed to by items must be set to a valid, unused class ID between BR_MEMORY_APPLICATION+1 and BR_MEMORY_MAX-1 (inclusive).

**int n**

Number of new resource classes to create.

*Preconditions:* Between **BrBegin()**[10] & **BrEnd()**[11].

*Result:* **br_uint_32**

Returns the number of resource classes successfully created.

*Remarks:* Remember, that each resource class structures must remain valid until it is removed.

---

# BrResClassRemove()

*Description:* Remove a resource class from use.

*Declaration:*
```
br_resource_class*
BrResClassRemove(br_resource_class* rclass)
```

*Arguments:* **br_resource_class * rclass**

A non-NULL pointer to a resource class (that has been previously created using **BrResClassAdd()**[335]).

*Preconditions:* Between **BrBegin()**[10] & **BrEnd()**[11].

*Result:* **br_resource_class ***

Returns a pointer to the removed resource class.

---

# BrResClassRemoveMany()

*Description:* Remove a number of resource classes from use.

*Declaration:* **br_uint_32 BrResClassRemoveMany(br_resource_class* const* items, int n)**

*Arguments:* **br_resource_class * const * items**

A non-NULL pointer to an array of pointers to resource classes (all having previously been created using **BrResClassAdd()**[335]).

**int n**

Number of resource classes to remove from use.

*Preconditions:* Between **BrBegin()**[10] & **BrEnd()**[11].

*Result:* **br_uint_32**

Returns the number of resource classes successfully removed from use.

---

## Copy/Assign

Beware of copying the structure if identifier has been allocated from the heap. Do not use the same structure for different resource classes.

## Access & Maintenance

While a current resource class, the members should not be changed. No maintenance required.

## Referencing & Lifetime

The structure must remain valid until the class is removed using **BrResClassRemove()**[336]. BRender will not necessarily make a copy of the **br_resource_class**[334] structure.

## Initialisation

The members should be set before the structure is passed to **BrResClassAdd()**[335].

## Construction & Destruction

The structure should ideally be statically constructed, but could be constructed and destroyed as in the following code:

```
br_resource_class* my_res_class;
my_res_class=BrResAllocate(NULL,sizeof(br_resource_class),
                                BR_MEMORY_RESOURCE_CLASS);
.../* Set members */
#define MY_RES_CLASS BR_MEMORY_APPLICATION+1
my_res_class->res_class=MY_RES_CLASS;
my_res_class->free_cb=NULL;
my_res_class->identifier="MyResClass";

BrResClassAdd(my_res_class);
```

```
...
BrResFree(my_res_class);
```

## Supplementary

The `identifier` may be used to specify a resource class by name. Not recommended for intensive use. Locate resources, and keep a record of pointers. The following functions provide searching, counting and enumeration facilities for resource classes.

---

# BrResClassFind()

*Description:* Find a resource class in the registry by name. A call-back function can be set up to be called if the search is unsuccessful. The search pattern can include the standard wild cards '*' and '?'.

*Declaration:* **br_resource_class* BrResClassFind(const char* pattern)**

*Arguments:* **const char * pattern**

Search pattern.

*Result:* **br_resource_class ***

Returns a pointer to the resource class if found, otherwise `NULL`. If a call-back exists and is called, the call-back's return value is returned.

---

# BrResClassFindMany()

*Description:* Find a number of resource classes in the registry by name. The search pattern can include the standard wild cards '*' and '?'.

*Declaration:* **br_uint_32 BrResClassFindMany(const char* pattern, br_resource_class** items, int max)**

*Arguments:* **const char * pattern**

Search pattern.

**br_resource_class ** items**

A pointer to an array of pointers to resource classes.

**int max**

The maximum number of resource classes to find.

*Result:* **br_uint_32**

Returns the number of resource classes found. The pointer array is filled with pointers to the resource classes.

## BrResClassFindHook()

*Description:* Set up a resource class find call-back. If **BrResClassFind()** [338] is unsuccessful and a call-back has been set up, the call-back is passed the search pattern as its only argument. The call-back should then return a pointer to a substitute or default resource class.

*Declaration:* **br_resclass_find_cbfn* BrResClassFindHook(br_resclass_find_cbfn* hook)**

*Arguments:* **br_resclass_find_cbfn * hook**

A pointer to a call-back function.

*Result:* **br_resclass_find_cbfn ***

Returns a pointer to the old call-back function.

*Example:*

```
br_resource_class BR_CALLBACK * example_callback(char* pattern)
{  br_resource_class default;
...
   return(&default);
}
{  br_resource_class *rc;
...
   BrResClassFindHook(&example_callback);
   rc = BrResClassFind("non_existant_class");
}
```

## BrResClassCount()

*Declaration:* **br_uint_32 BrResClassCount(const char* pattern)**

*Description:* Count the number of resource classes in the registry whose names match a given search pattern. The search pattern can include the standard wild cards '*' and '?'.

*Arguments:* **const char * pattern**

Search pattern.

*Result:* **br_uint_32**

Returns the number of resource classes matching the search string.

# BrResClassEnum()

*Description:* Call a call-back function for every resource class matching a given search pattern. The call-back is passed a pointer to each matching resource class, and its second argument is an optional pointer supplied by the user. The search pattern can include the standard wild cards '*' and '?'. The call-back itself returns a **br_uint_32**$_{358}$ value. The enumeration will halt at any stage if the return value is non-zero.

*Declaration:* **br_uint_32 BrResClassEnum(char* pattern, br_resclass_enum_cbfn* callback, void* arg)**

*Arguments:* **char * pattern**

Search pattern.

**br_resclass_enum_cbfn * callback**

A pointer to a call-back function.

**void * arg**

An optional argument to pass to the call-back function.

*Result:* **br_uint_32**

Returns the first non-zero call-back return value, of zero if all resource classes are enumerated.

*Example:*

```
br_uint_32 BR_CALLBACK example_callback(br_resource_class* item,
   void* arg)
{  br_uint_32 value;
...
   return(value);
}
{  br_uint_32 ev;
...
   ev = BrResClassEnum("pattern",&example_callback,NULL);
}
```

# br_resourcefree_cbfn

## The Call-Back Function

This type defines a 'destructor' function, optionally specified in the resource class descriptor **br_resource_class**$_{334}$, that is called just before a resource block is freed.

(See *fwproto.h* for a precise declaration)

*The* ***typedef***

**void** **br_resourcefree_cbfn(void*,br_uint8,br_size_t)** ***Destructor***

## Specification

---

## CBFnResourceFree()

*Description:* An application defined call-back function accepting details of a resource block just before it will be freed.

*Declaration:* **void BR_CALLBACK CBFnResourceFree(void* res, br_uint_8 res_class, br_size_t size)**

*Arguments:* **void * res**

The resource block about to be freed.

**br_uint_8 res_class**

The memory class of the resource block's resource class.

**br_size_t size**

The size of the resource block (useful for arrays).

*Preconditions:* BRender has completed initialisation.

*Effects:* Application defined.

Do not free the block – this will be performed after the function returns. This function will also be subsequently called for any child resource blocks still attached.

*See Also:* **BrResFree()**$_{51}$.

---

# br_scalar

## The Integral Type

The **br_scalar**$_{342}$ type is the general numerical representation used by BRender. Under the fixed point library, **br_scalar**$_{342}$ is a 32 bit fixed point number (sign + 15 bit integer + 16 bit fraction), and can represent numbers between approximately -32768 and +32768. Under the floating point library, **br_scalar**$_{342}$ is a `float`, and can represent numbers between approximately -3.4e38 and +3.4e38.

Use **br_scalar**$_{342}$ instead of int, float or double, in all numerical modelling aspects of an application program.

### *The* ***typedef***

(See *scalar.h* for precise declaration)

| | | |
|---|---|---|
| **float** | **br_scalar** | ***Floating point*** |
| **br_fixed_ls** | **br_scalar** | ***Fixed – Long Signed(15.16)*** |

## Arithmetic

No standard arithmetic operators are supported. The following macros should be used instead. All arguments and return values are of type **br_scalar**$_{342}$ (except where otherwise stated).

### *Macros for Standard Operations*

**BR_ADD(a,b)**

Return the equivalent of `a + b` .

**BR_SUB(a,b)**

Return the equivalent of `a – b` .

**BR_CONST_MUL(a,x)**

Return the equivalent of `a * x` (where `x` is a numeric constant).

**BR_MUL(a,b)**

Return the equivalent of `a * b` .

**BR_SQR(a)**

Return the equivalent of `a * a` .

**BR_RCP(a)**

Return the equivalent of `1.0 / a` .

**BR_CONST_DIV(a,x)**

Return the equivalent of `a / x` (where `x` is a numeric constant).

**BR_DIV(a,b)**

Return the equivalent of `a / b`.

**BR_DIVR(a,b)**

Return the equivalent of `a / b` (rounding toward zero instead of negative infinity*).

**BR_MULDIV(a,b,c)**

Return the equivalent of `a * b / c` .

### *Macros for Standard Functions*

**BR_ABS(a)**

Return the equivalent of `fabs( a )` .

**BR_POW(a,b)**

Return the equivalent of `pow( a , b )` .

**BR_SQRT(a)**

Return the equivalent of `sqrt( a )` .

### *Macros for Vector and Matrix Oriented Arithmetic*

#### For Two Pairs of Operands

**BR_SQR2(a,b)**

Return the equivalent of `a * a + b * b` .

**BR_LENGTH2(a,b)**

Return the equivalent of `sqrt( a * a + b * b )` .

**BR_RLENGTH2(a,b)**

Return the equivalent of `( 1.0 / sqrt( a * a + b * b ))` .

**BR_MAC2(a,b,c,d)**

Return the equivalent of `a * b + c * d` .

**BR_MAC2DIV(a,b,c,d,e)**

Return the equivalent of `( a * b + c * d ) / e` .

#### For Three Pairs of Operands

**BR_SQR3(a,b,c)**

Return the equivalent of `a * a + b * b + c * c` .

**BR_LENGTH3(a,b,c)**

Return the equivalent of `sqrt( a * a + b * b + c * c )` .

**BR_RLENGTH3(a,b,c)**

Return the equivalent of `( 1.0 / sqrt( a * a + b * b + c * c ))` .

---

* Sometimes this difference between `BR_DIV()` and `BR_DIVR()` can be important when dealing with fixed point values.

**BR_MAC3(a,b,c,d,e,f)**

Return the equivalent of `a * b + c * d + e * f .`

**BR_MAC3DIV(a,b,c,d,e,f,g)**

Return the equivalent of `( a * b + c * d + e * f ) / g .`

### For Four Pairs of Operands

**BR_SQR4(a,b,c,d)**

Return the equivalent of `a * a + b * b + c * c + d * d .`

**BR_LENGTH4(a,b,c,d)**

Return the equivalent of `sqrt( a * a + b * b + c * c + d * d ) .`

**BR_RLENGTH4(a,b,c,d)**

Return the equivalent of `( 1.0 / sqrt( a * a + b * b + c * c + d * d )) .`

**BR_MAC4(a,b,c,d,e,f,g,h)**

Return the equivalent of `a * b + c * d + e * f + g * h .`

**BR_MAC4DIV(a,b,c,d,e,f,g,h,i)**

Return the equivalent of `( a * b + c * d + e * f + g * h ) / i .`

## Comparison

Equality and comparison with zero are valid. Other standard comparison operators may be implemented by macros in future versions, but are currently valid.

## Conversion

### *From Numeric Constants*

**BR_SCALAR(x)**

Convert `x` from any numeric constant to scalar type. Also see `BR_CONST_MUL()` and `BR_CONST_DIV()`.

**BR_SCALAR_EPSILON**

Smallest representable positive scalar value.

**BR_SCALAR_MIN**

Largest representable negative scalar value.

**BR_SCALAR_MAX**

Largest representable positive scalar value.

*From Integral Types*

**br_scalar BrIntToScalar(int i)**

Converts `i` from any integer type to scalar type.

**br_scalar BrFloatToScalar(int i)**

Converts `i` from `float` or `double` type to scalar type.

*From Other BRender Types*

**br_scalar BrAngleToScalar(br_angle a)**

Converts `a` from angle type to scalar type.

**br_scalar BrFixedToScalar(br_fixed_ls f)**

Converts `f` from fixed type to scalar type.

**br_scalar BrFractionToScalar(br_fraction f)**

Converts `f` from fraction type to scalar type.

**br_scalar BrUFractionToScalar(br_ufraction u)**

Converts `u` from unsigned fraction type to scalar type.

*To Integral Types*

**int BrScalarToInt(br_scalar s)**

Converts `s` from scalar type to `int`.

**float BrScalarToFloat(br_scalar s)**

Converts `s` from scalar type to `float`.

*To Other BRender Types*

**br_angle BrScalarToAngle(br_scalar s)**

Converts `a` from scalar type to angle type.

**br_fixed_ls BrScalarToFixed(br_scalar s)**

Converts `s` from scalar type to fixed type.

**br_fraction BrScalarToFraction(br_scalar s)**

Converts `f` from scalar type to fraction type.

**br_ufraction BrScalarToUFraction(br_scalar s)**

Converts `u` from scalar type to unsigned fraction type.

## Copy/Assign

Only assign zero, or scalars. Use conversions in all other cases.

# br_size_t

## The Integral Type

BRender's size type – in case the target environment does not define a `size_t` type. Use this type to represent sizes, as returned by the `sizeof()` operator.

### *The* ***typedef***

(See *`compiler.h`* for precise declaration)

**`unsigned int`** **`br_size_t`** ***Generic size type***

## Arithmetic

All standard C arithmetic operators are valid (though not necessarily sensible) as with standard integer types.

## Comparison

All standard C comparison operators are valid as with standard integer types.

## Conversion

Use casts as with any other standard C type.

## Copy/Assign

Use as is the convention with the `size_t` type.

# br_table_enum_cbfn

## The Call-Back Function

This type defines a function, supplied to **BrTableEnum()** 297, and to be called by it for a selection of tables.

### *The* ***typedef***

(See *fwproto.h* for a precise declaration)

**br_uint_32 br_table_enum_cbfn(br_pixelmap*, void*)***Enumerator*

## Specification

---

## CBFnTableEnum()

*Description:* An application defined call-back function accepting a table and an application supplied argument (as supplied to **BrTableEnum()** 297).

*Declaration:* **br_uint_32 BR_CALLBACK CBFnTableEnum(br_pixelmap* table, void* arg)**

*Arguments:* **br_pixelmap * table**

One of the tables selected by **BrTableEnum()** 297.

**void * arg**

The argument supplied to **BrTableEnum()** 297.

*Preconditions:* BRender has completed initialisation.

*Effects:* Application defined. Avoid adding or removing tables within this function.

*Result:* **br_uint_32**

Any non-zero value will terminate the enumeration and be returned by **BrTableEnum()** 297. Return zero to continue the enumeration.

*See Also:* **BrTableEnum()** 297, **BrTableFind()** 298.

---

# br_table_find_cbfn

## The Call-Back Function

This type defines a function, registered with **BrTableFindHook()** 299, to be called when **BrTableFind()** 298 or **BrTableFindMany()** 298 fail to find any table.

### *The* ***typedef***

(See *fwproto.h* for a precise declaration)

**br_pixelmap*** **br_table_find_cbfn(const char*)** ***Find (when*** **BrTableFind()** ***fails)***

## Specification

---

## CBFnTableFind()

*Description:* An application defined call-back function used when **BrTableFind()** 298 or **BrTableFindMany()** 298 fail.

*Declaration:* **br_pixelmap* BR_CALLBACK CBFnTableFind(const char* name)**

*Arguments:* **const char * name**

The search pattern supplied to **BrTableFind()** 298 or **BrTableFindMany()** 298 that did not match any table.

*Preconditions:* BRender has completed initialisation. No table has an identifier that successfully matches the search pattern.

*Effects:* Application defined.

*Result:* **br_pixelmap ***

Either return an existing table that is deemed appropriate for the search pattern, or NULL if there isn't one. This value will be returned by **BrTableFind()** 298 or **BrTableFindMany()** 298.

*Remarks:* This could either be used to supply a default table or to create a table. If tables were created on demand, then this function could search another list of available tables (but not yet created) and see if the pattern matched any of them, if it did, one of them could be registered and returned. Note that there is no way to supply more than one table.

*See Also:* **BrTableFind()** 298, **BrTableFindMany()** 298, **BrTableFindHook()** 299, **BrTableFindFailedLoad()** 299.

---

# br_transform

## The Structure

This is BRender's generic transformation type, primarily used to specify a transformation from one actor's space to another's. In an actor it represents the transform to be applied to co-ordinates (such as of a model) in its space to bring them into the co-ordinate space of its parent.

### *The* ***typedef***

(See *transform.h* for precise declaration and ordering)

Transform Type

| | | |
|---|---|---|
| **br_uint_16** | **type** | ***Specifies how the transformation is represented.*** |

Translation Transform

| | | |
|---|---|---|
| **br_vector3** | **t.translate.t** | ***Translation for the Translation transform type*** |

Euler Transform

| | | |
|---|---|---|
| **br_vector3** | **t.euler.t** | ***Translation for the Euler transform type*** |
| **br_euler** | **t.euler.e** | ***Euler angle set of the Euler transform type*** |

Look Up Transform

| | | |
|---|---|---|
| **br_vector3** | **t.look_up.t** | ***Translation for the Look Up transform type*** |
| **br_vector3** | **t.look_up.look** | ***Look-at vector for the Look Up transform type*** |
| **br_vector3** | **t.look_up.up** | ***Look-up vector for the Look Up transform type*** |

Quaternion Transform

| | | |
|---|---|---|
| **br_vector3** | **t.quat.t** | ***Translation for the Quaternion transform type*** |
| **br_quat** | **t.quat.q** | ***Quaternion rotation for the Quaternion transform type*** |

Matrix Transform

| | | |
|---|---|---|
| **br_matrix34** | **t.mat** | ***Matrix34 for both Matrix transform types*** |

### *Related Functions*

Scene Modelling

See **BrActorToActorMatrix34()** 84 and **BrActorToScreenMatrix4()** 85.

### *Related Structures*

Scene Modelling

See **br_actor** 76.

# Members

### *Transform Type*

## br_uint_16 type

This member defines which other members of the transform structure have meaning. It should never be modified directly except for initialisation purposes. Refer to **BrTransformToTransform()** 353 for details of how to convert from one transform to another.

This member may have any one of the following values:

| Value Symbol | Meaning |
|---|---|
| BR_TRANSFORM_IDENTITY | The transform is the identity. |
| BR_TRANSFORM_TRANSLATION | The transform is a translation only (held in t.translate.t). |
| BR_TRANSFORM_EULER | The transform is represented by a Euler angle set (t.euler.e) and a translation (t.euler.t). |
| BR_TRANSFORM_LOOK_UP | The transform is represented by a look-at vector (t.look_up.look), an up vector (t.look_up.up) and a translation (t.look_up.t). |
| BR_TRANSFORM_QUAT | The transform is represented by a quaternion (t.quat.q) and a translation (t.quat.t). |
| BR_TRANSFORM_MATRIX34 | The transform is represented by a 3x4 affine matrix (t.mat), which is the most general representation. |
| BR_TRANSFORM_MATRIX34_LP | The transform is represented by a 3x4 length preserving matrix (t.mat). |

### *Identity Transform*

The identity transform is engaged when type is set to BR_TRANSFORM_IDENTITY. This is effectively a NULL transform, making actors effectively share the same co-ordinate space as their parent.

### *Translation Transform*

The translation transform is engaged when type is set to BR_TRANSFORM_TRANSLATION. This is solely a translation, i.e. no rotation or scaling is involved.

## br_vector3 t.translate.t

This member only has meaning for this transform type*. It contains the vector representing the translation.

* However, upon inspection of **br_transform** it can be seen that it is effective for all transforms apart from the identity. Use of this feature is for internal use only.

### *Euler Transform*

The Euler transform is engaged when `type` is set to `BR_TRANSFORM_EULER`. This consists of a three rotations about a combination of orthogonal axes and a translation (effectively applied after the rotations). See **`br_euler`**$_{121}$ for more information.

#### `br_vector3 t.euler.t`

This member contains the vector representing the translation component of the transform.

#### `br_vector3 t.euler.e`

This member contains the vector representing the rotation components of the transform.

### *Look Up Transform*[*]

The Look Up transform effectively consists of a pair of rotations and a translation. The first rotation is defined about the origin between the negative z axis and the `look` vector. The second rotation is defined about the new z axis between the new positive x axis and the cross product of the `up` and `look` vectors. The translation is then applied.

The Look Up transform is a convenient method of making an actor such as a camera point toward a particular position. It has the effect of making the actor's negative[†] z axis lie along the `look` vector, and its positive y axis lie in the plane defined by the `up` and `look` vectors (or more precisely, its positive x axis will lie along the cross product of the `up` and `look` vectors). Remember that the `look` and `up` vectors will be in the co-ordinate space of the actor's parent and thus from its point of view, so if pointing at a particular actor, in order to compute `look`, that actor's co-ordinates will also need to be transformed into the co-ordinate space of the parent (see **`BrActorToActorMatrix34()`**$_{84}$). Incidentally, things can be simplified by using intermediate dummy actors (`BR_ACTOR_NONE`).

The translation is applied at the end.

#### `br_vector3 t.look_up.t`

This member contains the vector representing the translation component of the transform.

#### `br_vector3 t.look_up.look`

This member contains the vector that defines the rotation of the negative z axis.

#### `br_vector3 t.look_up.up`

This member contains the vector that defines the rotation about the look vector.

---

* Not represented by a separate BRender structure.

† Negative rather than positive in order to be consistent with the camera definition (which is the predominant actor expected to utilise the Look Up transform).

br_transform

### *Quaternion Transform*

The unit quaternion transform represents a rotation about an arbitrary vector.

#### br_vector3 t.quat.t

This member contains the vector representing the translation component of the transform.

#### br_vector3 t.quat.q

This member contains the vector representing the rotation component of the transform.

### *Matrix Transforms*

Matrix transforms represent BRender's most general form of transformation. See **br_matrix34**[190] for details of how matrix transformations are performed.

BR_TRANSFORM_MATRIX34_LP is a slightly more specialised form which requires that the matrix have no scaling effect, i.e. is length preserving.

#### br_matrix34 t.mat

This member contains the 3D affine matrix representing the entire transform.

## Arithmetic

See **BrMatrix34PreTransform()**[201], **BrMatrix34PostTransform()**[206], and **BrMatrix4PreTransform()**[223].

## Conversion

Also see **BrMatrix34ToEuler()**[207], **BrMatrix34ToQuat()**[207], **BrEulerToMatrix34()**[123] and **BrQuatToMatrix34()**[323].

### *From Matrices and Other Transforms*

See **BrMatrix34ToTransform()**[208] and **BrTransformToTransform()**[353].

### *To Matrices and Other Transforms*

See **BrTransformToMatrix34()**[353] and **BrTransformToTransform()**[353] as described below.

## BrTransformToMatrix34()

*Description:* Convert a generic transform to a 3D affine matrix, that would have the same transformational effect.

*Declaration:* **void BrTransformToMatrix34(br_matrix34* mat, const br_transform* xform)**

*Arguments:* **br_matrix34 * mat**

A pointer to the destination matrix to receive the conversion.

**const br_transform * xform**

A pointer to the source generic transform.

*Effects:* *When the transform is the identity*

Uses to **BrMatrix34Identity()** 211 to re-initialise the destination matrix to the identity matrix.

*When the transform is a translation*

Uses to **BrMatrix34Translate()** 211 to re-initialise the destination matrix to a translation matrix.

*When the transform is a Euler angle set*

Calls **BrEulerToMatrix34()** 123 and copies the transforms' translation to the respective elements of the destination matrix.

*When the transform is a Look-Up*

The normalised and negated Look vector provides the third row of the matrix.

The normalised cross product of this with the Up vector provides the first row.

The cross product of these two rows provides the second row.

The translation vector provides the fourth and last row.

*When the transform is a quaternion*

Calls **BrQuatToMatrix34()** 323 and copies the transforms' translation to the respective elements of the destination matrix.

*When the transform is a 3×4 matrix*

Directly copies the matrix from the source transform.

*When the transform is a 3×4 length preserving matrix*

Directly copies the matrix from the source transform.

## BrTransformToTransform()

*Description:* Convert from one transform to another, that would have the same transformational effect (where possible).

*Declaration:* **void BrTransformToTransform(br_transform* dest, const br_transform* src)**

*Arguments:* **br_transform * dest**

A pointer to the destination generic transform to receive the conversion. The destination's `type` member is unchanged.

**const br_transform * src**

A pointer to the source generic transform to be converted.

*Effects:* *When the transforms' types are the same*

The transform structure is copied entire.

*When the destination transform is of matrix type*

Calls **BrTransformToMatrix34()** 353 then normalises if necessary.

*In other cases*

Converts the transform via an intermediate 3x4 matrix by first calling **BrTransformToMatrix34()** 353 and then **BrMatrix34ToTransform()** 208.

*Remarks:* The transformation type in the destination transform must be set before conversion is performed*.

In some cases it may not be possible to preserve all components of a transformation across the conversion. For example, a conversion of a matrix to a quaternion would lose any scaling or shearing components.

---

## Copy/Assign

Although copy by structure assignment currently works, use **BrTransformToTransform()** 353 to ensure compatibility.

## Access & Maintenance

Only the members corresponding to the currently set type should be accessed. The type should not be modified except for initialisation. The structure may be re-initialised, but any current references must be expecting this. No specific maintenance required, but refer to the descriptions of underlying structures for details of their maintenance requirements.

## Referencing & Lifetime

This structure may be freely referenced, though take care when changing the type of a transform that any current references will take note. In the same vein, try to avoid referencing members of 'live' transforms. The structure should remain valid as long as required by any references.

---

* This also applies to the `order` member of Euler transforms.

## Initialisation

First set the `type` member to the desired transform type and then initialise the appropriate members using their respective assignment methods. The simplest initialisation is to the Identity, which is also the only safe static initialisation.

# br_ufraction

## The Integral Type

The **br_ufraction**$_{356}$ type can be used to represent numbers in the range [0,+1)*. Although used internally, this type is not generally supported by the BRender API. One of the few places in which it is used is in specifying lighting coefficients of materials.

Under the floating point library, **br_ufraction**$_{356}$ is a `float`. Under the fixed point library, **br_ufraction**$_{356}$ is a 16 bit unsigned fixed point number.

### *The* ***typedef***

(See *scalar.h* for precise declaration)

| | | |
|---|---|---|
| **float** | **br_ufraction** | ***Floating point Unsigned Fraction*** |
| **br_fixed_luf** | **br_ufraction** | ***Fixed – Long Unsigned Fraction(0.16)*** |

## Arithmetic

No standard operators are supported. No macros are provided. Convert to **br_scalar**$_{342}$ and use that type's arithmetic macros instead.

## Comparison

Equality and comparison with zero are valid. Other standard comparison operators may be implemented by macros in future versions, but are currently valid.

## Conversion

### *From Numeric Constants*

**BR_UFRACTION(x)**

Convert `x` from any numeric constant to **br_ufraction**$_{356}$.

**BR_SCALAR_EPSILON**

Smallest positive fractional value.

### *From Integral Types*

To convert from integral types, use **br_scalar**$_{342}$ as an intermediary.

---

* Maximum value is thus `BR_ONE_LS-BR_SCALAR_EPSILON` (in the fixed point library).

*From* **br_scalar**$_{342}$

```
br_ufraction BrScalarToUFraction(br_scalar s)
```

Converts `s` from **br_scalar**$_{342}$ to **br_ufraction**$_{356}$. It is up to the application to ensure the value is in the required range.

### *To Integral Types*

To convert to integral types, use **br_scalar**$_{342}$ as an intermediary.

*To* **br_scalar**$_{342}$

```
br_scalar BrUFractionToScalar(br_ufraction f)
```

Converts `s` from **br_ufraction**$_{356}$ to **br_scalar**$_{342}$. It is up to the application to ensure the value is in the required range.

## Copy/Assign

Only assign zero, or fractions. Use conversions in all other cases.

# br_uint_8/16/32

## The Integral Type

BRender's unsigned integer types. Use this type where the integer word length is critical.

### *The* ***typedef***

(See *compiler.h* for precise declaration)

| | | |
|---|---|---|
| **unsigned char** | **br_uint_8** | ***8 bit unsigned integer*** |
| **unsigned short** | **br_uint_16** | ***16 bit unsigned integer*** |
| **unsigned long** | **br_uint_32** | ***32 bit unsigned integer*** |

## Arithmetic

All standard C arithmetic operators are valid as with standard integer types.

## Comparison

All standard C comparison operators are valid as with standard integer types.

## Conversion

Use casts as with any other standard C type.

## Copy/Assign

Freely assign. Use as a standard C type, as this type is only concerned with ensuring specific sizes – not representation.

# br_vector2

## The Structure

This is the two ordinate vector structure, typically used for 2D purposes. Functions are provided to allow it be used as though it were an integral type.

### *The* ***typedef***

(See *vector.h* for precise declaration and ordering)

**br_scalar** **v[2]** ***Ordinates (0=x, 1=y)***

### *Related Functions*

Image Support

See **BrOriginToScreenXY()** 255, **BrPointToScreenXY()** 255, **BrPointToScreenXYMany()** 256,

Maths

See **BrMatrix23ApplyP()** 176, **BrMatrix23ApplyV()** 178, **BrMatrix23TApplyP()** 178, **BrMatrix23TApplyV()** 178.

### *Related Structures*

See **br_matrix23**_174_, **br_vertex**_378_.

## Members

### br_scalar v[2]

First and second ordinate. Conventionally, the first ordinate is the x-axis component, and the second, the y axis component.

## Arithmetic

---

# BrVector2Negate()

*Description:* Negate a vector and place the result in a second destination vector. Equivalent to the expression:

$$\mathbf{v}_1 \Leftarrow -\mathbf{v}_2$$

*Declaration:* **void BrVector2Negate(br_vector2* v1, const br_vector2* v2)**

*Arguments:* **br_vector2 * v1**

A pointer to the destination vector (may be same as source).

**const br_vector2 * v2**

A pointer to the source vector.

---

## BrVector2Add()

*Description:* Add two vectors and place the result in a third destination vector.

Equivalent to the expression:

$$\mathbf{v}_1 \Leftarrow \mathbf{v}_2 + \mathbf{v}_3$$

*Declaration:* **void BrVector2Add(br_vector2* v1, const br_vector2* v2, const br_vector2* v3)**

*Arguments:* **br_vector2 * v1**

A pointer to the destination vector (may be same as either source).

**const br_vector2 * v2**

A pointer to the first vector of the sum.

**const br_vector2 * v3**

A pointer to the second vector of the sum.

---

## BrVector2Accumulate()

*Description:* Add one vector to another. Equivalent to the expression:

$$\mathbf{v}_1 \Leftarrow \mathbf{v}_1 + \mathbf{v}_2$$

*Declaration:* **void BrVector2Accumulate(br_vector2* v1, const br_vector2* v2)**

*Arguments:* **br_vector2 * v1**

A pointer to the accumulating vector (may be same as `v2`).

**const br_vector2 * v2**

A pointer to the vector to add.

---

## BrVector2Sub()

*Description:* Subtract one vector from another and place the result in a third destination vector.

Equivalent to the expression:

$$\mathbf{v}_1 \Leftarrow \mathbf{v}_2 - \mathbf{v}_3$$

*Declaration:* **void BrVector2Sub(br_vector2* v1, const br_vector2* v2, const br_vector2* v3)**

*Arguments:* **br_vector2 * v1**

A pointer to the destination vector (may be same as `v1` or `v2`).

**const br_vector2 * v2**

A pointer to the additive vector.

**const br_vector2 * v3**

A pointer to the subtractive vector.

---

# BrVector2Scale()

*Description:* Scale a vector by a scalar and place the result in a destination vector. Equivalent to the expression:

$$\mathbf{V}_1 \Leftarrow s\mathbf{V}_2$$

*Declaration:* **void BrVector2Scale(br_vector2* v1, const br_vector2* v2, br_scalar s)**

*Arguments:* **br_vector2 * v1**

A pointer to the destination vector (may be same as source).

**const br_vector2 * v2**

A pointer to the source vector.

**br_scalar s**

Scale factor.

---

# BrVector2InvScale()

*Description:* Scale a vector by the reciprocal of a scalar and place the result in a destination vector. Equivalent to the expression:

$$\mathbf{V}_1 \Leftarrow s^{-1}\mathbf{V}_2$$

*Declaration:* **void BrVector2InvScale(br_vector2* v1, const br_vector2* v2, br_scalar s)**

*Arguments:* **br_vector2 * v1**

A pointer to the destination vector (may be same as source).

**const br_vector2 * v2**

A pointer to the source vector.

**br_scalar s**

Reciprocal scale factor.

## BrVector2Dot()

*Description:* Calculate the dot product of two vectors. Equivalent to the expression:

$$\mathbf{v}_1 \cdot \mathbf{v}_2$$

*Declaration:* **br_scalar BrVector2Dot(const br_vector2* v1, const br_vector2* v2)**

*Arguments:* **const br_vector2 * v1**

Pointer to left hand vector (may be same as `v2`).

**const br_vector2 * v2**

Pointers to right hand vector.

*Result:* **br_scalar**

Returns the dot product of the two source vectors. Equivalent to:

$$(x_1 \quad y_1) \cdot (x_2 \quad y_2) \equiv x_1 x_2 + y_1 y_2$$

## BrVector2Length()

*Description:* Calculate the length of a vector. Equivalent to the expression:

$$|\mathbf{v}_1|$$

*Declaration:* **br_scalar BrVector2Length(const br_vector2* v1)**

*Arguments:* **const br_vector2 * v1**

A pointer to the source vector.

*Result:* **br_scalar**

Returns the length of the vector. Equivalent to:

$$|(x_1 \quad y_1)| \equiv \sqrt{x_1^2 + y_1^2}$$

*See Also:* **BrVector2LengthSquared()** 363

## BrVector2LengthSquared()

*Description:* Calculate the squared length of a vector. Equivalent to the expression:

$$|\mathbf{v}_1|^2$$

or

$$\mathbf{v}_1 \cdot \mathbf{v}_1$$

*Declaration:* **br_scalar BrVector2LengthSquared(const br_vector2* v1)**

*Arguments:* **const br_vector2 * v1**

A pointer to the source vector.

*Result:* **br_scalar**

Returns the squared length of the vector. Equivalent to:

$$\left|(x_1 \quad y_1)\right|^2 \equiv x_1^2 + y_1^2$$

*See Also:* **BrVector2Length()** 362

---

# BrVector2Normalise()

*Description:* Normalise a vector and place the result in a destination vector. Equivalent to the expression:

$$\mathbf{v}_1 \Leftarrow \frac{\mathbf{v}_2}{\left|\mathbf{v}_2\right|}$$

*Declaration:* **void BrVector2Normalise(br_vector2* v1, const br_vector2* v2)**

*Arguments:* **br_vector2 * v1**

A pointer to the destination vector (may be same as source).

**const br_vector2 * v2**

A pointer to the source vector.

*Remarks:* If the source vector's length is zero* the unit vector along the x axis is stored at the destination instead.

---

## Copy/Assign

Although copy by structure assignment currently works, use **BrVector2Copy()** 364 to ensure compatibility.

---

# BrVector2Copy()

*Description:* Copy a vector. Equivalent to the expression:

$$\mathbf{v}_1 \Leftarrow \mathbf{v}_2$$

*Declaration:* **void BrVector2Copy(br_vector2* v1, const br_vector2* v2)**

---

* Or, in the fixed point library, too close to zero, i.e. less than or equal to 2*BR_SCALAR_EPSILON.

*Arguments:* **br_vector2 * v1**

A pointer to the destination vector (may be same as source).

**const br_vector2 * v2**

A pointer to the source vector.

---

## Referencing & Lifetime

This structure may be freely referenced, though take care if there is potential to supply the same vector as more than one argument to the same function.

## Initialisation

The following macro may be used as a static initialiser.

**BR_VECTOR2(a,b)**

Macro expands to `{BR_SCALAR(a),BR_SCALAR(b)}`.

All other initialisation should use the functions, **BrVector2Set()** 364, **BrVector2SetInt()** 365, **BrVector2SetFloat()** 365 or **BrVector2Copy()** 364.

---

# BrVector2Set()

*Description:* Set a vector with a pair of scalars.

*Declaration:* **void BrVector2Set(br_vector2* v1, br_scalar s1, br_scalar s2)**

*Arguments:* **br_vector2 * v1**

A pointer to the destination vector.

**br_scalar s1**

The first vector element (x axis component).

**br_scalar s2**

The second vector element (y axis component).

*Example:*

```
br_vector2 *va;
...
BrVector2Set(va,BR_SCALAR(1.0),BR_SCALAR(-1.0));
```

---

# BrVector2SetFloat()

*Description:* Set a vector from a pair of `standard C floating point numbers`.

*Declaration:* **void BrVector2SetFloat(br_vector2* v1, float f1, float f2)**

*Arguments:* **br_vector2 * v1**

A pointer to the destination vector.

**float f1**

The first vector element (x axis component).

**float f2**

The second vector element (y axis component).

*Example:*

```
br_vector2 *va;
...
BrVector2Set(va,1.f,-1.f);
```

## BrVector2SetInt()

*Description:* Set a vector from a pair of standard C integers.

*Declaration:* **void BrVector2SetInt(br_vector2* v1, int i1, int i2)**

*Arguments:* **br_vector2 * v1**

A pointer to the destination vector.

**int i1**

The first vector element (x axis component).

**int i2**

The second vector element (y axis component).

*Example:*

```
br_vector2 *va;
...
BrVector2Set(va,1,-1);
```

# br_vector3

## The Structure

This is the three ordinate vector structure, typically used for 3D calculations. Functions are provided to allow it be used as though it were an integral type.

### *The* ***typedef***

(See *vector.h* for precise declaration and ordering)

| | | |
|---|---|---|
| **br_scalar** | **v[3]** | ***Ordinates (0=x, 1=y, 2=z)*** |

### *Related Functions*

#### Image Support

See **BrOriginToScreenXYZO()** 256, **BrPointToScreenXYZO()** 257, **BrPointToScreenXYZOMany()** 258.

#### Maths

See **BrMatrix34[Pre|Post]Rotate()** [200|205], **BrMatrix[34|4][T]Apply[V|P]()** [193-196|219-222].

### *Related Structures*

See **br_matrix23**$_{174}$, **br_matrix34**$_{190}$, **br_model**$_{232}$, **br_vertex**$_{378}$, **br_transform**$_{349}$, **br_bounds**$_{108}$.

## Members

### br_scalar v[3]

First, second and third ordinate. Conventionally, the first ordinate is the x-axis component, the second, the y axis component, and the third, the z axis component. Remember that BRender has a right handed co-ordinate system and so, with the x axis positive to the right, and the y axis positive upwards, the z axis is therefore positive toward you (typically, the z axis points out of the screen).

## Arithmetic

---

# BrVector3Negate()

*Description:* Negate a vector and place the result in a second destination vector. Equivalent to the expression:

$$\mathbf{v}_1 \Leftarrow -\mathbf{v}_2$$

*Declaration:* **`void BrVector3Negate(br_vector3* v1, const br_vector3* v2)`**

*Arguments:* **`br_vector3 * v1`**

A pointer to the destination vector (may be same as source).

**`const br_vector3 * v2`**

A pointer to the source vector.

---

# BrVector3Add()

*Description:* Add two vectors and place the result in a third destination vector.

Equivalent to the expression:

$$\mathbf{v}_1 \Leftarrow \mathbf{v}_2 + \mathbf{v}_3$$

*Declaration:* **`void BrVector3Add(br_vector3* v1, const br_vector3* v2, const br_vector3* v3)`**

*Arguments:* **`br_vector3 * v1`**

A pointer to the destination vector (may be same as either source).

**`const br_vector3 * v2`**

A pointer to the first vector of the sum.

**`const br_vector3 * v3`**

A pointer to the second vector of the sum.

---

# BrVector3Accumulate()

*Description:* Add one vector to another. Equivalent to the expression:

$$\mathbf{v}_1 \Leftarrow \mathbf{v}_1 + \mathbf{v}_2$$

*Declaration:* **`void BrVector3Accumulate(br_vector3* v1, const br_vector3* v2)`**

*Arguments:* **br_vector3 * v1**

A pointer to the accumulating vector (may be same as v2).

**const br_vector3 * v2**

A pointer to the vector to add.

---

## BrVector3Sub()

*Description:* Subtract one vector from another and place the result in a third destination vector.

Equivalent to the expression:

$$\mathbf{V}_1 \Leftarrow \mathbf{V}_2 - \mathbf{V}_3$$

*Declaration:* **void BrVector3Sub(br_vector3* v1, const br_vector3* v2, const br_vector3* v3)**

*Arguments:* **br_vector3 * v1**

A pointer to the destination vector (may be same as v2 or v3).

**const br_vector3 * v2**

A pointer to the additive vector.

**const br_vector3 * v3**

A pointer to the subtractive vector.

---

## BrVector3Scale()

*Description:* Scale a vector by a scalar and place the result in a destination vector. Equivalent to the expression:

$$\mathbf{V}_1 \Leftarrow s\mathbf{V}_2$$

*Declaration:* **void BrVector3Scale(br_vector3* v1, const br_vector3* v2, br_scalar s)**

*Arguments:* **br_vector3 * v1**

A pointer to the destination vector (may be same as source).

**const br_vector3 * v2**

A pointer to the source vector.

**br_scalar s**

Scale factor.

## BrVector3InvScale()

*Description:* Scale a vector by the reciprocal of a scalar and place the result in a destination vector. Equivalent to the expression:

$$\mathbf{v}_1 \Leftarrow s^{-1}\mathbf{v}_2$$

*Declaration:* **void BrVector3InvScale(br_vector3* v1, const br_vector3* v2, br_scalar s)**

*Arguments:* **br_vector3 * v1**

A pointer to the destination vector (may be same as source).

**const br_vector3 * v2**

A pointer to the source vector.

**br_scalar s**

Reciprocal scale factor.

## BrVector3Dot()

*Description:* Calculate the dot product of two vectors. Equivalent to the expression:

$$\mathbf{v}_1 \cdot \mathbf{v}_2$$

*Declaration:* **br_scalar BrVector3Dot(const br_vector3* v1, const br_vector3* v2)**

*Arguments:* **const br_vector3 * v1**

Pointer to left hand vector (may be same as `v2`).

**const br_vector3 * v2**

Pointers to right hand vector (may be same as `v1`).

*Result:* **br_scalar**

Returns the dot product of the two source vectors. Equivalent to:

$$(x_1 \;\; y_1 \;\; z_1) \cdot (x_2 \;\; y_2 \;\; z_2) \equiv x_1x_2 + y_1y_2 + z_1z_2$$

## BrVector3Cross()

*Description:* Calculate the cross product of two vectors and store the result in a destination vector. Equivalent to the expression:

$$\mathbf{v}_1 \Leftarrow \mathbf{v}_2 \times \mathbf{v}_3$$

*Declaration:* **void BrVector3Cross(br_vector3* v1, const br_vector3* v2, const br_vector3* v3)**

*Arguments:* **br_vector3 * v1**

Pointer to destination vector (must be different from both `v1` and `v2`).

**const br_vector3 * v2**

Pointer to left hand vector.

**const br_vector3 * v3**

Pointers to right hand vector.

*Remarks:* The cross product of the two source vectors is equivalent to:

$$(x_2 \quad y_2 \quad z_2) \times (x_3 \quad y_3 \quad z_3) \equiv (y_2 z_3 - z_2 y_3 \quad z_2 x_3 - x_2 z_3 \quad x_2 y_3 - y_2 x_3)$$

---

# BrVector3Length()

*Description:* Calculate the length of a vector. Equivalent to the expression:

$$|\mathbf{v}_1|$$

*Declaration:* **br_scalar BrVector3Length(const br_vector3* v1)**

*Arguments:* **const br_vector3 * v1**

A pointer to the source vector.

*Result:* **br_scalar**

Returns the length of the vector. Equivalent to:

$$|(x_1 \quad y_1 \quad z_1)| \equiv \sqrt{x_1^2 + y_1^2 + z_1^2}$$

*See Also:* **BrVector3LengthSquared()** 371

---

# BrVector3LengthSquared()

*Description:* Calculate the squared length of a vector. Equivalent to the expression:

$$|\mathbf{v}_1|^2$$

or

$$\mathbf{v}_1 \cdot \mathbf{v}_1$$

*Declaration:* **br_scalar BrVector3LengthSquared(const br_vector3* v1)**

*Arguments:* **const br_vector3 * v1**

A pointer to the source vector.

*Result:* **br_scalar**

Returns the squared length of the vector. Equivalent to:

$$|(x_1 \quad y_1 \quad z_1)|^2 \equiv x_1^2 + y_1^2 + z_1^2$$

*See Also:* **BrVector3Length()** 371

## BrVector3Normalise()

*Description:* Normalise a vector and place the result in a destination vector. Equivalent to the expression:

$$\mathbf{v}_1 \Leftarrow \frac{\mathbf{v}_2}{|\mathbf{v}_2|}$$

*Declaration:*
```
void BrVector3Normalise(br_vector3* v1,
const br_vector3* v2)
```

*Arguments:* **br_vector3 * v1**

A pointer to the destination vector (may be same as source).

**const br_vector3 * v2**

A pointer to the source vector.

*Remarks:* If the source vector's length is zero* the unit vector along the x axis is stored at the destination instead.

## BrVector3NormaliseQuick()

*Description:* Normalise a vector with non-zero length and place the result in a destination vector. Equivalent to the expression:

$$\mathbf{v}_1 \Leftarrow \frac{\mathbf{v}_2}{|\mathbf{v}_2|}$$

*Declaration:*
```
void BrVector3NormaliseQuick(br_vector3* v1,
const br_vector3* v2)
```

*Arguments:* **br_vector3 * v1**

A pointer to the destination vector (may be same as source).

**const br_vector3 * v2**

A pointer to the source vector.

*Remarks:* No check made for zero length, hence quicker.

* Or, in the fixed point library, too close to zero, i.e. less than or equal to `2*BR_SCALAR_EPSILON`.

## BrVector3NormaliseLP()

*Description:* Normalise a vector with non-zero length using low precision* arithmetic and place the result in a destination vector. Equivalent to the expression:

$$\mathbf{v}_1 \Leftarrow \left|\mathbf{v}_2\right|^{-1}\mathbf{v}_2$$

*Declaration:*
```
void BrVector3NormaliseLP(br_vector3* v1,
const br_vector3* v2)
```

*Arguments:* **br_vector3 * v1**

A pointer to the destination vector (may be same as source).

**const br_vector3 * v2**

A pointer to the source vector.

*Remarks:* No check is made for zero length. The destination vector will be left unchanged if the reciprocal of length of the vector is zero.

## Copy/Assign

Although copy by structure assignment currently works, use **BrVector3Copy()** 373 to ensure compatibility.

## BrVector3Copy()

*Description:* Copy a vector. Equivalent to the expression:

$$\mathbf{v}_1 \Leftarrow \mathbf{v}_2$$

*Declaration:*
```
void BrVector3Copy(br_vector3* v1, const br_vector3* v2)
```

*Arguments:* **br_vector3 * v1**

A pointer to the destination vector (may be same as source).

**const br_vector3 * v2**

A pointer to the source vector.

## Referencing & Lifetime

This structure may be freely referenced, though take care if there is potential to supply the same vector as more than one argument to the same function.

* Thus LP in the function name – do not be confused with the use of this mnemonic for 'length preserving'.

## Initialisation

The following macro may be used as a static initialiser.

**BR_VECTOR3(a,b,c)**

Macro expands to `{BR_SCALAR(a),BR_SCALAR(b),BR_SCALAR(c)}`.

All other initialisation should use the functions, **BrVector3Set()** 374, **BrVector3SetInt()** 375, **BrVector3SetFloat()** 374 or **BrVector3Copy()** 373.

---

# BrVector3Set()

*Description:* Set a vector with a triple of scalars.

*Declaration:* **void BrVector3Set(br_vector3* v1, br_scalar s1, br_scalar s2, br_scalar s3)**

*Arguments:* **br_vector3 * v1**

A pointer to the destination vector.

**br_scalar s1**

The first vector element (x axis component).

**br_scalar s2**

The second vector element (y axis component).

**br_scalar s3**

The third vector element (z axis component).

*Example:*

```
br_vector3 *va;
...
BrVector3Set(va,BR_SCALAR(1.0),BR_SCALAR(-1.0),BR_SCALAR(2.0));
```

---

# BrVector3SetFloat()

*Description:* Set a vector from a triple of `standard C floating point numbers`.

*Declaration:* **void BrVector3SetFloat(br_vector3* v1, float f1, float f2, float f3)**

*Arguments:* **br_vector3 * v1**

A pointer to the destination vector.

**float f1**

The first vector element (x axis component).

**float f2**

The second vector element (y axis component).

**float f3**

The third vector element (z axis component).

*Example:*

```
br_vector3 *va;
...
BrVector3Set(va,1.f,-1.f,1.5f);
```

## BrVector3SetInt()

*Description:* Set a vector from a triple of standard C integers.

*Declaration:* **void BrVector3SetInt(br_vector3* v1, int i1, int i2, int i3)**

*Arguments:* **br_vector3 * v1**

A pointer to the destination vector.

**int i1**

The first vector element (x axis component).

**int i2**

The second vector element (y axis component).

**int i3**

The third vector element (z axis component).

*Example:*

```
br_vector3 *va;
...
BrVector3Set(va,1,-1,2);
```

# br_vector4

## The Structure

This is the four ordinate vector structure, typically used to hold homogenous 3D co-ordinates. Functions are provided to allow it be used as though it were an integral type.

### *The* ***typedef***

(See *vector.h* for precise declaration and ordering)

| | | |
|---|---|---|
| **br_scalar** | **v[4]** | ***Ordinates (0=x, 1=y, 2=z, 3=w)*** |

### *Related Functions*

See **BrMatrix34TApply()** 195, **BrMatrix4[T]Apply[V|P]()** 219-222.

### *Related Structures*

See **br_matrix4** *217*.

## Members

### br_scalar v[4]

First, second, third and fourth ordinate. Conventionally, the first ordinate is the x-axis component, the second, the y axis component, the third, the z axis component, and the fourth, the divisor.

Remember that BRender has a right handed co-ordinate system and so, with the x axis positive to the right, and the y axis positive upwards, the z axis is therefore positive toward you (typically, the z axis points out of the screen).

## Arithmetic

---

# BrVector4Dot()

*Description:* Calculate the dot product of two vectors. Equivalent to the expression:

$$\mathbf{v}_1 \cdot \mathbf{v}_2$$

*Declaration:*
```
br_scalar BrVector4Dot(const br_vector4* v1,
const br_vector4* v2)
```

*Arguments:* **const br_vector4 * v1**

Pointer to left hand vector (may be same as source).

**const br_vector4 * v2**

Pointers to right hand vector.

*Result:* **br_scalar**

Returns the dot product of the two source vectors. Equivalent to:

$$(x_1 \;\; y_1 \;\; z_1 \;\; w_1) \cdot (x_2 \;\; y_2 \;\; z_2 \;\; w_2) \equiv x_1x_2 + y_1y_2 + z_1z_2 + w_1w_2$$

---

## Copy/Assign

Although copy by structure assignment currently works, use **BrVector4Copy()** 377 to ensure compatibility.

---

# BrVector4Copy()

*Description:* Copy a vector. Equivalent to the expression:

$$\mathbf{v}_1 \Leftarrow \mathbf{v}_2$$

*Declaration:* **void BrVector4Copy(br_vector4* v1, const br_vector4* v2)**

*Arguments:* **br_vector4 * v1**

A pointer to the destination vector (may be same as source).

**const br_vector4 * v2**

A pointer to the source vector.

---

## Referencing & Lifetime

This structure may be freely referenced, though take care if there is potential to supply the same vector as more than one argument to the same function.

## Initialisation

The following macro may be used as a static initialiser.

**BR_VECTOR4(a,b,c,d)**

Macro expands to
`{BR_SCALAR(a),BR_SCALAR(b),BR_SCALAR(c),BR_SCALAR(d)}`.

All other initialisation should use member-wise initialisation or the function **BrVector4Copy()** 377.

# br_vertex

## Overview

The vertex data structure, describing a single vertex in a model.

### *The* ***typedef***

(See *model.h* for precise declaration and ordering)

#### Position

| | | |
|---|---|---|
| **br_vector3** | **p** | ***Co-ordinates of a point in model space*** |

#### Texture and Lighting

| | | |
|---|---|---|
| **br_vector2** | **map** | ***Texture co-ordinates of this vertex*** |
| **br_uint_8** | **index** | ***Colour index for pre-lit models*** |
| **br_uint_8** | **red, grn, blu** | ***True colour values for pre-lit model*** |

### *Related Functions*

See **BrModelApplyMap()** 235 and **BrModelFitMap()** 238 to set texture co-ordinates. See **BrSceneModelLight()** 238 to set pre-lit values.

### *Related Structures*

See **br_model** *232* for the structure in which vertices are used.

See **br_material** *154* for details of how pre-lighting information is used.

## Members

### *Position*

#### br_vector3 p

The co-ordinates of a point (in the model's co-ordinate space) representing the vertex of a group of faces (typically triangular).

For example, a cube has eight vertices. If it had unit side and was centred about (0,0,0), it would have vertices (0.5,0.5,0.5), (0.5,0.5,-0.5), etc. Faces are polygons (typically triangles) described in terms of a number of vertex indices.

br_vertex

*Texture and Lighting*

### br_vector2 map

The 2D co-ordinates at which this vertex appears in an infinitely* tiled texture map.

### br_uint_8 index

Pre-computed lighting index at this vertex. See **br_material**154 for how this relates to pre-lit materials.

### br_uint_8 red, grn, blu

Pre-computed light level at this vertex (in terms of colour intensities†). See **br_material**154 for how this relates to pre-lit materials.

## Copy/Assign

Do not use structure assign. Use member-wise copy only.

## Access & Maintenance

Members may be freely accessed. Models referencing vertices that are changed should be updated using **BrModelUpdate()**241 before rendering. There are private members of vertices that are modified during **BrModelUpdate()**241.

## Referencing & Lifetime

Be careful of referencing vertices especially ones allocated by **BrModelAllocate()**243, they are liable to be moved around during **BrModelUpdate()**241, say. Vertices are generally only allocated as arrays completely describing a model. Always access using indexing from the model's vertices member.

## Initialisation

Use memset(,0,sizeof(br_vertex)) and set members as appropriate. Updating of models referencing initialised vertices will be needed before there are involved in rendering.

## Construction & Destruction

Vertices may be constructed conventionally, but BR_MODF_KEEP_ORIGINAL must be specified in any model that refers to them. Otherwise use **BrModelAllocate()**243.

---

* Subject of course to limits of **br_scalar** representation.

† Not all platforms support coloured lights.

## Supplementary

When constructed by **BrModelAllocate()**$_{243}$ vertices are allocated from the "VERTICES" memory class. It is probably better to organise any enumeration around models (see **br_model**$_{232}$).

## Import & Export

Vertices are included with model definitions. See **br_model**$_{232}$ for details of import/export functions.

# brfile_advance_cbfn

## The Call-Back Function

This type defines a stream advance function, primarily intended for the `advance` member of the **`br_filesystem`**[128] structure.

### *The* ***`typedef`***

(See *`brfile.h`* for a precise declaration)

**`void`** **`brfile_advance_cbfn(br_size_t,void*)`** ***Advance***

## Specification

---

## CBFnFileAdvance()

| | |
|---|---|
| *Description:* | An application defined call-back function advancing a file pointer a number of bytes through a binary stream. |
| *Declaration:* | **`void BR_CALLBACK CBFnFileAdvance(br_size_t count, void* f)`** |
| *Arguments:* | **`br_size_t count`** |
| | Number of bytes to advance. |
| | **`void * f`** |
| | Valid file handle - as returned by ***`CBFnFileOpenRead()`***[387] and ***`CBFnFileOpenWrite()`***[389]. |
| *Preconditions:* | BRender has completed initialisation. BRender is the only direct caller of this function. |
| *Effects:* | Advance file position by `count` bytes. |
| *Example:* | See *`stdfile.c`* for examples of filing system functions. |

---

# brfile_attributes_cbfn

## The Call-Back Function

This type defines a filing system capabilities enquiry function, primarily intended for the `attributes` member of the **`br_filesystem`**$_{128}$ structure.

### *The* ***`typedef`***

(See *`brfile.h`* for a precise declaration)

**`br_uint_32`** **`brfile_attributes_cbfn(void)`** ***Get attributes of filing system***

## Specification

---

## CBFnFileAttributes()

*Description:* An application defined call-back function returning capabilities of the filing system.

*Declaration:* **`br_uint_32 BR_CALLBACK CBFnFileAttributes(void)`**

*Preconditions:* BRender has completed initialisation. BRender is the only direct caller of this function.

*Effects:* None.

*Result:* **`br_uint_32`**

The attributes of the filing system. Its capabilities as defined by a combination of the following flag values:

| Flag | Attribute |
|---|---|
| `BR_FS_ATTR_READABLE` | Filing system can read files |
| `BR_FS_ATTR_WRITEABLE` | Filing system can write files |
| `BR_FS_ATTR_HAS_TEXT` | Filing system can interpret ASCII text files |
| `BR_FS_ATTR_HAS_BINARY` | Filing system can support binary files, i.e. maintains integrity of streams of any combination of bytes (8 bit). |
| `BR_FS_ATTR_HAS_ADVANCE` | Filing system can directly skip bytes |

*Example:* See *`stdfile.c`* for examples of filing system functions.

---

# brfile_close_cbfn

## The Call-Back Function

This type defines a file close function, primarily intended for the `close` member of the **`br_filesystem`**[128] structure.

### *The* ***`typedef`***

(See *`brfile.h`* for a precise declaration)

**`void`** **`brfile_close_cbfn(void*)`** ***Close file***

## Specification

---

## CBFnFileClose()

| | |
|---|---|
| *Description:* | An application defined call-back function closing a previously opened file. |
| *Declaration:* | **`void BR_CALLBACK CBFnFileClose(void* f)`** |
| *Arguments:* | **`void * f`** |
| | Valid file handle - as returned by ***`CBFnFileOpenRead()`***[387] and ***`CBFnFileOpenWrite()`***[389]. |
| *Preconditions:* | BRender has completed initialisation. BRender is the only direct caller of this function. |
| *Effects:* | Close file. |
| *Example:* | See *`stdfile.c`* for examples of filing system functions. |

---

# brfile_eof_cbfn

## The Call-Back Function

This type defines an end of file test function, primarily intended for the `eof` member of the **br_filesystem**[128] structure.

### *The* ***typedef***

(See *brfile.h* for a precise declaration)

```
int                     brfile_eof_cbfn(const void *)
```
*Test for end of file*

## Specification

---

## CBFnFileEOF()

*Description:* An application defined call-back function testing a file pointer for end of file.

*Declaration:* **int BR_CALLBACK CBFnFileEOF(const void* f)**

*Arguments:* **const void * f**

Valid file handle - as returned by ***CBFnFileOpenRead()***[387] and ***CBFnFileOpenWrite()***[389].

*Preconditions:* BRender has completed initialisation. BRender is the only direct caller of this function.

*Effects:* None.

*Result:* **int**

Returns a non-zero value after the first read operation that attempts to read past the end of the file. It returns 0 if the current position is not end of file.

*Example:* See *stdfile.c* for examples of filing system functions.

---

# brfile_getchr_cbfn

## The Call-Back Function

This type defines a get character function, primarily intended for the `getchr` member of the **br_filesystem**[128] structure.

### *The* ***typedef***

(See *brfile.h* for a precise declaration)

```
int                     brfile_getchr_cbfn(void *)
```
*Get character*

## Specification

---

## CBFnFileGetChr()

*Description:* An application defined call-back function reading a character from a file.

*Declaration:* **int BR_CALLBACK CBFnFileGetChr(void* f)**

*Arguments:* **void * f**

Valid file handle - as returned by ***CBFnFileOpenRead()***[387].

*Preconditions:* BRender has completed initialisation. BRender is the only direct caller of this function.

*Effects:* If the file position is at the end of the file, the file enters the end-of-file state, otherwise a character is read and the file position is advanced.

*Result:* **int**

The character read from the file is returned (as though the character had been cast as `(int)(unsigned char)`). If a character could not be read because the file position was at the end of the file, `BR_EOF` is returned.

*Example:* See *stdfile.c* for examples of filing system functions.

---

# brfile_getline_cbfn

## The Call-Back Function

This type defines a get line function, primarily intended for the `getline` member of the **br_filesystem**$_{128}$ structure.

### *The* ***typedef***

(See *brfile.h* for a precise declaration)

**br_size_t** **brfile_getline_cbfn(char*,br_size_t,void*)** ***Getline***

## Specification

---

## CBFnFileGetLine()

*Description:* An application defined call-back function reading a line of text (excluding terminators) from a file.

*Declaration:* **br_size_t BR_CALLBACK CBFnFileGetLine(char* buf, br_size_t buf_len, void* f)**

*Arguments:* **char * buf**

Buffer to hold text read.

**br_size_t buf_len**

Length of buffer (maximum number of characters to store - including `'\0'`).

**void * f**

Valid file handle - as returned by ***CBFnFileOpenRead()***$_{387}$.

*Preconditions:* BRender has completed initialisation. BRender is the only direct caller of this function.

*Effects:* Read characters into supplied buffer until `buf_len-1` characters have been read, end of line has been read, or end of file has been reached. If the last character read was `'\n'` it is removed from the buffer.

*Result:* **br_size_t**

The number of characters stored in the buffer is returned. If at the end of file upon entry, zero will be returned.

*Example:* See *stdfile.c* for examples of filing system functions.

---

# brfile_open_read_cbfn

## The Call-Back Function

This type defines a stream advance function, primarily intended for the `open_read` member of the **`br_filesystem`**$_{128}$ structure.

### *The* ***`typedef`***

(See *`brfile.h`* for a precise declaration)

```
void*                   brfile_open_read_cbfn(const char*, br_size_t,
                                              br_mode_test_cbfn*, int*)
```

## Specification

---

## CBFnFileOpenRead()

*Description:* An application defined call-back function to open a file for read access.

*Declaration:* **`void* BR_CALLBACK CBFnFileOpenRead(const char* name, br_size_t n_magics, br_mode_test_cbfn* mode_test, int* mode_result)`**

*Arguments:* **`const char * name`**

Name of file.

**`br_size_t n_magics`**

Number of characters required for `mode_test` to determine file type (less than or equal to `BR_MAX_FILE_MAGICS`).

**`br_mode_test_cbfn mode_test`**

Call-back function that can be used to determine file type given the first `n_magics` characters of a file. Will not be used if `NULL`.

**`int * mode_result`**

If this argument is non-`NULL`, the file type (if it could be determined) will be stored at the address pointed to.

*Preconditions:* BRender has completed initialisation. BRender is the only direct caller of this function.

*Effects:* Searches for a file called `name`, if no path is specified with the file, looks in the current directory, if not found tries, in order, the directories listed in BRENDER_PATH (if defined). Having found the file, use `mode_test` (if supplied) to find out if the file is text, binary or unknown. Store the result through `mode_result` (if non-`NULL`). Obtain a handle to the file.

*Result:* **`void *`**

Return a file handle or `NULL` if the file could not be opened.

*Remarks:* Text mode files are primarily used for debugging but can be useful to allow hand editing of input data.

*Example:* See `stdfile.c` for examples of filing system functions.

---

# brfile_open_write_cbfn

## The Call-Back Function

This type defines a function to open a file for writing, primarily intended for the `open_write` member of the **`br_filesystem`**$_{128}$ structure.

### *The* ***`typedef`***

(See *`brfile.h`* for a precise declaration)

**`void *`** **`brfile_open_write_cbfn(const char*, int)`** ***Open file for writing***

## Specification

---

## CBFnFileOpenWrite()

*Description:* Open a file for writing, overwriting any existing file of the same name.

*Declaration:* **`void* BR_CALLBACK CBnFileOpenWrite(const char* name, int mode)`**

*Arguments:* **`const char * name`**

Name to open file as.

**`int mode`**

Mode in which to open file (`BR_FS_MODE_TEXT` or `BR_FS_MODE_BINARY`).

*Preconditions:* BRender has completed initialisation. BRender is the only direct caller of this function.

*Effects:* Overwrite or create file using specified `name` and `mode`.

*Result:* **`void *`**

Return a file handle or `NULL` if the file could not be opened.

*Example:* See *`stdfile.c`* for examples of filing system functions.

---

# brfile_putchr_cbfn

## The Call-Back Function

This type defines function to write a character to a file, primarily intended for the `putchr` member of the **`br_filesystem`**$_{128}$ structure.

### *The* ***`typedef`***

(See *`brfile.h`* for a precise declaration)

```
void                    brfile_putchr_cbfn(int, void *)
```
*Write character to stream*

## Specification

---

## CBFnFilePutChr()

| | |
|---|---|
| *Description:* | An application defined call-back function writing a single character to file. |
| *Declaration:* | **`void BR_CALLBACK CBFnFilePutChr(int c, void* f)`** |
| *Arguments:* | **`int c`** |
| | Character to write. |
| | **`void * f`** |
| | Valid file handle - as returned by ***`CBFnFileOpenWrite()`***$_{389}$. |
| *Preconditions:* | BRender has completed initialisation. BRender is the only direct caller of this function. |
| *Effects:* | Write the character to the file. |
| *Example:* | See *`stdfile.c`* for examples of filing system functions. |

---

# brfile_putline_cbfn

## The Call-Back Function

This type defines a function to write a line of text to a file, primarily intended for the `putline` member of the **br_filesystem**[128] structure.

### *The* ***typedef***

(See *brfile.h* for a precise declaration)

**void** **brfile_putline_cbfn(const char*, void*)** ***Write a line of text***

## Specification

---

## CBFnFilePutLine()

*Description:* An application defined call-back function writing a line of text to a file, followed by writing the new-line character (`'\n'`).

*Declaration:* **void BR_CALLBACK CBFnFilePutLine(const char* buf, void* f)**

*Arguments:* **const char * buf**

Pointer to zero terminated string containing line of text to be written.

**void * f**

Valid file handle - as returned by ***CBFnFileOpenWrite()***[389].

*Preconditions:* BRender has completed initialisation. BRender is the only direct caller of this function.

*Effects:* Write the string to the file. Write the new-line character to the file (`'\n'`).

*Example:* See *stdfile.c* for examples of filing system functions.

---

# brfile_read_cbfn

## The Call-Back Function

This type defines a function to read a block from a file, primarily intended for the `read` member of the **br_filesystem**$_{128}$ structure.

### *The* ***typedef***

(See *brfile.h* for a precise declaration)

**br_size_t brfile_read_cbfn(void*,br_size_t,unsigned,void*)** ***Read block***

## Specification

---

## CBFnFileRead()

*Description:* An application defined call-back function reading a block from a file.

*Declaration:* **br_size_t BR_CALLBACK CBFnFileRead(void* buf, br_size_t size, unsigned int nelems, void* f)**

*Arguments:* **void * buf**

Buffer to receive block.

**br_size_t size**

Size of each element in block.

**unsigned int nelems**

Maximum number of elements to read.

**void * f**

Valid file handle - as returned by ***CBFnFileOpenRead()***$_{387}$.

*Preconditions:* BRender has completed initialisation. BRender is the only direct caller of this function.

*Effects:* Read up to `nelems` elements of `size` bytes from the file `f` and store them in `buf`.

*Result:* **br_size_t**

Return the number of complete elements read, which may be less than `nelems` if the end of file is encountered before all the elements could be read.

*Example:* See *stdfile.c* for examples of filing system functions.

---

# brfile_write_cbfn

## The Call-Back Function

This type defines a function to write a block to a file, primarily intended for the `write` member of the **br_filesystem**$_{128}$ structure.

### *The* ***typedef***

(See *brfile.h* for a precise declaration)

**br_size_t** **brfile_write_cbfn(const void*,br_size_t,unsigned,void*)** ***Write block***

## Specification

---

## CBFnFileWrite()

*Description:* An application defined call-back function writing a block to a file.

*Declaration:* **br_size_t BR_CALLBACK CBFnFileWrite(const void* buf, br_size_t size, unsigned int nelems, void* f)**

*Arguments:* **const void * buf**

Buffer containing block to be written.

**br_size_t size**

Size of each element in block.

**unsigned int nelems**

Maximum number of elements to write.

**void * f**

Valid file handle - as returned by ***CBFnFileOpenWrite()***$_{389}$.

*Preconditions:* BRender has completed initialisation. BRender is the only direct caller of this function.

*Effects:* Write up to `nelems` elements from `buf` to the file `f`.

*Result:* **br_size_t**

Return the number of complete elements written, which may be less than `nelems` if an error occurs (such as running out of file space).

*Example:* See *stdfile.c* for examples of filing system functions.

---

# brmem_allocate_cbfn

## The Call-Back Function

This type defines a memory allocation function, primarily intended for the `allocate` member of the **`br_allocator`**$_{102}$ structure.

### *The* ***`typedef`***

(See *`brmem.h`* for a precise declaration)

**`void *`** **`brmem_allocate_cbfn(br_size_t,br_uint_8)`** ***Allocator***

## Specification

---

## CBFnMemAllocate()

*Description:* An application defined call-back function returning a pointer to newly allocated memory of specified size and type.

*Declaration:* **`void* BR_CALLBACK CBFnMemAllocate(br_size_t size, br_uint_8 type)`**

*Arguments:* **`br_size_t size`**

Size in bytes of memory block required. More bytes may be allocated, but the caller will only use the first `size` bytes. Zero is a valid value.

**`br_uint_8 type`**

Class (type ID) of memory required. Although it is possible for the class to be ignored, it may be used for diagnostic purposes, or could be used to provide a more efficient allocation scheme. See Memory Classes for a description of possible values.

*Preconditions:* The diagnostic handler has been setup (**`BrDiagHandlerSet()`**$_{119}$). BRender has not necessarily completed initialisation. This may not be the first allocator to have been called. BRender is the only direct caller of this function.

*Effects:* Obtain a fixed, contiguous, persistent, readable, writable, non-volatile area of memory of at least the size specified (even if zero), that will remain so until a corresponding call of ***`CBFnMemFree()`***$_{396}$ has occurred.

*Result:* **`void *`**

The address of a block of memory of the required size. If the request cannot be satisfied, return `NULL`.

No assumption can be made concerning any relation between the address returned by this call and any previous or subsequent call.

*Remarks:* Although BRender's default diagnostic handling system does not use the memory handler, if you supply a diagnostic handler that does, you may have to handle memory allocation failures directly.

The value returned by ***CBFnMemInquire()*** [397] does not guarantee the subsequent success or failure of ***CBFnMemAllocate()*** [394].

*Example:* See `stdmem.c` for examples of memory allocation functions.

*See Also:* ***CBFnMemFree()*** [396], ***CBFnMemInquire()*** [397].

---

# brmem_free_cbfn

## The Call-Back Function

This type defines a memory deallocation function, primarily intended for the `free` member of the **`br_allocator`**[102] structure.

### *The* ***`typedef`***

(See *`brmem.h`* for a precise declaration)

```
void                    brmem_free_cbfn(void *)Deallocator
```

## Specification

---

## CBFnMemFree()

| | |
|---|---|
| *Description:* | An application defined call-back function that will reclaim memory previously allocated by ***`CBFnMemAllocate()`***[394]. |
| *Declaration:* | **`void BR_CALLBACK CBFnMemFree(void* block)`** |
| *Arguments:* | **`void * block`** |
| | Address of memory no longer required, previously returned by the corresponding ***`CBFnMemAllocate()`***[394] (with no intervening ***`CBFnMemFree()`***[396]). `NULL` is ignored. |
| *Preconditions:* | The diagnostic handler has been setup (**`BrDiagHandlerSet()`**[119]). BRender has not necessarily completed initialisation. Other allocation handlers may have already been used. BRender is the only direct caller of this function. The application has disposed of all references to the memory (obtained using the supplied address). |
| *Effects:* | Release supplied memory (presumably making it available for re-use). |
| | If the supplied address is invalid (not recognised as previously allocated by the corresponding ***`CBFnMemAllocate()`***[394]), consider this a failure condition and handle appropriately. Note that the BRender diagnostic handling system may not be available, or even capable of handling this condition. |
| *Remarks:* | The memory freed will not necessarily increase the value returned by ***`CBFnMemInquire()`***[397], though it is of course a hopeful effect. |
| | The application should immediately set to `NULL` all pointers referencing the memory freed, (at least in a debug version). |
| *Example:* | See *`stdmem.c`* for examples of memory deallocation functions. |
| *See Also:* | ***`CBFnMemAllocate()`***[394], ***`CBFnMemInquire()`***[397]. |

---

# brmem_inquire_cbfn

## The Call-Back Function

This type defines a memory inquiry function, primarily intended for the `inquire` member of the **`br_allocator`**$_{102}$ structure.

### *The* ***typedef***

(See *brmem.h* for precise declaration)

**`br_size_t`** **`brmem_inquire_cbfn(br_uint_8)`** ***Inquiry function***

## Specification

---

## CBFnMemInquire()

*Description:* An application defined call-back function providing details of memory availability.

*Declaration:* **`br_size_t BR_CALLBACK CBFnMemInquire(br_uint_8 type)`**

*Arguments:* **`br_uint_8 type`**

Class of memory for which information is required. See Memory Classes for a description of possible values.

*Preconditions:* The diagnostic handler has been setup (**`BrDiagHandlerSet()`**$_{119}$). BRender has not necessarily completed initialisation. Other allocation handlers may have already been used. BRender is the only direct caller of this function.

*Effects:* Calculate an estimate of available memory for a particular memory class. This should be the total number of free bytes, irrespective of fragmentation or likely block-header overheads. If the class memory space would be extended, then the estimate should be of the potential maximum size available.

*Result:* **`br_size_t`**

The total number of bytes remaining unallocated in the specified memory class.

*Remarks:* This function is only intended to provide an estimate of available memory. It does not guarantee that a particular allocation would succeed (or fail). However, it could be used as an indication of likely success or failure.

There is no relation defined between the values returned for each class. In some allocation schemes, each class may have a separate and fixed amount of memory set aside, whereas in other schemes, all classes may be sharing all memory.

There is no relation defined between the true amount of memory (whether virtual or physical) in a system, and the value returned by this function.

*Example:* See `stdmem.c` for examples of memory inquiry functions.

*See Also:* ***CBFnMemAllocate()*** 394, ***CBFnMemFree()*** 396.

---

# Indices 5

# Macro Index

# Macro Index



## D



## O



## P



## V

# Macro Index

# Function Index

# Function Index

# Function Index

# General Index

# General Index



## B

## D

## E

# General Index

# General Index

## N

# General Index

## Q



## R

# General Index



## S

# General Index

# Notes

# Notes

# Notes

# Notes

# BRender API Quick Reference

## Functions

# BRender API Quick Reference

# BRender API Quick Reference

# BRender API Quick Reference

# BRender API Quick Reference

## Example Call-Back Functions

# BRender API Quick Reference