

# The RayBooster Ray Tracing Acceleration Library Programming Interface 2.1

18 July 2008



## Table of Content

1 Introduction .....	4
1.1 What is RayBooster?.....	4
1.2 Requirements.....	4
1.3 Conventions .....	4
1.3.1 Syntax .....	4
1.3.2 Return Values .....	5
2 Database Operations.....	6
2.1 The RayBooster Database.....	6
2.1.1 What's in the RayBooster Database?.....	6
2.1.2 Database Initialization.....	6
2.1.3 Database destruction.....	7
2.2 Surface Definition.....	7
2.2.1 Bloc Organization and IDs.....	7
2.2.2 Triangle Bloc creation.....	7
2.3 Other Surface Functions.....	8
2.3.1 Surface Visibility.....	8
2.3.2 Reading Surface Blocs .....	9
3 Ray Tracing.....	11
3.1 Rays and impacts.....	11
3.1.1 The RBrayimpact Structure.....	11
3.1.2 Ray initialization.....	12
3.2 Ray Tracing Operations.....	12
3.2.1 Find first impact.....	12
3.2.2 Find a serie of impacts.....	13
3.2.3 Single Intersection.....	14
4 Dynamic Ray Tracing .....	15
4.1 About Ray Tracing Acceleration Data.....	15
4.2 Simple changes to the Database.....	15
4.2.1 Adding a Surface Bloc.....	15
4.2.2 Removing a Surface Bloc.....	15
4.2.3 Changing a Surface Bloc.....	15
4.3 Introduction to Multiple Database Ray Tracing.....	16

4.4 Childhood.....	16
4.4.1 Principle.....	16
4.4.2 Setting or destroying a childhood link.....	17
4.4.3 Reading Children.....	17
4.4.4 Ray-Tracing on childhood databases.....	18
5 Instances.....	19
5.1 Principle.....	19
5.1.1 Create Instances.....	19
5.1.2 Update and destruction.....	20
5.1.3 Ray tracing on Instances.....	20
6 Miscellaneous Features .....	22
6.1 Database Options.....	22
6.1.1 Option description.....	22
6.1.2 Manipulating options.....	22
6.1.3 Advanced Option Management.....	23
6.1.3.1 The RBOption Structure.....	23
6.1.3.2 Defining Options.....	24
6.1.3.3 Accessing Option Structures.....	24
6.1.3.4 Manipulating Bloc Options.....	25
6.2 Multithreading.....	26
6.3 Allocations in the Database.....	26
6.4 Licensing.....	28
6.5 Outputs.....	28

## 1 Introduction

This document describes the RayBooster Ray Tracing Acceleration Library. We assume that the reader has notions of programming in C or C++ languages, as well as notions of Ray Tracing and geometrics. The reader can check [www.hpc-sa.com](http://www.hpc-sa.com) for a newer version of this document.

### 1.1 What is RayBooster?

RayBooster is a Ray Tracing Acceleration Library. It provides a Ray Tracing software (from now on referred as the Host Application) several functions that allow it to perform high performance intersection calculations.

A programmer shall use RayBooster in two distinct steps:

- **Database Operation:** during this step, the RayBooster main object, the Database, is built and the Host Application describes the scene geometry on which to perform Ray Tracing Operations.
- **Ray Tracing:** the Host Application sends RayBooster intersection queries, i.e. asks RayBooster to find intersections with the scene geometry along given rays.

### 1.2 Requirements

RayBooster is a C library.

Being essentially a mathematical library, RayBooster does not need any compatibility requirement such as MFC, OpenGL, DirectX or any other. It is written in standard ANSI C, and uses system specific functionalities only for synchronization, system queries or advanced assembly programming. Anyway these calls are transparent and shall not perturb the Host Application's functioning.

Additionally, RayBooster is available on the following platforms, in **32 and 64 bytes** flavours:

- **WIN 32 systems** (Windows 98, 2000, ME, XP, Vista, ...)
- **MAC OS X**
- **LINUX**
- **UNIX**

### 1.3 Conventions

#### 1.3.1 Syntax

The RayBooster Interface obeys to the following syntax:

- **Constants and precompiler directives:** RB\_CONSTANT\_NAME
- **Functions:** rb\_function\_name
- **Macros:** rb\_MACRO\_NAME
- **Structures:** RBstructure\_name

### 1.3.2 Return Values

Most RayBooster functions return an integer value. If the function had a normal behaviour, the return value is positive or zero. Otherwise, it returns an error code, which is a negative integer. Error code can be one of the following:

Error Code Value	Description
RB_RETURN_BAD_DATABASE	The input Database is not a valid RayBooster Database.
RB_RETURN_BAD_ID	ID in input is invalid. This applies to Bloc_id, Surface_id, RB_id.
RB_RETURN_NOT_AVAILABLE	The function or data is not available in this version.

## 2 Database Operations

### 2.1 The RayBooster Database

#### 2.1.1 What's in the RayBooster Database?

The first thing to do when using RayBooster is to create a RayBooster **Database** (referred from now on as the Database). A Database is an object that contains all information needed by RayBooster to compute intersections. This includes:

- **Access to host scene geometry:** RayBooster does not hold a copy of the Scene Geometry, that would be too memory consuming. Instead, it has ways to access the Host Scene Geometry, in the form of Callbacks.
- **State Variables:** several variables to set the behaviour of RayBooster. These variables are very few, as one of RayBooster's main assets is to find itself the best way to perform intersection calculations as fast as possible.
- **Acceleration Structure:** this structure is hidden from the Host Application, and contains some data built by RayBooster to speed-up intersection calculation.

#### 2.1.2 Database Initialization

A Database is created and initialized with the following function:

---

```
void *rb_init_database();
```

---

- **Return value:** a pointer to the newly created Database, NULL if an error occurred.
- 

#### Remarks:

- The main reason to have `rb_init_database` returning NULL is RayBooster was unable to check the licence. Please see 6.4 for additional information about licences.
- In the case were previous calls to `rb_init_database` succeeded, you may be experiencing an allocation error. You have to free some memory if possible, then retry the database creation.

**Advanced programming:** When a RayBooster function receives a Database pointer, it looks at the first word (4 or 8 bytes) it points to, as if they were a pointer. If this second pointer is different from the Database pointer, the function follows it to find the actual Database pointer. This process is repeated until the pointed value equals its address. As a consequence, a Programmer can manipulate its own database structure, if the first word of this structure is the RayBooster Database Pointer, all RayBooster functions can be called with the host structure address as database.

### 2.1.3 Database destruction

At the end of a RayTracing process, the Host can destroy the RayBooster Database, thus freeing all the memory allocated within the Database.

---

```
int rb_destroy_database(void *database);
```

---

- **Return value:** 0 or an error code.
- 

## 2.2 Surface Definition

### 2.2.1 Bloc Organization and IDs

Inside the RayBooster Database, surfaces are organized within homogeneous **Surface Blocs**. Each Surface Bloc contains surfaces of the same type. Surface Blocs can be added, destroyed or changed independently.

Surfaces can be of three different types: **Triangles**, **Instances** or **Client Defined**. Instances and Client Defined Surfaces are discussed in the next chapter.

A given Database can contain up to 16.384 Surface Blocs, one bloc can have from 1 to 268.435.456 ( $2^{28}$ ) surfaces, though the entire database can contain a maximum of  $2^{28}$  surfaces.

**Surface Blocs** are identified with a unique **Bloc ID**, the return value of the RayBooster function that created it.

A surface is identified by its **Bloc ID** and the surface index in the bloc: Surface ID in the range  $[0, n[$  where  $n$  is the number of surfaces in the bloc.

Additionally, RayBooster identifies a surface with a single id: **Rb ID**, which can be computed either by (Bloc ID + Surface ID) or by (Bloc ID | Surface ID).

#### Remark:

- If you do not need Surface Blocs, just organize all your surfaces in a single bloc.

### 2.2.2 Triangle Bloc creation

Host Application uses Triangle Bloc creation to provide RayBooster with information on a set of triangles.

---

```
int rb_add_triangle_bloc(nsurfaces, vertex_callback, cldata, database);
```

---

- **Return value:** the Bloc Id or an Error Code.
- **int nsurfaces [in]:** the number of triangles in the bloc.
- **int (\*vertex\_callback)(int, void\*, double\*, double\*, double\*) [in]:** the vertex access callback. This function has to be defined by the Host Application, and will be called by RayBooster each time it needs to access the vertices of a particular triangle.
- **void \*cldata [in]:** a client pointer used as an input value in the vertex\_callback function. If you do not need it, just use NULL.

- **void \*database [in]:** the RayBooster Database.

**vertex\_callback** is highly used by RayBooster, and is typically called several times for each surface during a Ray Tracing process. So a programmer should see that it is **optimized**.

Another rule is that **vertex\_callback** must have a **constant behaviour**: from the creation of the surface bloc until the end of a Ray Tracing process, the function must provide the same answers with the same call. If surfaces are changed by the Host Application, RayBooster must be warned through the **rb\_notify\_changed** function (see below).

**vertex\_callback** has the following prototype:

---

```
int vertex_callback(surface_id, cldata, p0, p1, p2);
```

---

- **Return value:** non 0 unless the triangle is to be definitively ignored.
  - **int surface\_id [in]:** the triangle index in the bloc, in the range [0 .. nsurfaces - 1].
  - **void \*cldata [in]:** the client data pointer defined above.
  - **double p0[3], p1[3], p2[3] [out]:** double precision arrays where the callback stores the vertices coordinates.
- 

## 2.3 Other Surface Functions

### 2.3.1 Surface Visibility

It is possible to change temporarily the surface visibility regarding ray tracing operations, i.e. some surfaces can be flagged “invisible” for all ray tracing functions. This can be useful to render partially a scene without actually removing surfaces. Invisibility interferes only with Ray Tracing functions and does not alter ray tracing acceleration data.

To do so, a **Visibility Callback** is attached to each bloc. When a ray tracing function encounters a surface, it first calls this Callback to see whether the surface is visible or not.

Such a callback is set with the function:

---

```
int rb_set_visibility_callback(bloc_id, visibility_callback, database);
```

---

- **Return value:** 0 or an Error Code.
  - **int bloc\_id [in]:** the Bloc to set the Callback to.
  - **int (\*visibility\_callback)(int, void\*) [in]:** the Visibility Callback, defined by the Host.
  - **void \*database [in]:** the RayBooster Database.
- 

#### Remark:

The Visibility Callback can be one of the following constants:

- **RB\_VISIBILITY\_CALLBACK\_TRUE** (default),
- **RB\_VISIBILITY\_CALLBACK\_FALSE**,

- An user defined function.

With the first all surfaces in the Bloc are visible, with the second, all surfaces are invisible.

Otherwise, the Host can set a general Visibility Callback that affects visibility for each surface in the Bloc. In that case, the Visibility Callback has the following prototype:

---

```
int visibility_callback(surface_id, cldata);
```

---

- **Return value:** if 0, the surface is invisible.
  - **int surface\_id [in]:** the surface index in the Bloc, in the range [0 .. nsurfaces – 1].
  - **void \*cldata [in]:** the client data pointer, from the bloc creation function call.
- 

**Remark:**

This callback can have a non constant behaviour.

### 2.3.2 Reading Surface Blocs

Surface Blocs can be accessed through the `rb_read_bloc` function. This function fills a structure according to the bloc data and type:

---

```
int rb_read_bloc(bloc_id, bloc_info, database);
```

---

- **Return Value:** 0 or an Error Code.
  - **int bloc\_id [in]:** the Bloc Id of the Surface Bloc to read.
  - **RBsurface\_bloc \*bloc\_info [out]:** a pointer on a `RBsurface_bloc` structure (see bellow) filled by the function.
  - **void \*database [in]:** the RayBooster Database.
- 

The `RBsurface_bloc` structure has the following fields:

---

**RBsurface\_bloc**

---

- **int id:** the Bloc Id.
- **int nb\_surfaces:** the number of surfaces in the bloc.
- **int (\*visibility\_callback)(int, void\*):** the visibility callback, from `rb_set_visibility_callback`
- **void \*cldata:** the client data pointer, from the bloc creation call.
- **RBsurface\_type surface\_type:** can be either `RB_TRIANGLE` or `RB_INSTANCE`.
- **int (\*vertex\_access)(int, void\*, double\*, double\*, double\*):** the vertex callback for Triangle Blocs. NULL for Instance Blocs.
- **int (\*matrix\_access)(int, void\*, double\*, double\*):** NULL for Triangle Blocs. The matrix access callback for Instance Blocs, from the instance bloc creation. See chapter 5 for additional

information on Instances.

- **void \*pattern\_database:** NULL for triangle blocs. For Instance Blocs, the Pattern Database from the instance bloc creation. See chapter 5 for for additional information on Instances.
-

## 3 Ray Tracing

### 3.1 Rays and impacts

#### 3.1.1 The RBrayimpact Structure

Ray tracing operations consist in finding impacts with surfaces from the scene along a ray, i.e. a line described by a point and a direction.

RayBooster uses the same data structure to describe both rays and impacts. This allows changing easily an impact into a ray, for example when computing reflections or transparencies.

The **ray impact** structure is described in the table below. For each structure field, description is given as a ray or as an impact. Default value refers to values set by the `rb_INITIALIZE_RAYIMPACT` macro.

Field name	Type	Ray description	Impact description	Default value
<b>position</b>	double[3]	Ray origin	Impact position	{0,0,0}
<b>direction</b>	double[3]	Ray direction	Undefined	{0,0,0}
<b>normal</b>	double[3]	Unused	Surface normal on impact	{0,0,0}
<b>param</b>	double[3]	Unused	Impact parametric coordinates	{0,0,0}
<b>min_distance</b>	double	Min distance from ray origin to impact	Undefined	0
<b>max_distance</b>	double	Max distance from ray origin to impact	Undefined	1e32
<b>precision</b>	double	Min distance between ray origin and impact, or between two consecutive impacts in <b>rb_get_impacts</b>	Undefined	1e-6
<b>distance</b>	double	Unused	Distance between ray origin and impact	0
<b>rb_id</b>	int	Surface on which the ray origin lies, or -1 if none	Impact surface	-1
<b>bloc_id</b>	int	Unused	Impact surface bloc	Undefined
<b>surface_id</b>	int	Unused	Impact surface index	Undefined

Field name	Type	Ray description	Impact description	Default value
<b>cldata</b>	void*	Unused	Impact surface bloc client pointer	Undefined
<b>database</b>	void*	Database of surface on which the ray origin lies	Impact database	NULL

**Remarks:**

- **direction** does not need to be normalized.
- For triangles, **param** is the set of barycentric coordinates {b0, b1, b2}. For a triangle with vertices V0, V1, V2, the impact position P is given by:  $P = b0.V0 + b1.V1 + b2.V2$
- **min\_distance**, **max\_distance**, **distance** and **precision** depend on the **direction** norm. If direction is normalized, these are actual distances. Otherwise, an impact position is given by the formula:  $impact.position = ray.position + impact.distance \cdot ray.direction$
- **rb\_id** and **database** are used by a ray to avoid self-intersections: this surface is then avoided when searching for impacts. If the ray origin does not lie on a surface, the value must be negative (-1).

**3.1.2 Ray initialization**

RayBooster provides a macro that fills the RayImpact structure with the default values:

---

```
rb_INITIALIZE_RAYIMPACT(rayimpact);
```

---

- **RBrayimpact \*rayimpact**: the address of the ray impact structure to initialize.
- 

**3.2 Ray Tracing Operations****3.2.1 Find first impact**

This function searches for the first impact within a scene.

---

```
int rb_get_first_impact(ray, impact, database);
```

---

- **Return value**: 1 if an impact was found, 0 otherwise, or an error code.
  - **RBrayimpact \*ray [in]**: description of the ray.
  - **RBrayimpact \*impact [out]**: the impact structure filled by the function if an impact was found. This can be a NULL pointer if impact geometry is not required.
  - **void \*database [in]**: the RayBooster Database.
-

**Remark:**

A significant speedup can be obtained when impact geometry is not needed, by using NULL instead of an RBrayimpact pointer. This can be useful when computing occlusion, for example in shadow computation.

**3.2.2 Find a serie of impacts**

This function computes several impacts along a ray.

---

```
int rb_get_impacts(ray, impacts, nimpacts, opacity_callback, cldata, database);
```

---

- **Return value:** the number of impacts found, in the range [0 .. nimpacts], or an error code.
  - **RBrayimpact \*ray [in]:** description of the ray.
  - **RBrayimpact impact[] [out]:** an array of impacts filled by the function.  
Cannot be a NULL pointer, and must not be smaller than nimpacts\*sizeof(RBrayimpact).
  - **int nimpacts [in]:** the maximum number of impacts.
  - **int (\*opacity\_callback)(ray, impact, nimpact, cldata, database) [in]:** an opacity callback, called by the function each time an impact is found. This callback can stop the search for further impacts. If NULL, no callback is called and the function stops at the end of the ray or when nimpacts were found. See below for a full description of this callback.
  - **void \*cldata [in]:** a client pointer transferred to the opacity callback.
  - **void \*database [in]:** the RayBooster Database.
- 

The **opacity callback** is called each time the ray finds an intersection. These calls are ordered according to the impact's **distance** field. The callback can alter any field of the impact structure except **distance**. The callback must not alter the **ray** structure.

The callback has the following prototype:

---

```
int opacity_callback(ray, impact, nimpact, cldata);
```

---

- **Return value:** if 0, **rb\_get\_impacts** stops searching for impacts after this one. If non zero, it continues.
  - **RBrayimpact \*ray [in]:** description of the ray.
  - **RBrayimpact \*impact [in|out]:** the impact that was just found. All fields can be modified by the function except the **distance** field.
  - **int nimpact [in]:** the impact number, in the range [0 .. nimpacts]. 0 means that this is the first impact found.
  - **void \*cldata [in|out]:** the client pointer from **rb\_get\_impacts**.
-

### 3.2.3 Single Intersection

The next function computes the intersection of a Ray with a given Surface in the Database.

---

```
int rb_single_intersection(ray, impact, database);
```

---

- **Return value:** 1 if the intersection exists, 0 otherwise, or an Error Code.
  - **RBrayimpact \*ray [in]:** description of the ray.
  - **RBrayimpact \*impact [in|out]:** the impact filled by the function. The **rb\_id** field is used in input to identify the surface to compute intersection with.
  - **void \*database [in]:** the RayBooster Database.
-

## 4 Dynamic Ray Tracing

### 4.1 About Ray Tracing Acceleration Data

When the first ray is cast upon a Database, RayBooster scans surfaces, builds one part of the acceleration data needed to enhance ray tracing speed. Another part of the acceleration data is built when needed, according to which rays are cast: some data is geography dependant, and is built only when ray pass through certain areas.

On the other hand, after a ray tracing simulation, the host can change the geometry, by removing or adding surface blocs, or simply changing them. In the process RayBooster removes a large part of its acceleration data that has to be rebuilt in the next ray-tracing step, and so on.

This acceleration data construction can be quite time consuming. This chapter discusses various approaches to manage acceleration data reconstruction.

### 4.2 Simple changes to the Database

All operations described in this chapter deeply affect the current Acceleration Data Structure. If rays have already been cast upon a Database, doing one of these operations makes RayBooster remove a large part of its Acceleration Data. When rays are cast after these operations, the Acceleration Data construction process starts again.

#### 4.2.1 Adding a Surface Bloc

A Surface Bloc can be added (via `rb_add_surface_bloc`) at any time in a Database, with Surface Bloc creation functions no matter if rays have been cast before.

#### 4.2.2 Removing a Surface Bloc

A Surface Bloc of any type is removed from the Database with the following function:

---

```
int rb_destroy_bloc(bloc_id, database);
```

---

- **Return value:** 0 or an Error Code.
  - **int bloc\_id [in]:** the Bloc ID, returned by the bloc creation function.
  - **void \*database [in]:** the RayBooster Database.
- 

#### 4.2.3 Changing a Surface Bloc

Another way to change the database is simply to change the behaviour of Bloc Callbacks, but in that case RayBooster has to be warned in order to re-perform its initialization processes.

A call to the next function notifies that one or all blocs have changed:

---

```
int rb_notify_changed(bloc_id, database);
```

---

- **Return value:** 0 or an Error Code.
- **int bloc\_id [in]:** the **Bloc ID** of the Surface Bloc that has changed, or a **negative value** to

update all blocs in the database.

- **void \*database [in]:** the RayBooster Database.
- 

### 4.3 Introduction to Multiple Database Ray Tracing

A common point in ray tracing is that the surfaces can evolve between two ray tracing simulations. In a typical use of RayBooster, this is done as we have seen in previous chapter. This causes RayBooster to perform some heavy initializations of its whole ray tracing acceleration structure, which can be quite time consuming. This duration can be roughly evaluated as the cost of 1 to 5 rays for each surface: in a 100.000 triangles scene, the initialization process takes the same duration as casting 100.000 to 500.000 extra rays. In some cases, it is possible to reduce this initialization process, by splitting the surfaces in several databases. At each beginning of the simulation, only the databases that actually need to be updated perform the initializations.

RayBooster provides two tools to save initialization time:

- **Childhood:** the principle is to split the surfaces in several databases. Ray-tracing operations see them as one, but database operations are performed independently.
- **Instances:** the host creates one or several independent Pattern Databases. The main database can then contain one or more instances (matrix transformations) of these patterns. Updating the main database does not make it necessary to update the Pattern Databases, i.e. instances can move into the main database with no extra initialization cost of the pattern Database.

## 4.4 Childhood

### 4.4.1 Principle

RayBooster provides a way to ray trace simultaneously on several Databases organized according to a parent / childhood tree structure. Childhood is a way to split a scene into several databases that will be raytraced as one, but can be managed separately. The point is that between two ray tracing simulations, one database can be changed without affecting the other, saving initialization time.

A typical use of Childhood can be useful when the host performs several ray tracing simulations on a partially evolving scene: the host can make one root database with all the surfaces that do not evolve between simulations, and another one with all the surfaces that evolve. Between two simulations, the host updates only the second database. When performing ray tracing, a childhood link between the 2 databases makes that computing a ray in one of the database also computes it in the other.

A database can have an unlimited number of children. These children can themselves have children. Childhood link is not reciprocal: 2 different databases can share the same child database.

#### 4.4.2 Setting or destroying a childhood link

The following function creates a childhood link between two databases:

---

```
int rb_add_child(database1, database2);
```

---

- **Return value:** 1, or an Error Code.
  - **void \*database1 [in]:** child database.
  - **void \*database2 [in]:** parent database.
- 

##### Remarks:

- Each Database can have any number of children.
- This process adds **database1** as a child of **database2**. **Database1** can simultaneously be the child of any other database. **Database2** can have any other child.

The next function removes a childhood link:

---

```
int rb_remove_child(database1, database2);
```

---

- **Return value:** 1, or an Error Code.
  - **void \*database1 [in]:** child database.
  - **void \*database2 [in]:** parent database.
- 

#### 4.4.3 Reading Children

The following function returns the index<sup>th</sup> child of a database:

---

```
void *rb_get_child(index, database);
```

---

- **Return value:** the child database pointer, or NULL.
  - **int index [in]:** index of the child. The first child has index 0.
  - **void \*database [in]:** parent database.
- 

##### Remark:

- The latest added child has **index 0**.

#### 4.4.4 Ray-Tracing on childhood databases

When calling `rb_get_impacts` on childhood databases, the **opacity callback** is called first for all impacts on the parent database, then on the children, last added child first.

So the host must take into account that **the opacity callback can be called for various impacts without respect of the impact distance order**. Moreover, the callback can be called for impacts that will not be kept in the final impact list: when ray-tracing the first child database, the process might find impacts and call the opacity callback for impacts that will maybe be invalidated when raytracing another child.

## 5 Instances

### 5.1 Principle

Instancing is a way to handle very large triangle sets that would be processed with difficulty in a classical RayBooster Process. Instances are to be used when the Host Application manages multiple instances of a complex surface object (from now on referred as the **Pattern**).

The point is to use an intermediate Database (the **Pattern Database**) to store the Pattern description, and then to create **instances** of this pattern in the main RayBooster Database. Instance geometry is a matrix transformation of the Pattern geometry. The main database can then contain different instances of the same pattern, i.e. the pattern can be replicated at different locations, with different orientations, or different sizes.

The **Pattern Database** is not a special database, it is created as any other database, can be raytraced independently, can have children and can even have instances of other **Pattern Databases**.

#### 5.1.1 Create Instances

To create instances of a Pattern Database, the host adds a special surface bloc into the database: an Instance Bloc. All instances in this bloc refer to the same Pattern Database. Such a bloc is created with the function:

---

```
int rb_add_instance_bloc(ninstances, matrix_callback, pattern_database,
                        cldata, database);
```

---

- **Return Value:** the **Bloc ID** or an Error Code.
  - **int ninstances [in]:** the number of instances in the bloc.
  - **int (\*matrix\_callback)(int, void\*, double\*, double\*) [in]:** the matrix access callback. This function has to be defined by the Host Application, and will be called by RayBooster each time it needs to access the matrix transformation of a particular instance.
  - **void \*pattern\_database [in]:** the Pattern database.
  - **void \*cldata [in]:** a client pointer used as an input value in the matrix\_callback function.
  - **void \*database [in]:** the RayBooster Database in which to create instances.
- 

Such a surface bloc can be managed as any other surface bloc: it can be destroyed, read, can use visibility callback and so on.

The matrix callback must be defined according to the following prototype:

---

```
int matrix_callback(surface_id, cldata, matrix, inv_matrix);
```

---

- **Return Value:** see bellow.
- **int surface\_id [in]:** the instance index in the bloc, between **0** and **ninstances-1**.
- **void \*cldata [in]:** the client data pointer defined above.

- **double matrix[16] [out]:** double precision array where the callback stores the matrix.
- **double inv\_matrix[16] [out]:** double precision array where the callback stores the inverse matrix.

The matrix callback is called only once for each instance at initialization. RayBooster needs both the direct and the inverse matrices, though the host can provide only 1, according to the return value:

- **If callback return value is 3 or greater,** RayBooster assumes that **both matrices** have been set by the callback, and does not check if the matrices are actually inverse.
- **If callback return value is 2,** RayBooster assumes that **only inv\_matrix** has been set by the callback, and computes the matrix itself.
- **If callback return value is 1,** RayBooster assumes that **only matrix** has been set by the callback, and then computes the inverse matrix itself.
- **If callback return value is 0,** RayBooster assumes that **matrix is identity**, and does not take into account nor matrix nor inv\_matrix.
- If the callback return value is **negative**, RayBooster definitively removes the instance.

If  $M$  is an instance Matrix, and  $P(x,y,z)$  a point in the Pattern Database,  $P'(x',y',z')$  the transformation of  $P$  in the main database is given by:

$$x' = x*M[0] + y*M[4] + z*M[8] + M[12]$$

$$y' = x*M[1] + y*M[5] + z*M[9] + M[13]$$

$$z' = x*M[2] + y*M[6] + z*M[10] + M[14]$$

### 5.1.2 Update and destruction

If the **Pattern Database** happens to change, it is not necessary to notify the change of an Instance Bloc that uses this Pattern Database. Changes on the Pattern Database do not affect in any way a Database that instantiates it. Note that if the bounding box of the Pattern Database grows bigger, the Instance Boxes may not be appropriate anymore. In that case, Instances Boxes have to be changed, and so does the instancing Database.

The destruction of an **Instance Bloc** or the destruction of a Database that contain an **Instance Bloc** does not destroy the **Pattern Database**. When manipulating Instances, the Host Application has to manage Pattern Databases destruction when necessary.

Additionally, the same **Pattern Database** can be instanced by different **Databases**.

### 5.1.3 Ray tracing on Instances

When an impact is found in an instance after a ray-tracing operation, the fields of the **RBrayimpact** structure **rb\_id**, **surface\_id**, **bloc\_id** give the ids of the surface in the Pattern Database. The field **database** is the Pattern Database, not the main Database.

In some cases the host might want to know the hierarchy of instances of the impact. In this case, the host uses the **hierarchy** and **hierarchy\_depth** fields of the impact structure :

- **hierarchy\_depth** is the number of hierarchy elements used to describe the impact. For a simple triangle impact, **hierarchy\_depth** is 0. For an impact on a triangle in an instance,

hierarchy\_depth is 1. And so on.

- **hierarchy:** this an array of RB\_MAX\_HIERARCHY\_DEPTH elements. The elements are of type **RBsurface\_descriptor**. This structure contains fields that describe an impact: **rb\_id**, **surface\_id**, **bloc\_id**, **cldata** (the client pointer of the surface bloc) and **database**. For example, if an impact describes a triangle in an instance, the main rb\_id, surface\_id, bloc\_id, cldata and database (in the RBrayimpact structure) refer to the pattern database of the instance. Then the first **RBsurface\_descriptor** structure refers to the instance id in the main database.

## 6 Miscellaneous Features

### 6.1 Database Options

#### 6.1.1 Option description

Several options can be set / read in a Database. An option is described by:

- **Option name:** a constant integer value. Name is used according to a constant of type **RB\_OPTION\_NAME**.
- **Type:** Options can be character, integer, float, double, etc.
- **Default Value:** value at Database creation.
- **Initial status:** some options are not activated in default mode: these options are then OFF as an initial status.
- **Changeable status:** if YES, the status (ON or OFF) of an option can be changed.
- **Read Only:** options than can not be set.

The following table gives the different options available in RayBooster:

Option Name	Type	Default	Initial Status	Changeable Status	Read Only
RB_OPTION_BOUNDING_BOX	double[6]	{0,0,0,0,0,0}	ON	-	YES
RB_OPTION_IMPOSED_BOUNDING_BOX	double[6]	{0,0,0,1,1,1}	OFF	YES	-

- **BOUNDING\_BOX:** the limits of surfaces in a scene. The first 3 values are minimal coordinates respectively in x, y, z. The last 3 values are maximum coordinates.
- **IMPOSED\_BOUNDING\_BOX:** if ON, RayBooster uses this data instead of the Bounding Box, no matter what the actual bounds of the scene are. Otherwise (default), RayBooster computes the Bounding Box according to the scene geometry.

#### 6.1.2 Manipulating options

An option is read with the following function:

---

```
int rb_get_database_option(option_name, value, database);
```

---

- **Return value:** the size of the option, in bytes, or an error code.
  - **int option\_name [in]:** the name of the option to read.
  - **void \*value [out]:** a pointer to a memory area to read the option.
  - **void \*database [in]:** the RayBooster Database.
- 

#### Remarks:

- If option is **OFF**, the function returns the error code **RB\_RETURN\_NON\_AVAILABLE**.

- It is possible to use a **NULL** pointer instead of **value**, in order to know if the function is **ON** or **OFF**.

An option is set with the following function:

---

```
int rb_set_database_option(option_name, value, database);
```

---

- **Return value:** the size of the option, in bytes, or an error code.
  - **int option\_name [in]:** the name of the option.
  - **void \*value [in]:** a pointer to a memory area that describes the new option value.
  - **void \*database [in]:** the RayBooster Database.
- 

#### Remarks:

- If option is **OFF**, the function turns it **ON** if the Status is changeable, and sets it.
- If option is Read Only, the function returns the error code **RB\_RETURN\_NON\_AVAILABLE**.

The following function changes the status (**ON** or **OFF**) of an option:

---

```
int rb_switch_database_option(option_name, status, database);
```

---

- **Return value:** the size of the option, in bytes, or an error code.
  - **int option\_name [in]:** the name of the option.
  - **int status [in]:** 0 to switch the option **OFF**, any other value to switch it **ON**.
  - **void \*database [in]:** the RayBooster Database.
- 

#### Remarks:

If option is not changeable, the function returns the error code **RB\_RETURN\_NON\_AVAILABLE**.

### 6.1.3 Advanced Option Management

RayBooster provides tools for the user to create its own options inside a database. There are two kinds of options:

- Database Options: just like **BOUNDING\_BOX**, the user can add its own options.
- Bloc Options: user can define options that are different for each Surface Bloc.

#### 6.1.3.1 The RBoption Structure

Options are described through the RBoption structure, which has the following fields:

- **char label[256]:** the option label. All options must have different labels; the host can not create an option whose name has already been used.
- **int status:** the option status, a logical sum of the following:

- **RB\_STATUS\_ON** if the option is to be initialized ON.
- **RB\_STATUS\_CHANGE** if the option status is to be changed
- **RB\_STATUS\_WRITE** if the option is not Read Only.
- **int size**: the size of the option value, in bytes.
- **void \*value**: a pointer to the option default value.

### 6.1.3.2 Defining Options

Options must be defined **before** the Database Creation, in order to be taken into account. Options defined after a Database Creation will be used only for Databases created after this point.

A **Database Option** is defined with the following function:

---

```
int rb_define_database_option(option) ;
```

---

- **Return value**: an integer value that can be used from now on as the **name** of the option, or an error code.
  - **RBoption \*option [in]**: a pointer to the option descriptor.
- 

A **Bloc Option** is defined with:

---

```
int rb_define_bloc_option(option) ;
```

---

- **Return value**: an integer value that can be used from now on as the **name** of the option, or an error code.
  - **RBoption \*option [in]**: a pointer to the option descriptor.
- 

#### Remarks:

- If an option with the same label is already defined, the function returns the error code **RB\_RETURN\_BAD\_VALUE**.
- The functions make their own copy of the **RBoption** structure, as well as of the **value** field, according to the **size** field. Changing the RBoption values after this call has no effect.

### 6.1.3.3 Accessing Option Structures

Accessing Option Structures can be convenient, for example to manage options automatically, or to change the default values.

The following functions fill the RBoption structure according to the Option characteristics:

---

```
int rb_get_database_option_descriptor(option_name, option) ;
```

---

- **Return value**: 1, or an error code.
  - **int option\_name [in]**: the option name.
  - **RBoption \*option [out]**: a pointer to the option descriptor.
-

---

```
int rb_get_bloc_option_descriptor(option_name, option);
```

---

- **Return value:** 1, or an error code.
  - **int option\_name [in]:** the option name.
  - **RBoption \*option [out]:** a pointer to the option descriptor.
- 

#### **Remarks:**

- If Option does not exist, the function returns the error code **RB\_RETURN\_BAD\_ID**. As the Option Names are integer values starting at 0, a convenient way to read all the options is to loop from 0 until the return value is an error code.
- After a call to one of these functions, the **value** field of the **RBoption** structure is the actual default value. Changing it will affect the default value of the options in all databases created after this point.

#### 6.1.3.4 Manipulating Bloc Options

Bloc options are used in a similar way as Database options, with the following functions:

---

```
int rb_get_bloc_option(bloc_id, option_name, value, database);
```

---

- **Return value:** the size of the option, in bytes, or an error code.
  - **int bloc\_id [in]:** the id of the bloc from which to read the option.
  - **int option\_name [in]:** the name of the option to read.
  - **void \*value [out]:** a pointer to a memory area to read the option.
  - **void \*database [in]:** the RayBooster Database.
- 

---

```
int rb_set_bloc_option(bloc_id, option_name, value, database);
```

---

- **Return value:** the size of the option, in bytes, or an error code.
  - **int bloc\_id [in]:** the id of the bloc where to set the option.
  - **int option\_name [in]:** the name of the option to read.
  - **void \*value [in]:** a pointer to a memory area that describes the new option value.
  - **void \*database [in]:** the RayBooster Database.
- 

---

```
int rb_switch_bloc_option(bloc_id, option_name, status, database);
```

---

- **Return value:** the size of the option, in bytes, or an error code.
- **int bloc\_id [in]:** the id of the bloc from which to read the option.
- **int option\_name [in]:** the name of the option to read.

- **int status [in]:** 0 to switch the option **OFF**, any other value to switch it **ON**.
  - **void \*database [in]:** the RayBooster Database.
- 

## 6.2 Multithreading

RayBooster is thread-safe: a process using RayBooster can create as many threads as it needs and use RayBooster from any number of threads without any additional synchronization.

RayBooster's internal synchronization allows the actual parallel execution of some functions, including Ray Tracing Operations: if several Ray Tracing Operations execute in different threads, they are independent from one another, and this produces an actual speedup on multiprocessor hardware.

On the other hand, other operations, including Database definition, or changes, are internally synchronized so that only one of these operations is achieved at a time. Parallel calls to such functions is legal, but no speedup can be achieved this way.

No error occurs when Ray Tracing Operation and other operations are achieved simultaneously, but due to the internal representation of acceleration data, and the hidden initialization process when computing the first ray after a Database change, this can lead to performance issues. Additionally, changing the database in parallel with raytracing computations should be avoided, as it will lead to non-predictible results. In a multithreaded application, the Host should better check that all Ray Tracing Operations have returned before doing alterations to the Database.

## 6.3 Allocations in the Database

RayBooster provides two utilities to perform allocation in the Database. A Host can use these function to allocate data that has the same lifetime as the Database or one of its Surface Blocs.

The first function allocates data in the Database. This data will be freed only at the Database Destruction.

---

```
void *rb_alloc(size, database);
```

---

- **Return value:** a pointer on allocated data, or NULL if error.
  - **size\_t size [in]:** the size to allocate, in bytes.
  - **void \*database [in]:** the RayBooster Database.
- 

The second Function allocates data in an existing Surface Bloc. This data will be freed at the Bloc destruction.

---

```
void *rb_alloc_bloc(size, bloc_id, database);
```

---

- **Return value:** a pointer on allocated data, or NULL if error.
  - **size\_t size [in]:** the size to allocate, in bytes.
  - **int bloc\_id [in]:** the Bloc ID where data is to be allocated.
  - **void \*database [in]:** the RayBooster Database.
-

To manage out of memory situations, the Host Application can register a callback that is called by RayBooster when a memory allocation fails. This callback is first called to give an opportunity to free some memory: when the callback returns, RayBooster try to allocate again. If this second try fails the callback is called again to let the Host Application terminate in a clean way; it is not supposed to return to RayBooster, otherwise RayBooster will force the process to exit.

---

```
void rbu_set_oom_callback (oom_callback) ;
```

---

- **void (\*oom\_callback)(int) [in]:** a callback function that RayBooster calls if a memory allocation fails. See below for a full description of this callback. Can be NULL to remove a previously defined callback.
- 

---

```
void oom_callback (int status) ;
```

---

- **int status [in]:** if 0, the Host Application has an opportunity to free some memory, and the callback can return to give RayBooster a second chance to allocate the memory. If 1, the Host Application should save all its data, inform the user that the process ran out of memory, and quit. If the callback returns to RayBooster when status is 1, RayBooster calls exit.
- 

To obtain information about the memory amount used by RayBooster at a given time, the Host Application can read the RBU\_MEMORY\_SIZE variable. This size is the total size allocated by RayBooster for all the databases, including user allocations made through calls to `rb_alloc` and `rb_alloc_bloc`.

---

```
size_t RBU_MEMORY_SIZE;
```

---

To obtain information about the memory amount used by a given database, the Host Application can call the following function:

---

```
int rb_print_tree_stats(database) ;
```

---

- **void \*database [in]:** the RayBooster Database.
- 

The most interesting information listed by this function are:

- **Global:** the total memory size for this database,
- **rb\_alloc:** the memory size allocated in this database by the Host Application using `rb_alloc`.

**Remark:**

The memory size allocated by the host using `rb_alloc_bloc` is not listed here. This size is accounted in the **Blocs** rubric. If needed, the Host Application has to compute this size.

## 6.4 Licensing

RayBooster is available according to two different licenses kinds:

- **Demo License:** Time limited license, some functionalities are not available (rb\_destroy\_database, rb\_destroy\_bloc).
- **Full License:** fully functional, no time limit.

The next function provides information about the type of RayBooster license currently in use:

---

```
int rb_get_license_status(date);
```

---

- **Return value: a negative value** if license cannot be granted, a positive value otherwise. If the value is positive, it is a bit mask that can be checked against various constant (using (value & constant) != 0) :
    - the **RB\_DEMO\_LICENCE** bit is set if the licence is a **demo** license,
    - the **RB\_FULL\_LICENCE** bit is set if the licence is a **full license**,
    - the **RB\_TIMEBOMBED\_LICENCE** bit is set if the licence has an expiration date,
    - the **RB\_HOST\_LOCKED\_LICENCE** bit is set if the licence is locked to a specific host,
    - the **RB\_USER\_LOCKED\_LICENCE** bit is set if the licence has user restrictions.
  - **int date[3] [out]:** in the case of a **RB\_TIMEBOMBED\_LICENCE** month, day, year of license expiration. Unchanged otherwise. If NULL, only the status is computed
- 

Some information can be printed about the licence:

---

```
void rb_dump_license_info (void);
```

---

- Prints information about the licence.
- 

## 6.5 Outputs

By default, any RayBooster output is made using the **printf** library call. The Host Application can change this by the mean of the following function:

---

```
rbu_set_print_function(print_function);
```

---

- **int (\*print\_function)(const char \*format, ...) [in]:** This function has to be defined by the Host Application, and will be called by RayBooster for all its outputs.
- 

### **Remark:**

- The type of print\_function is the same as the type of printf.
- To return to the printf behaviour, just call rbu\_set\_print\_function with NULL.