Raydium API Reference

CQFD Corp.

This document is the most up-to-date version. This is a work in progress:
there’s again some errors and wrong informations. Try, wait, or contribute ;)

Index of chapters
Index of all Raydium functions

This document is autogenerated, any change will be lost,
use RaydiumApiReferenceComments for any need.
Generated: 2005-09-21 20:54:19, for Raydium 0.680

1 Introduction to Raydium:

1.1 About:

Well, first of all, let me talk about Raydium goals: this project
aims to be simple, easy to use, portable, and quite fast.

Raydium is a C written abstract layer, on top of OpenGL,
GLU and GLUT: this means you can write an entire 3D
application without calling any OpenGL function.
Want to draw an object ? call the suitable Raydium function,
and all textures and vertices will be loaded, and your object drawn.
Want to make an explosion ? Same thing: call the right function.
Note that you can call OpenGL functions anyway, if necessary.

About portability, I can say a few things: Raydium was initially
planned for linux only, but with a "clean" (nearly ANSI) code,
and, in facts, we have been able to compile Raydium under Visual Studio (Windows)
and mingw with a very few modifications.
So you can expect a correct result on any system providing
OpenGL (at least 1.2), GLU, GLUT and a C compiler.

As we (Corp?.) needed a library for our own games, demos,
and... and things like that, and as I was interested by OpenGL,
I starts to write Raydium.

Raydium is perfect for outdoors spaces, integrating a landscape engine,
with suitable physic, supports dynamic lighting, fog, blending, water and
waves, terraforming, and more, but also provides everything for indoor,
with radiosity lightmaps for example.

Some other advanced features are available : physics, scripting,
live video, transparent networking, GUI, ...

This features list will probably grow up during Raydium developpement, see Raydium website: http://raydium.cqfd-corp.org/

You'll find, in this document, a list of many functions and possibilities of Raydium, but if it's your first view of Raydium, you should start with tutorials (http://raydium.yoopla.org/wiki/RaydiumTutorials) and packaged demo programs.

After this short introduction, let's talk about the API itself, starting with the main file (from the programmer's point of vue) of Raydium: common.c

1.2 Defines:

As mentioned above, the file common.c is quite interesting, for several reasons: first, as this file includes all others Raydium's files, you can have an overview of the whole project, just by looking at this.

It can also be used as a "quick help", since all variables are declared here, and not in the corresponding files. I mean, for example, that "`raydium_light_intensity...`" will be declared in common.c, not in light.c . There's many reasons for using such "style", but you must only retain that it is simpler for you :)

Ok, after this little disclaimer, we can have a look to the first part of our file.

After usual #include (nothing interesting here), we find some #defines.

generic limits

The first #define block determine limits of your application, and here you are the actual values for basic defines:

```
#define RAYDIUM_MAX_VERTICES 500000
#define RAYDIUM_MAX_TEXTURES 256
#define RAYDIUM_MAX_LIGHTS 8
#define RAYDIUM_MAX_NAME_LEN 255
#define RAYDIUM_MAX_OBJECTS 1024
```

- As you may expect, `MAX_VERTICES` defines the amount of memory you'll waste with vertex tables. These tables will contain all loaded objects, then remember each time you draw something (object), Raydium loads it (if not already done). Currently, there is no "delete" mechanism implemented (except by deleting all objects).

Let me give you a scale: with an Athlon XP1900+, GeForce 3, actual Raydium devel. version 0.31, with around 100 000 vertices, losts of options (sky, blending, 2 lights, 15 textures, ...), Raydium renders ~ 45 FPS. Beyond this, a very correct object uses less than 10 000 vertices. So 500 000 vertices, the actual default, is quite large. It’s also important to talk about memory: Linux is very efficient on this point, and allocates only "really used" memory. Under Linux, with the above scene, Raydium used about 20 MB (data only), instead of "much more" (~ 5x). I haven’t made any test about this under Windows, but we can expect worse results.

- There’s nothing really important to say about `MAX_TEXTURES`, since that doesn’t influence the amount of memory used. You are not limited to 8 bits values, but 256 seems very comfortable (and you must pay attention to the capacities of your 3D hardware !)

- The next define, `MAX_LIGHTS` is very important: OpenGL, for now (version 1.3 and lower), impose 8 lights at least, and all current hardware doesn’t manage more. If this situation is likely to evolve, we will move this #define to a variable, and will ask hardware for its capacities at initialization, but, for the moment, do not exceed 8.

- Next, `NAME_LEN`, limits the maximum length of strings (textures and objects names) used by Raydium. Default value should be perfect. (avoid higher values, since it could slow down name searches)

- `MAX_OBJECTS` use the same mechanism as `MAX_TEXTURES`, and addition with the fact that hardware is not concerned, it can be ignored.

Options and parameters

This is the next part of our #define section, I will not explain these constants here, but in respective sections, so you’ll have just you to remember they’re declared here.

1.3 Basic vars:

This section aims to describe each variable Raydium use, one by one. Some (most ?) of them are used internaly only, but you could need to access it. Moreover, you’ll better understand how Raydium works by looking at these variables.

Keyboard input

Following variables can be found:

`raydium_key_last` will always contains the last key (normal or special) pressed down. You’ll find a explanation about normal and special keys above.

`raydium_key[]` hosts all special keys state. Currently, you must use GLUT define's (Raydium aliases will come soon), limited to following keys:

- `GLUT_KEY_F1` to `GLUT_KEY_F12`
- `GLUT_KEY_LEFT`, `GLUT_KEY_RIGHT`, `GLUT_KEY_UP`, `GLUT_KEY_DOWN`
- `GLUT_KEY_PAGE_UP`, `GLUT_KEY_PAGE_DOWN`
- `GLUT_KEY_HOME`, `GLUT_KEY_END`, `GLUT_KEY_INSERT`

These are "special" keys: they have 2 states. released (0), and pressed (non zero). It means you can do something (move an object, turn on a light) UNTIL user stops to press the key. "Normal" keys have a different behavior: you can do something IF user press a key (exit from application if ESC is pressed, for example). You'll have no information about key's release.

A normal key is sent through `raydium_key_last`, a special one through `raydium_key[]` AND `raydium_key_last`.

You must see `raydium_key_last` as an "event", fired when the user press a key (ANY key: special or not). When a normal key is pressed, you'll get the ASCII value + 1000 assigned to `raydium_key_last`. (1027 for "ESC", for example)

Here is a method to use special keys:

```
if(raydium_key[GLUT_KEY_UP]) move_car();
```

Yes, it's easy. You can also use

```
if(raydium_key_last""==""GLUT_KEY_UP) explose();
```

for example, if you need to carry out a specific action.

It's ok for you ? use `raydium_key[]` to keep the car moving until user release UP key, or use `raydium_key_last` to explode the car when the user tries to start it :)

Mouse input

Easy.

You can get actual mouse position on the window (relative to window's position on screen, I mean) with `raydium_mouse_x` and `raydium_mouse_y` (GLuint), starting at (0,0) for upper left
(Warning: some GLUT implementations can give mouse position even when mouse is out of the window ! Check boundaries before using these values).

Raydium use: 1 for left button, 2 for right button, and 3 for middle button (0 for none) with `raydium_mouse_clic` for the last clic value. (generated one time per clic)

You can permanently get a button’s state, up (0) or down (non zero), using `raydium_mouse_button[x]`, where x is 0 for left button, 1 for right one, and 2 for middle button.

Textures

`raydium_texture_index` and `raydium_texture_current_main` (GLuint) are used internaly to determine repectively how many textures are loaded, wich is the current one.

The next variable, `raydium_texture_filter`, is very important. You can assign `RAYDIUM_TEXTURE_FILTER_NONE` (default), `RAYDIUM_TEXTURE_FILTER_BILINEAR` or `RAYDIUM_TEXTURE_FILTER_TRILINEAR` (recommended).

Using no texture filter can gives you higher framerate on old 3D hardware, but this is quite ugly.

You can activate bilinear filtering without any framerate impact on most recent video cards, and get a much more attractive rendering.

Trilinear filtering uses Bilinear filtering and MipMaps. A MipMaped? texture is a duplicated texture (3 times, with Raydium), but at different sizes. A 512x512 texture will generate, for example, a (smoothed) 256x256 texture, and a (smoothed) 128x128 one. Your video card will use these textures according to distance from POV (point of vue), reducing flickering effect.

This is the best filtering Raydium can use, for a great rendering quality. Good and recent 3D hardware can do trilinear filtering in a single pass, so it must be the default setting for your application.

About `raydium_texture_filter` itself: changing this variable will not modify the rendering, but the way to load textures. It means you can (for example) use trilinear only for landscape textures, and bilinear for others. It also means you must reload (erase) a texture to change it’s filter.

Note that Raydium will never use trilinear filter with blended (transparent) textures, for good reasons :)

Let’s talk quickly about next (internal) texture variables: `raydium_texture_blended[]` is a flag table, where each element is non zero for a blended (RGBA) texture, and 0 for an RGB one.

For Raydium, when a texture does not contain a "bitmap" (texture file, for example), it contains a plain color, and this color is stored in

`raydium_texture_rgb[][4]` (4 is for RGBA, values between 0 and 1).
You can load an rgb texture with "rgb" keyword. For example, instead of loading "red.tga", you can load "rgb(0.8,0.1,0.1)".

`raydium_texture_name[]` table simply contains texture filenames.

Last thing, `raydium_texture_to_replace`,
can be used to erase an already loaded texture.
Set the variable to n, and load a new texture: texture number "n" will be replaced in memory.

Projection

Raydium supports 2 types of projection: `RAYDIUM_PROJECTION_ORTHO` (orthographic) and `RAYDIUM_PROJECTION_PERSPECTIVE`.

First of all, let us point out what "projection" is. Using a "perspective" projection, closest objects will looks larger than the orthers. It is typically used in video games (since human eye runs like that), by opposition to orthographic projection, wich is mostly used by 3D modeling tools. The principle is simple, discover it by yourself :)

Raydium reads `raydium_projection` to determine wich method to use.
Each projection is configured with `raydium_projection_*` variables.
Some of these variables are used both by "perspective" and "orthographic" projections.

Here is what common.c says:

```
GLFLOAT RAYDIUM_PROJECTION_FOV; // PERSPECTIVE ONLY
GLFLOAT RAYDIUM_PROJECTION_NEAR; // PERSPECTIVE & ORTHO
GLFLOAT RAYDIUM_PROJECTION_FAR; // PERSPECTIVE & ORTHO
GLFLOAT RAYDIUM_PROJECTION_LEFT; // ORTHO ONLY
GLFLOAT RAYDIUM_PROJECTION_RIGHT; // ORTHO ONLY
GLFLOAT RAYDIUM_PROJECTION_BOTTOM; // ORTHO ONLY
GLFLOAT RAYDIUM_PROJECTION_TOP; // ORTHO ONLY
```

You’ve probably noticed that orthographic projection defines a "box" with your screen: near, far, left, right, bottom. Everything out ouf this box will never be displayed.

Perspective projection is based on FOV: Field Of Vision, given in degrees.
A common "human" fov is 60° , up to 90° without any noticeable deformation.
"near" and "far" are used for many things: Z-Buffer precision is affected, and clipping too: as with "orthographic", nothing will be displayed beyond "far", and fog, if enabled, will hide this "limit". This is right for "near", too, but without fog, obviously :)

Also remember that decreasing FOV will zoom in.

You must call `raydium_window_view_update()` after any modification on one (or more) of these variables (see "Window Managment" section for more information)

Frame size and color

`raydium_window_tx` and `raydium_window_ty` are read-only variables, providing you actual frame size.

`raydium_background_color[4]` is a RGBA table, and will be used for frame clearing, and fog color. You can change this variable, and call respective update functions (frame and fog), or simply use `raydium_background_color_change(GLfloat r, GLfloat g, GLfloat b, GLfloat a)`.

More informations in corresponding sections.

Vertices

Vertices data structure is distributed in 4 parts:

- `raydium_vertex_*` : these tables will simply contains vertices coordinates

- `raydium_vertex_normal_*` : vertices normals. Raydium will maintain two distinct normal tables, and this one will be used for calculations.

- `raydium_vertex_normal_visu_*` : the other normal table, used for lighting. Smoothing "visu" normals will provides a better rendering, and Raydium includes all necessary functions to automate this task.

- `raydium_vertex_texture_u`, `*raydium_vertex_texture_v`, `*raydium_vertex_texture` contains, for each vertex stored in the vertices data structure, u and v mapping information, and associated texture number. U and V are texture mapping coordinates.

Raydium can automatically generates some of these data (normals and uv coords, that is), Read "Vertices" section above for more information.

PLEASE, do not write directly in these tables, use dedicated functions.

Objects

Objects are loaded in Vertices stream, identified by a "start" and an "end" (`raydium_object_start[]` and `raydium_object_end[]`) in this stream. An index is incremented each time you load an object

(`GLuint raydium_object_index`). Filename is also stored in `raydium_object_name[][]`. Go to "Objects" section to know more.

Lights

First of all, `raydium_light_enabled_tag` contains 0 when light is disabled, non-zero otherwise. This is a read-only variable, so use suitable functions.

Currently, for Raydium, a light can have 3 states: on, off, or blinking. `raydium_light_internal_state[]` stores this.

Next comes all light’s features: position, color, intensity. You can modify directly these variables, and call update fonctions, if needed (not recommended).

Next, `raydium_light_blink_*` are used internaly for blinking lights, setting lowest, higher light intensity, and blinking speed. Do noy modify these variables, use suitable functions.

You should read the chapter dedicated to lights for more information.

Fog

Only one variable, here: `raydium_fog_enabled_tag`, switching from zero to non zero if fog is enabled. Do NOT use this variable to enable or disable fog, but suitable functions, this variable is just a tag.

Camera

Since many calls to camera functions are done during one frame, Raydium must track if any call to these functions was already done, using `raydium_frame_first_camera_pass` boolean.

`raydium_camera_pushed`, also used as a boolean, stores stack state. When you place your camera in the scene with Raydium, it pushes matrix on top of the stack, so you can modify it (the matrix), placing an object for example, an restore it quickly after, by popping matrix off.

2 Maths:

2.1 Little introduction to trigo.c:

This section is mostly designed for internal uses, but provides some usefull maths functions, mostly for trigonometrical uses.

2.2 GLfloat raydium_trigo_cos (GLfloat i):

Obvious (degrees)

2.3 GLfloat raydium_trigo_sin (GLfloat i):
Obvious (degrees)

2.4 GLfloat raydium_trigo_cos_inv (GLfloat i):
Obvious (degrees)

2.5 GLfloat raydium_trigo_sin_inv (GLfloat i):
Obvious (degrees)

2.6 raydium_trigo_abs(a) (macro):
Obvious

2.7 raydium_trigo_min(a,b) (macro):
Obvious

2.8 raydium_trigo_max(a,b) (macro):
Obvious

2.9 raydium_trigo_isfloat(a) (macro):
Test two cases : "Not a Number" and "Infinite"

2.10 void raydium_trigo_rotate (GLfloat * p, GLfloat rx, GLfloat ry, GLfloat rz, GLfloat * res):
Rotate p (GLfloat * 3) by (rx,ry,rx) angles (degrees).
Result is stored in res (GLfloat * 3)

2.11 void raydium_trigo_pos_to_matrix (GLfloat * pos, GLfloat * m):
Generates a ODE style matrix (16 Glfloat) from pos (GLfloat * 3)

2.12 void raydium_trigo_pos_get_modelview (GLfloat * res):
Stores the current OpenGL MODELVIEW matrix in res (16 GLfloat)

2.13 int raydium_trigo_pow2_next(int value):
Returns next power of two of `value`. Ugly.

3 Logging:

3.1 Introduction to log.c:
Raydium uses and provides his own logging system,
hidden behind a single function, as shown below.

3.2 void raydium_log (char *format, ...):
This function must be used like "printf", using a format
("%s, %i, %x, ...") and then, suitable variables,
but without the end-line char ('\n')

```
raydium_log("You are player %i, %s",player_number,player_name);
```

For now, this function writes to the parent terminal and the in-game console, with "Raydium: " string prefix.
The user can force logging to a file, using `--logfile` command line switch.

4 Random:

4.1 Introduction:

These functions deals with random numbers generation.

4.2 void raydium_random_randomize (void):

This function initialize the random number generator
with current time for seed.
Note: You are not supposed to use this function.

4.3 GLfloat raydium_random_pos_1 (void):

"positive, to one": 0 <= res <= 1

4.4 GLfloat raydium_random_neg_pos_1 (void):

"negative and positive, one as absolute limit": -1 <= res <= 1

4.5 GLfloat raydium_random_0_x (GLfloat i):

"zero to x": 0 <= res <= x

4.6 GLfloat raydium_random_f (GLfloat min, GLfloat max):

min <= res <= max (float)

4.7 int raydium_random_i (int min, int max):

min <= res <= max (integer)

4.8 signed char raydium_random_proba (GLfloat proba):

Returns true or false (0 or 1) depending of "proba" factor.
`proba` must be: 0 <= proba <=1
ex: 50% = 0.5

5 Fog:

5.1 Introduction:

Fog is usefull for two major reasons:

1. Realism: Just try, and you’ll understand:
amazing depth impression, no ?

2. Speed: For a correct fog effect (i’m talking

about estetic aspect), you must bring near_clipping to a closer value,
reducing the overall number of triangles displayed at the same time.

5.2 void raydium_fog_enable (void):

Obvious

5.3 void raydium_fog_disable (void):

Obvious

5.4 void raydium_fog_color_update (void):

If you have modified `raydium_background_color` array, you must
call this function, applying the specified color to hardware.
See also: `raydium_background_color_change`

5.5 void raydium_fog_mode (void):

Do not use. Instable prototype.

6 Window management:

6.1 Introduction:

Some important functions, used for window creation and managment.

6.2 void raydium_window_close (void):

This function is called by Raydium, do not use.

6.3 void raydium_window_create (GLuint tx, GLuint ty, signed char rendering, char *name):

You must call this function once in your program, with following arguments:

1. `tx`, `ty`: window size, in pixel
2. `rendering`: window mode: `RAYDIUM_RENDERING_*` (NONE, WINDOW, FULLSCREEN)
3. `name`: window's name

Raydium is using GLUT for window management, and GLUT fullscreen is not
the same between various implementations, and can fail,
so use a standard window size (640x480, 800x600, ...) for fullscreen mode.

Note that user can force fullscreen using `--fullscreen` on the command line.

6.4 void raydium_window_resize_callback (GLsizei Width, GLsizei Height):

This function is automaticaly called during a window resize,
and resize OpenGL rendering space.

There is almost no reason to call this function by yourself.

6.5 void raydium_window_view_update (void):

If you've changed 3D window size (clipping: raydium_projection_*), apply to hardware with this fonction.

6.6 void raydium_window_view_perspective(GLfloat fov, GLfloat fnear, GLfloat ffar):

All-in-one function: sets all "perspective" variables, and updates.

7 Capture (2D):

7.1 Quickview:

Captures are made in TGA format (without RLE compression) and saved into the current directory.
This function may fail (garbage in resulting capture) if frame size if not "standard", mostly after a window resize.
A new function is available for JPEG captures.

Raydium also allow you to capture movies: activate `DEBUG_MOVIE` option in `raydium/config.h` with the needed framerate, and press F11. Raydium will use a dedicated time line, allowing smooth capture. This system may cause strange behaviours with movies providing network action.
The movie is stored in multiples files in `movie` directory, and you can use mencoder like this:

```
mencoder -ovc lavc -lavcopts vcodec=mpeg4:vhq:vbitrate=780
mf://\*.tga -vf scale=320:240 -mf fps=25 -o ~/ray.avi
```

You can also use audio file adding this:
`-audiofile audio.mp3 -oac copy` for example.

7.2 void raydium_capture_frame(char *filename):

Capture current frame to `filename`.

7.3 void raydium_capture_frame_auto(void):

Same as above, but to an auto-generated filename (raycap*).

7.4 void raydium_capture_frame_jpeg(char *filename):

Same as `raydium_capture_frame()` but using JPEG image format.
See `raydium/config.h` for quality setting.

8 Background:

8.1 void raydium_background_color_change (GLfloat r, GLfloat g, GLfloat b, GLfloat a):

Will change `raydium_background_color` array and apply this modification.
(will update fog color, obviously).

9 Frame clearing:

9.1 void raydium_clear_frame (void):

You need to call this function every frame to clear all hardware buffers.

9.2 void raydium_clear_color_update (void):

Will apply background color modification. Probably useless for you.

10 Lights:

10.1 Introduction to Raydium light system:

When we starts Raydium development, the main idea was to use native OpenGL lights, and not lightmaps or another method.

This method (native lights) provides 8 simultaneous movable lights, and is quite effective with recent OpenGL hardware.

You can modify intensity, position, color, you can turn on any light at any time, make them blinking... Mixing all theses features can result many effects, as realtime sunset, flashing lights for cars, explosions, ...

Usage is very easy: no need to create lights, just turn them on.

See also: LightMaps

10.2 void raydium_light_enable (void):

Obvious.

10.3 void raydium_light_disable (void):

Obvious.

10.4 GLuint raydium_light_to_GL_light (GLuint l):

Probably useless for end user. (internal uses)

10.5 void raydium_light_on (GLuint l):

Turns `l` light on (0 <= l <= RAYDIUM_MAX_LIGHTS)

10.6 void raydium_light_off (GLuint l):

Turns `l` light off

10.7 void raydium_light_switch (GLuint l):

Will swith `l` light state (from "on" to "off", for example).

10.8 void raydium_light_update_position (GLuint l):

Updates `raydium_light_position[l]` array changes to hardware.
This function is now used internaly by Raydium,
so you have no reasons to call it by yourself.

10.9 void raydium_light_update_position_all (void):

See above.

10.10 void raydium_light_update_intensity (GLuint l):

See above.

10.11 void raydium_light_update_all (GLuint l):

See above.

10.12 void raydium_light_move (GLuint l, GLfloat * vect):

Moves light to position `vect` for light `l` (vect is GLfloat[4]: x,y,z,dummy).

Just move your lights before camera placement, or your changes
will be applied to the next frame only.

10.13 void raydium_light_reset (GLuint l):

This function will restore all defaults for `l` light.

10.14 void raydium_light_blink_internal_update (GLuint l):

Useless for end-user.

10.15 void raydium_light_blink_start (GLuint l, int fpc):

Makes `l` light blinking at `fpc` (frames per cycle) rate.
This function will use timecalls soon ("fpc" -> "hertz")

10.16 void raydium_light_callback (void):

Useless for end-user.

11 Keyboard & keys:

11.1 void raydium_key_normal_callback (GLuint key, int x, int y):

Internal callback.

11.2 void raydium_key_special_callback (GLuint key, int x, int y):

Internal callback.

11.3 void raydium_key_special_up_callback (GLuint key, int x, int y):

Internal callback.

11.4 int raydium_key_pressed (GLuint key):

Will return state of `key` in the `raydium_keys[]` array.
This function is usefull to test keyboard from PHP, since RayPHP doest not
support array for now.

12 Mouse:

12.1 Introduction:

Mouse API is almost explainded at the top of this guide, but here it
is some other usefull functions (macros, in facts)

12.2 raydium_mouse_hide() (macro):

Hides mouse cursor.

12.3 raydium_mouse_show() (macro):

Shows mouse cursor.

12.4 raydium_mouse_move(x,y) (macro):

Moves cursor to (x,y) position (in pixel).
Example if you want to move cursor at window’s center:

```
raydium_mouse_move(raydium_window_tx/2, raydium_window_ty/2);
```

12.5 signed char raydium_mouse_isvisible(void):

Returns true or false (0 or 1), if the mouse is visible or not.
See `raydium_mouse_show()` and `raydium_mouse_hide()` above.

12.6 void raydium_mouse_init (void):

Internal use.

12.7 void raydium_mouse_click_callback (int but, int state, int x, int y):

Internal callback.

12.8 void raydium_mouse_move_callback (int x, int y):

Internal callback.

12.9 int raydium_mouse_button_pressed (int button):

returns `button` state. (See first part of this document)

13 Textures:

13.1 Introduction:

For now, Raydium only handles TGA uncompressed texture.
As explainded in the first part of this guide, Raydium provides three
texture filters (none, bilinear, trilinear using MipMaps).

Texture sizes must be a power of two, 8 (alpha mask), 24 (RGB) or 32 (RGBA) bits.

Raydium now supports materials with a simple "rgb(r,g,b)" string
as texture name, where r, g and b are 0 <= x <= 1 (floats).
With 3 negative values, you will generate a "phantom texture". Phantom textures
are only drawn into the z-buffer (and not color buffer).
Texture clamping and multitexturing are supported by Raydium, but not
documented here for now. If you’re interested, have a look at source code, or
take a look at the Wiki.

Tips: "BOX_", ";", "|".

13.2 signed char raydium_texture_size_is_correct (GLuint size):

Returns true if `size` is a correct texture size, depending of
hardware capacities and "power of 2" constraint.

13.3 GLuint raydium_texture_load_internal(char *filename, char *as, signed char faked, int live_id_fake):

Internal use.

13.4 GLuint raydium_texture_load (char *filename):

Loads "filename" texture into hardware memory. Function results
texture index, but in most cases, you can identify later a texture
by his name, without providing his index, so you can probably ignore
this value.

0 is returned if texture loading have failed.

13.5 GLuint raydium_texture_load_erase (char *filename, GLuint to_replace):

Same as above, but `to_replace` texture (index) is erased with `filename`.

13.6 signed char raydium_texture_current_set (GLuint current):

Switch active texture to "current" index. Mostly used for runtime object
creation:
"set current texture, add vertices, set another texture,
add vertices, ... and save all to an objet"
(see below for vertices management).

13.7 signed char raydium_texture_current_set_name (char *name):

Same as above, but using texture name. This function will load `name`
if not alread done.

13.8 GLuint raydium_texture_find_by_name (char *name):

Returns index for texture "name", and load it if not already done.

13.9 void raydium_texture_filter_change (GLuint filter):

This function will change all filters at anytime.
Please note that this function will reload every texture and can be very slow.

```
// will switch all textures to bilinear filter.
raydium_texture_filter_change(RAYDIUM_TEXTURE_FILTER_BILINEAR)
```

14 Rendering:

14.1 void raydium_render_lightmap_color(GLfloat *color):

You may force a new lightmap rendering color "filter" anytime with this function, allowing advanced lighting effects.
HUGE WARNING: You must turn off display lists if you change this value after first object's render.
See `raydium_rendering_displaylists_disable()` if needed.

14.2 void raydium_render_lightmap_color_4f(GLfloat r, GLfloat g, GLfloat b, GLfloat a):

Same as above, using 4 values.

14.3 int raydium_rendering_prepare_texture_unit (GLenum tu, GLuint tex):

This function will "prepare" hardawre texture unit `tu` to render `tex` texture.
There almost no reason to call this function by yourself.

14.4 void raydium_rendering_internal_prepare_texture_render (GLuint tex):

Same as above, but for texture unit #0 only.

14.5 void raydium_rendering_internal_restore_render_state (void):

Internal. Deprecated.

14.6 void raydium_rendering_from_to (GLuint from, GLuint to):

Renders vertices from `from` to `to`.
Using object management functions is a better idea.

14.7 void raydium_rendering (void):

Renders all vertices (probably useless, now).

14.8 void raydium_rendering_finish (void):

You must call this function at the end of each frame. This will flush all commands to hardware, fire a lot off callbacks, and prepare next frame.

14.9 void raydium_rendering_wireframe (void):

Switch to wireframe rendering.

14.10 void raydium_rendering_normal (void):

Switch back to standard rendering.

14.11 void raydium_rendering_rgb_force (GLfloat r, GLfloat g, GLfloat b):

Force all RGB colored vertices to take `(r,g,b)` color. One example of this use is for making "team colored" cars : Do not apply textures to some faces while modelling, and force to team color each time you render a car.

14.12 void raydium_rendering_rgb_normal (void):

Disable "rgb force" state. See above.

14.13 void raydium_rendering_displaylists_disable(void):

Disable display lists usage.
Some old video cards and broken drivers may get better performances WITHOUT

display lists (on large objects, mainly).

14.14 void raydium_rendering_displaylists_enable(void):

Enable display lists usage. default state.

15 Particle engine:

15.1 Introduction:

This is the second version of Raydium's particle engine. This engine is build on top of a dedicated file format (.prt and .sprt files), describing most (up to all, in facts) properties of generators.
It probably better to start by an example (fountain.prt) :

```
// Simple blue fountain (change 'vector' if needed)
ttl_generator=5;
ttl_particles=1.5;
ttl_particles_random=0;

particles_per_second=200;

texture="flare_nb.tga";

size=0.1;
size_inc_per_sec=0.1;

gravity={0,0,-5};
vector={0,0,4};
vector_random={0.2,0.2,0.2};

// RGBA
color_start={0.6,0.6,1,0.5};
color_start_random={0,0,0.2,0};
color_end={1,1,1,0.1};

// end of file.
```

.prt files are readed using parsing functions (see appropriate chapter, if needed), and the list of all available properties can be found in particle2.c source file. A full toturial is also available on Raydium's Wiki.

Once the particle file is written, you only need to load the file using the suitable function (see below). Some anchor are available to link generators to physic entities, if needed, as callbacks for a few events (one, for now).

.sprt files are used to create a "snapshot" of particles, used for example by 3D captures, and are not meant to be edited by hand.

15.2 void raydium_particle_name_auto (char *prefix, char *dest):

Will generate a unique string using `prefix`. The string is created using space provided by `dest`.

You can use this function when building a new generator.

15.3 void raydium_particle_init (void):
Internal use.

15.4 signed char raydium_particle_generator_isvalid (int g):
Internal use, but you can call this function if you want to verify if a
generator’s id is valid (in bounds, and loaded).

15.5 int raydium_particle_generator_find (char *name):
Lookups a generator using is name. Returns -1 if `name` is not found.

15.6 int raydium_particle_find_free (void):
Finds a free particle slot.

15.7 void raydium_particle_generator_delete (int gen):
Deletes a generator.

15.8 void raydium_particle_generator_delete_name (char *gen):
Same as above, but using generator’s name.

15.9 void raydium_particle_generator_enable (int gen, signed char enabled):
Activate a disabled generator (see below).

15.10 void raydium_particle_generator_enable_name (char *gen, signed char enable):
Disable a generator (TTL is still decremented).

15.11 void raydium_particle_preload (char *filename):
Loads .prt file and associated textures into suitable caches.
Call this function if you want to avoid (small) jerks caused by "live"
loading a generator.

15.12 void raydium_particle_generator_load_internal (int generator, FILE * fp, char *filename):
Internal use.

15.13 int raydium_particle_generator_load (char *filename, char *name):
Loads generator from `filename` as `name`. This `name` will be used for
future references to this generator, as the returned interger id.

15.14 void raydium_particle_generator_update (int g, GLfloat step):
Internal use.

15.15 void raydium_particle_update (int part, GLfloat step):
Internal use.

15.16 void raydium_particle_callback (void):

Internal use.

15.17 int raydium_particle_state_dump(char *filename):

Dumped current particles to `filename` (.sprt [static particles]).

15.18 int raydium_particle_state_restore(char *filename):

Append .sprt `filename` to current scene.

15.19 void raydium_particle_draw (raydium_particle_Particle * p, GLfloat ux, GLfloat uy, GLfloat uz, GLfloat rx, GLfloat ry, GLfloat rz):

Internal use.

15.20 void raydium_particle_draw_all (void):

Internal use.

15.21 void raydium_particle_generator_move (int gen, GLfloat * pos):

Moves `gen` generator to `pos` position (3 * GLfloat array).

15.22 void raydium_particle_generator_move_name (char *gen, GLfloat * pos):

Same as above, but using generator’s name.

15.23 void raydium_particle_generator_move_name_3f (char *gen, GLfloat x, GLfloat y, GLfloat z):

Same as above, using 3 different GLfloat values.

15.24 void raydium_particle_generator_particles_OnDelete (int gen, void *OnDelete?):

Sets a callback for `gen`, fired when any particle of this generator is
deleted, providing a easy way to create "cascading" generators.
The callback must respect the following prototype:

```
void cb(raydium_particle_Particle *)
```

Do not free the provided particle.

15.25 void raydium_particle_generator_particles_OnDelete_name (char *gen, void *OnDelete?):

Same as above, but using generator’s name.

15.26 void raydium_particle_scale_all(GLfloat scale):

Will scale all particles with `scale` factor. Use with caution.
Default is obviously 1.

16 Callbacks:

16.1 Introduction:

This file contains many initializations, a few internal callbacks, but will provides a very important function for end-user, wich will gives user display function to Raydium: see below

16.2 void raydium_callback_image (void):

Internal use.

16.3 void raydium_callback_set (void):

Internal use.

16.4 void raydium_callback (void (*loop)):

This function will loop over the provided display function, indefinitely. "loop" must be:

```
void loop(void)
```

17 Normals:

17.1 Introduction:

This file provides some usefull functions for normal generation and smoothing. You can find some more informations about normals at the top of this guide.

17.2 void raydium_normal_generate_lastest_triangle (int default_visu):

Generate normal for the last created triangle (see `raydium_vertex_index`) if `default_visu` is true (!= `0`), this function will restore "visu" normals too.

17.3 void raydium_normal_restore_all (void):

This function restore visu normals with standard ones (`raydium_vertex_normal_*`)

17.4 void raydium_normal_regenerate_all (void):

This function will regenerate standard and visu normals for the whole scene (ground, objects, ...).

17.5 void raydium_normal_smooth_all (void):

This function will smooth the whole scene, using adjacent vertices. Note this function can take a lot of time.

18 vertices:

18.1 Introduction:

You can create objets at runtime, if needed, using the following functions. Each of theses functions adds only one vertex so, obviously, you need to call three time the same function to add one triangle.

18.2 void raydium_vertex_add (GLfloat x, GLfloat y, GLfloat z):

Adds a vertex at (`x,y,z`).

18.3 void raydium_vertex_uv_add (GLfloat x, GLfloat y, GLfloat z, GLfloat u, GLfloat v):

Same as above, but providing texture mapping informations with `u` and `v`.

18.4 void raydium_vertex_uv_normals_add (GLfloat x, GLfloat y, GLfloat z, GLfloat nx, GLfloat ny, GLfloat nz, GLfloat u, GLfloat v):

Same as above, giving vertex's normal with (`nx,ny,nz`).

19 Land:

19.1 Introduction:

Historically, this file was quite complex, since Raydium was using
his own physic. Now, this file is almost empty, since ODE integration
now provides new landscape functions.

20 Sky and environement boxes:

20.1 Introduction:

Skyboxes are mostly automated.

For now, Raydium will use `BOXfront.tga`, `BOXback.tga`, `BOXleft.tga`,
`BOXright.tga`, `BOXbottom.tga` and `BOXtop.tga` and will draw a
skybox only if fog is disabled (this is not for technical reasons,
but only for realism, just think about it ;)... but you can force
skybox with fog using `raydium_sky_force` if you really want).

20.2 void raydium_sky_box_cache (void):

As skybox texture are sometimes large files, you can pre-load skybox
with this function. If you don't do it, Raydium will load textures
during the first frame of your application.

20.3 void raydium_sky_box_render (GLfloat x, GLfloat y, GLfloat z):

Internal use.

21 "Internal" informations access:

21.1 void raydium_internal_dump (void):

This function is now systematically called by Raydium at application's exit,
displaying some informations about loaded textures, objects, registered data,
network statistics.

21.2 void raydium_internal_dump_matrix (int n):

Dumps matrix to console.

n values are:

```
0 for GL_PROJECTION_MATRIX
1 for GL_MODELVIEW_MATRIX
```

22 Files:

22.1 Warning:

It’s important to use only functions with `raydium_file_*` prefix.
All other functions may change or disappear. Upper level functions are available (see `object.c`).

22.2 Introduction:

`file.c` use .tri mesh files (text), available in 3 versions:

1. version 1: providing normals and uv texture mapping informations.
2. version 0: providing normals.
3. version -1: only providing vertices.

Version 1 example file:

```
1
5.1 15.75 -3.82 0.0000 0.0000 -1.0000 0.5158 0.5489 rgb(0.5,0.5,0.5)
6.3 11.75 -3.82 0.0000 0.0000 -1.0000 0.5196 0.5365 rgb(0.5,0.5,0.5)
5.0 11.75 -3.82 0.0000 0.0000 -1.0000 0.5158 0.5365 rgb(0.5,0.5,0.5)
...
```

You can find the file version on first line, and then data.
Next lines: vertex position (x,y,z), normal (x,y,z), texture mapping (u,v) and texture (string).

22.3 void raydium_file_dirname(char *dest,char *from):

Reliable and portable version of libc’s `dirname` function.
This function extracts directory from `from` filename, and writes it to `dest`.
No memory allocation will be done by the function.

22.4 void raydium_file_log_fopen_display(void):

Display (console) all filenames that were opened before the call.
`--files` command line option will call this function at the application’s exit, closed or not.

22.5 FILE *raydium_file_fopen(char *file, char *mode):

Raydium wrapper to libc’s `fopen` function.
This function will:
- Update some stats

- Try to download the file from repositories if no local version is found, or will try to update the file if asked (`--repository-refresh` or `repository-force`). See R3S on Raydium's Wiki.

22.6 void dump_vertex_to (char *filename):

This function save all scene to filename (.tri file) in version 1.
Vertice may be sorted.
Please, try to do not use this function.

22.7 void dump_vertex_to_alpha (char *filename):

Now useless and deprecated.

22.8 int raydium_file_set_textures (char *name):

Internal use.
This function analyze texture filename, and search for extended multitexturing informations (u,v and another texture).

22.9 void read_vertex_from (char *filename):

Loads filename. Again, avoid use of this function.

23 Camera:

23.1 Introduction:

Raydium provides camera management functions, allowing the coder to move camera with very simple functions, even for complex moves.
You have to place your camera once per frame (not more, not less).

"look_at" style functions can be affected by `raydium_camera_look_at_roll` global variable, if needed.

A few words about camera path: Take a look to a .cam file if you want to understand this simple file format, but you probably only need the `cam.c` application, dedicated to camera path creation.

Some camera functions are provided by physics module, see suitable chapter.

23.2 void raydium_camera_vectors (GLfloat * res3):

This function will return two vectors (2 * 3 * GLfloat), giving the camera orientation (front vector and up vector). At this day, the up vector is always the same as the world up vector, even if the camera is rotated or upside down (and yes, this MUST be corrected :).

Designed for internal uses, before all.

23.3 void raydium_camera_internal_prepare(void):

Internal use. (pre)

23.4 void raydium_camera_internal (GLfloat x, GLfloat y, GLfloat z):

Internal use. (post)

23.5 void raydium_camera_place (GLfloat x, GLfloat y, GLfloat z, GLfloat lacet, GLfloat tangage, GLfloat roulis):

Sets the camera at (x,y,z) position, and using (lacet,tangage,roulis)
as rotation angles.

23.6 void raydium_camera_look_at (GLfloat x, GLfloat y, GLfloat z, GLfloat x_to, GLfloat y_to, GLfloat z_to):

Sets the camera at (x,y,z) position, and looks at (x_to,y_to,z_to).

23.7 void raydium_camera_replace (void):

You'll need to reset camera position and orientation after each object drawing.
If this is unclear to you, read the "example" section, below.

You will need to make your own 3D transformations (GLRotate, GLTranslate,
...) to draw your objects, or you can use the following function.

23.8 void raydium_camera_replace_go (GLfloat * pos, GLfloat * R):

This function will replace the camera, as `raydium_camera_replace()`,
but will place "3D drawing cursor" at position `pos` (3 GLfloat) with
rotation `R` (4 GLfloat quaternion).

No eulers (rotx, roty, rotz) version of this function is provided for now..
Do you really need it ?

23.9 Example of camera use:

1. place camera
2. move "drawing cursor" to object's place
3. draw object
4. reset camera to initial place (the one given at step 1)
5. move "drawing cursor" to another object's place
6. draw another object
7. [...]

Steps 4 and 5 can be done with raydium_camera_replace_go().

23.10 void raydium_camera_rumble(GLfloat amplitude, GLfloat ampl_evo, GLfloat secs):

Camera (any type) will rumble for `secs` seconds, with `amplitude` (radians).
This `amplitude` will be incremented of `ampl_evo` every second (negative
values are allowed).
An `amplitude` is always positive.

23.11 void raydium_camera_smooth (GLfloat px, GLfloat py, GLfloat pz, GLfloat lx, GLfloat ly, GLfloat lz, GLfloat zoom, GLfloat roll, GLfloat step):

Smooth style clone of `raydium_camera_look_at`.
Roll is given by `roll` and not global variable `raydium_camera_look_at_roll`

as for regular look_at function.
`zoom` is the requested FOV.
Play with step to modify smoothing level of the movement. A good way to use this function is the following usage :

```
raydium_camera_smooth(cam[0],cam[1],cam[2],pos[1],-pos[2],pos[0],70,0,raydium_frame_time*3
```

23.12 void raydium_camera_path_init (int p):

Internal use.

23.13 void raydium_camera_path_init_all (void):

Internal use.

23.14 int raydium_camera_path_find (char *name):

Lookups path’s id using filename `name`.
This function will not try to load a camera path if it’s not found, and will return -1.

23.15 int raydium_camera_path_load (char *filename):

Obvious : use this function to load a camera path.

23.16 void raydium_camera_path_draw (int p):

Draws `p` camera path, as red lines. This must be done at each frame.

23.17 void raydium_camera_path_draw_name (char *path):

Same as above, but using camera path’s name.

23.18 signed char raydium_camera_smooth_path (char *path, GLfloat step, GLfloat * x, GLfloat * y, GLfloat * z, GLfloat * zoom, GLfloat * roll):

Returns the (`x,y,z`) point of the camera path for step `step`, using provided `zoom` (FOV) and `roll` angle.
It’s important to note that `step` is a float.
Mostly for internal use.

23.19 void raydium_camera_path_reset(void):

Next smooth call will be instantaneous.

23.20 void raydium_camera_smooth_path_to_pos (char *path, GLfloat lx, GLfloat ly, GLfloat lz, GLfloat path_step, GLfloat smooth_step):

"Camera on path looking to a point"
simple `raydium_camera_smooth` version:
Give a path name, a "look_at" point (`lx,ly,lz`), a current `step`, and a `smooth_step` time factor (see `raydium_camera_smooth` example above).

23.21 void raydium_camera_smooth_pos_to_path (GLfloat lx, GLfloat ly, GLfloat lz, char *path, GLfloat path_step, GLfloat smooth_step):

"Camera on point looking at a path"
Same style as previous function.

23.22 void raydium_camera_smooth_path_to_path (char *path_from, GLfloat path_step_from, char *path_to, GLfloat path_step_to, GLfloat smooth_step):

"Camera on a path looking at another path"
Same style as previous functions.

24 Objects:

24.1 Introduction:

With the following functions, you can easily draw and manage
mesh objects (.tri file).

24.2 GLint raydium_object_find (char *name):

Lookups an object by his `name`. This function will return -1 if the
object's not found, and will not try to load the .tri file.

24.3 GLint raydium_object_find_load (char *name):

Same as above (`raydium_object_load`), but will try to load object

24.4 void raydium_object_reset (GLuint o):

Internal use. Do not call.

24.5 int raydium_object_load (char *filename):

Load `filename` as a .tri file, and returns corresponding id, or
-1 in case of error.

24.6 void raydium_object_draw (GLuint o):

Draws `o` (index) object, using current matrixes.

24.7 void raydium_object_draw_name (char *name):

Same as above, but you only have to provide object's `name` (".tri file").
If this object was not already loaded, this function will do it for you.

24.8 void raydium_object_deform (GLuint obj, GLfloat ampl):

Early devel state. Useless as is.

24.9 void raydium_object_deform_name (char *name, GLfloat ampl):

Early devel state. Useless as is.

24.10 GLfloat raydium_object_find_dist_max (GLuint obj):

This function will return will return the distance form (0,0,0)
to the farest point of `obj` object.

24.11 void raydium_object_find_axes_max (GLuint obj, GLfloat * tx, GLfloat * ty, GLfloat * tz):

This function returns the (maximum) size of the bounding box of `obj` (relative to (0,0,0)).

25 Initialization:

25.1 Introduction:

This file is mainly designed for internal uses, but there’s anyway some interesting functions.

25.2 int raydium_init_cli_option (char *option, char *value):

This function will search command line `option`.
If this option is found, the functions stores any argument to `value` and returns 1.
The function will return 0 if `option` is not found.

Example (search for: `--ground`)

```
char ground[RAYDIUM_MAX_NAME_LEN];
if(raydium_init_cli_option("ground",model))
{
setground(model);
}
```

25.3 int raydium_init_cli_option_default (char *option, char *value, char *default_value):

Same as above, but allows you to provide a default value (`default`) if the `option` is not found on command line.

25.4 void raydium_init_lights (void):

Internal use. Must be moved to light.c.

25.5 void raydium_init_objects (void):

Internal use. Must be moved to object.c.

25.6 void raydium_init_key (void):

Internal use. Must be moved to key.c.

25.7 void raydium_init_reset (void):

This function is supposed to reset the whole Raydium engine:
textures, vertices, lights, objects, ...
Never tested yet, and probaly fails for many reasons when called more than one time.

25.8 void raydium_init_engine (void):

Internal use. Never call this function by yourself, it may cause huge memory leaks.

25.9 void raydium_init_args (int argc, char * *argv):

You must use this function, wich send application arguments to Raydium
and external libs (GLUT, OpenAL, ...).
This must be done before any other call to Raydium.
Example:

```
int main(int argc, char **argv)
{
raydium_init_args(argc,argv);
[...]
```

26 Signals:

26.1 Quickview:

There almost nothing to said about signals management, except that Raydium
will try to catch SIGINT signal (sended by CTRL+C sequence, for example).
There's nothing else for now, but we plan a user callback for this signal.

27 Sound and music:

27.1 Introduction:

The Raydium sound API is pretty easy to use and there's only need to use a
few functions to make your program ouput sounds or music.

On top of this, there are a bunch of functions to modify the sound behavior.

Raydium uses OpenAL and OggVorbis?? for its sounds and musics, for a basic
use of our sound API you only need to know one thing: OpenAL uses buffers
for its sounds and you need to be able to address the sounds separately.
For this we use ALuint in our code. Each buffer is associated to a source,
we have an array of all available sources and then, you only need to have
a simple int that acts as an index in this array. See below for more
informations.

Music is readed thru libogg, streamed from disk. If you want to play an
OGG audio track, the only thing you've to do is to call the suitable function.
You can use `raydium_sound_music_eof_callback` if needed. This event is
fired when sound track ends, allowing you to switch to another file.
Prototype for this callback is `int callback(char *new_track)`, allowing
you to do something like `strcpy(new_track,"foobar.ogg"); return 1;`.
Return 0 if you do not want to switch to another audio file (this will stops
music playback).

This document is not an alternative to OpenAL papers, and only provides
informations about Raydium's interface to OpenAL.
See specifications here: http://www.openal.org/documentation.html

27.2 void raydium_sound_verify (char *caller):

This functions checks if any error occured during last OpenAL operation.
You don’t have to call this function by yourself, since every function of
this API will do it.

27.3 int raydium_sound_Array3IsValid(ALfloat *a):

Since OpenAL is very sensitive to malformed values, this function is used
internally to check consistency of provided ALfloat arrays.

27.4 void raydium_sound_InitSource (int src):

Internal use.

27.5 int raydium_sound_LoadWav (const char *fname):

This function tries to load the `fname` wav file into a buffer, if
successful, it returns the source id, else 0.

27.6 int raydium_sound_SourceVerify (int src):

Internal id checks.

27.7 int raydium_sound_SetSourceLoop (int src, signed char loop):

Modifies the `loop` property of the `src` source (loops if loop is non-zero,
default value for a source is "true").
Returns 0 if ok, -1 if error.

27.8 int raydium_sound_GetSourcePitch (int src, ALfloat * p):

Returns current pitch for `src` source.

27.9 int raydium_sound_SetSourcePitch (int src, ALfloat p):

Sets pitch for `src` source.
Current OpenAL spec is not clear about pitch’s limits. Raydium will
clamp values to to]0,2] interval.

27.10 int raydium_sound_GetSourceGain (int src, ALfloat * g):

Returns current gain ("volume") for `src` source.

27.11 int raydium_sound_SetSourceGain (int src, ALfloat g):

Sets gain ("volume") for `src` source.
Current OpenAL spec is not clear about pitch’s limits. Raydium do not allows
negative values, but no upper limit is set.
Warning: some OpenAL implementations will provide strange gain curves. More
work is needed on this issue.

27.12 int raydium_sound_SetSourcePos (int src, ALfloat Pos[]):

Sets 3D position of `src` source.
`Pos` is a 3 * ALfloat array.

27.13 int raydium_sound_SetSourcePosCamera(int src):

Sets 3D position of `src` source on the current camera position.

27.14 int raydium_sound_GetSourcePos (int src, ALfloat * Pos[]):

Returns current 3D position of `src` source.
`Pos` is a 3 * ALfloat array.

27.15 int raydium_sound_SetSourceDir (int src, ALfloat Dir[]):

Sets 3D direction of `src` source.
`Dir` is a 3 * ALfloat array.

27.16 int raydium_sound_GetSourceDir (int src, ALfloat * Dir[]):

Returns current 3D direction of `src` source.
`Dir` is a 3 * ALfloat array.

27.17 int raydium_sound_SetSourceVel (int src, ALfloat Vel[]):

Sets 3D velocity of `src` source.
`Vel` is a 3 * ALfloat array.

27.18 int raydium_sound_GetSourceVel (int src, ALfloat * Vel[]):

Returns current 3D velocity of `src` source.
`Vel` is a 3 * ALfloat array.

27.19 void raydium_sound_SetListenerPos (ALfloat Pos[]):

Sets 3D position of listener.
This is done automatically by Raydium, each frame, using camera informations
`Pos` is a 3 * ALfloat array.

27.20 void raydium_sound_GetListenerPos (ALfloat * Pos[]):

Returns current 3D position of listener.
`Pos` is a 3 * ALfloat array.

27.21 void raydium_sound_SetListenerOr (ALfloat Or[]):

Sets 3D orientation of listener.
This is done automatically by Raydium, each frame, using camera informations
`Or` is a 3 * ALfloat array.

27.22 void raydium_sound_GetListenerOr (ALfloat * Or[]):

Returns current 3D orientation of listener.
`Or` is a 3 * ALfloat array.

27.23 void raydium_sound_SetListenerVel (ALfloat Vel[]):

Sets 3D velocity of Listener.
`Vel` is a 3 * ALfloat array.

27.24 void raydium_sound_GetListenerVel (ALfloat * Vel[]):

Returns current 3D velocity of Listener.
`Vel` is a 3 * ALfloat array.

27.25 void raydium_sound_init (void):

Internal use.

27.26 int raydium_sound_SourcePlay (int src):

Plays the `src` source.
If `src` was already in "play" state, the buffer is rewinded.
Returns 0 if ok, -1 if error.

27.27 int raydium_sound_SourceStop (int src):

Stops the `src` source.
Returns 0 if ok, -1 if error.

27.28 int raydium_sound_SourcePause (int src):

Will pause the `src` source.
Returns 0 if ok, -1 if error.

27.29 int raydium_sound_SourceUnpause (int src):

`src` will restart playback after being paused.
Returns 0 if ok, -1 if error.

27.30 void raydium_sound_close (void):

Internal use.

27.31 int raydium_sound_load_music (char *fname):

Opens fname OGG music file and prepairs Raydium for playing it.
The music will be automatically played after a call to this function.
This function will use R3S (data repositories) if needed.
To switch to another audio track, simply call again this function.
Send `NULL` or an empty string to cancel music playback.
Returns 0 if ok, -1 if error

See also `raydium_sound_music_eof_callback` at the top of this chapter.

27.32 void raydium_sound_music_callback (void):

Internal use.

27.33 void raydium_sound_callback (void):

Internal use.

27.34 void raydium_sound_source_fade(int src, ALfloat len):

This function will fade down source `src` over `len` seconds.
Since gain is not linear, you may have to play with `len` to
find the correct value.
Use source 0 for music source.

27.35 Sound API Example:

```
int sound;
sound=raydium_sound_LoadWav("explo.wav");
raydium_sound_SetSourceLoop(sound,0);
[...]
if(explosion) raydium_sound_SourcePlay(sound);
```

28 Timecalls:

28.1 Concept:

As you may already know, in a real time application (as a game), you need to control in-game time evolution.
For example, you cannot increment a car position by 1 at each frame since it will generate an irregular scrolling (a frame is never rendered within the same time as the previous or the next one).

Raydium supports timecalls, wich are a great solution for this problem.
Usage is very simple: write a simple function, and ask Raydium to call it at the desired rate.

28.2 Constraints:

There is an important risk with timecalls: infinite loops.
If a callback is long, it may take more CPU time than he would, as in this very simple example:

foo() is a function, taking 200 ms for his own execution. If you ask for a 6 Hz execution, Raydium will execute foo() six times on the first frame, taking 1200 ms. On the next frame, Raydium will need to execute foo() 7 times (the asked 6 times, and one more for the 200 ms lost during the last frame), taking 1400 ms, so 8 times will be needed for the next frame, then 9, ...

So you need to create callbacks as short as possible, since long callbacks may cause a game freeze on slower machines than yours. (1 FPS syndrom)

28.3 Hardware devices and methods:

Raydium must use a very accurate system timer, and will try many methods:
`/dev/rtc` , `gettimeofday()` (Linux only) and
`QueryPerformanceCounter`? for win32.

`gettimeofday()` will use a CPU counter and is extremely accurate.
It’s far the best method. (0.001 ms accuracy is possible)

`/dev/rtc` is quite good, and Raydium will try to configure RTC at
`RAYDIUM_TIMECALL_FREQ_PREFERED` rate (8192 Hz by default), but may require a "`/proc/sys/dev/rtc/max-user-freq`" modification:

```
echo 8192 > /proc/sys/dev/rtc/max-user-freq
```

You may want to look at common.c for interesting defines about timecalls.

28.4 void raydium_timecall_raydium (GLfloat step):
Internal Raydium callback.

28.5 float raydium_timecall_internal_w32_detect_modulo(int div):
Internal, WIN32 only: Returns timer resolution for `div` divisor.

28.6 int raydium_timecall_internal_w32_divmodulo_find(void):
Internal, WIN32 only: Detects the best timer divisor for the current CPU.

28.7 unsigned long raydium_timecall_devrtc_clock (void):
Internal, Linux only: Reads and return RTC clock.

28.8 unsigned long raydium_timecall_clock (void):
Returns current "time".

28.9 signed char raydium_timecall_devrtc_rate_change (unsigned long new):
Internal, Linux only: Modifies RTC clock rate.

28.10 void raydium_timecall_devrtc_close (void):
Internal, Linux only: Will close RTC clock.

28.11 unsigned long raydium_timecall_devrtc_init (void):
Internal, Linux only: Will open RTC clock.

28.12 int raydium_timecall_detect_frequency (void):
Internal: This function will find the best timer available for current platform, and adjust properties to your hardware (rate, divisor, ...).

28.13 void raydium_timecall_init (void):
Internal use.

28.14 int raydium_timecall_add (void *funct, GLint hz):
There is two sort of timecalls with Raydium:

1. Standard ones:

```
raydium_timecall_add(function,800);
```

`void function(void)` will be called 800 times per second.

2. Elastic timed ones:

```
raydium_timecall_add(function,-80);
```

`void function(float step)` will be called for each frame, with a "`step` factor" as argument. In the above example, a 160 Hz game will call function with step = 0.5, but step = 2.0 for a 40 Hz game.

A standard timecall will use `void(void)` function and a positive `hertz` argument, as an elasitc one will use `void(float)` and negative `hertz` argument.

28.15 void raydium_timecall_freq_change (int callback, GLint hz):

This function changes the `callback` frequency. See above for possibles values of `hz` (negative and positive values).

28.16 void raydium_timecall_callback (void):

Internal use (frame fired callback).

29 Network:

29.1 Bases of Raydium’s networking API:

Raydium supports networking via UDP/IP, providing high level functions for multiplayer game development.
Raydium servers are limited to 256 clients for now.

You will find in network.c a set of functions and vars dedicated to networked games: players names, event callbacks, UDP sockets, broadcasts, ...
All this is ready to use. As it’s not done in the introduction of this guide, We will explain here some variables defined in common.c.

```
#define RAYDIUM_NETWORK_PORT          29104
#define RAYDIUM_NETWORK_PACKET_SIZE   230
#define RAYDIUM_NETWORK_TIMEOUT       5
#define RAYDIUM_NETWORK_PACKET_OFFSET 4
#define RAYDIUM_NETWORK_MAX_CLIENTS   8
#define RAYDIUM_NETWORK_MODE_NONE     0
#define RAYDIUM_NETWORK_MODE_CLIENT   1
#define RAYDIUM_NETWORK_MODE_SERVER   2
```

Here, we can find network port declaration (Raydium will use only one port, allowing easy port forwarding management, if needed), default timeout (unit: second), and the three mode possible for a Raydium application.

But there is also two other very important defines: packet size (unit: byte) and max number of clients.. This is important because Raydium uses UDP sockets, and UDP sockets required fixed length packets, and as you need to set packet size as small as possible (for obvious speed reasons), you must calculate you maximum information packet size (players position, for example), multiply it by `RAYDIUM_NETWORK_MAX_CLIENTS`,and add `RAYDIUM_NETWORK_PACKET_OFFSET` wich represent the required header of the packet.

It’s more easy than it seems, look:

My game will support 8 players.
I will send players state with 3 floats (x,y,z).
My packet size must be: 8*3*sizeof(float)+RAYDIUM_NETWORK_PACKET_OFFSET = 100 bytes.

Please, do not change packet offset size, since Raydium will use it
for packet header.

```
#define RAYDIUM_NETWORK_DATA_OK     1
#define RAYDIUM_NETWORK_DATA_NONE   0
#define RAYDIUM_NETWORK_DATA_ERROR -1
```

This three defines are used as network functions result:

```
if(raydium_network_read_flushed(&id,&type,buff)==RAYDIUM_NETWORK_DATA_OK)
{
...
```

```
#define RAYDIUM_NETWORK_PACKET_BASE 20
```

In most network functions, you will find a "type" argument, used to
determine packet goal. This type is 8 bits long (256 possible values),
but Raydium is already using some of them. So you can use
`RAYDIUM_NETWORK_PACKET_BASE` as a base for your own types:

```
#define NORMAL_DATA RAYDIUM_NETWORK_PACKET_BASE
#define BALL_TAKEN (NORMAL_DATA+1)
#define SCORE_INFO (NORMAL_DATA+2)
#define HORN (NORMAL_DATA+3)
...
```

Variables:

Your own player id (0<= id < RAYDIUM_NETWORK_MAX_CLIENTS),
read only: `int raydium_network_uid;`
Special value "-1" means that you’re not connected (see below).

Current network mode (none, client, server),
read only: `signed char raydium_network_mode;`

Boolean used to determine client state (connected or not), read only:
`signed char raydium_network_client[RAYDIUM_NETWORK_MAX_CLIENTS];`

example:

```
if(raydium_network_client[4])
draw_player(4);
```

Can be used by a server to send data to his clients. Read only:
`struct sockaddr raydium_network_client_addr[RAYDIUM_NETWORK_MAX_CLIENTS];`

Players names, read only:
`char raydium_network_name[RAYDIUM_NETWORK_MAX_CLIENTS][RAYDIUM_MAX_NAME_LEN];`

`OnConnect`? and `OnDisconnect`? events (server only):
`void * raydium_network_on_connect;`
`void * raydium_network_on_disconnect;`

You can place your owns callbacks (`void(int)`) on these events, as in this example:

```
void new_client(int client)
{
raydium_log("New player: %s", raydium_network_nameclient);
}

...

int main(int argc, char **argv)
{
...
raydium_network_on_connect=new_client;
...
```

29.2 Reliablility versus Speed:

As explained above, Raydium is using UDP network packets, and as you may know, UDP is not a reliable protocol, aiming speed before all.
This system is interesting for sending non-sensible data, as player positions, for example.
But Raydium can handle more important data, using some of methods of TCP protocol, as Timeouts, ACK, resending, ...
This TCP style packets are available thru "Netcalls".

29.3 High level API: "Netcalls" and "Propags":

Netcalls provides you a good way to handle network exchanges using callbacks functions, like a simple RPC system.

The idea is simple, built over the notion of "type". See suitable functions for more information about this system.

Another available mechanism is called Propags, and allows you to "share" variables over the network (scores, game state, ...) in a very few steps. You only need to "create" a type, and link a variable to it (any C type or structure is allowed). After each modification of this (local copy of the) variable, just call `raydium_network_propag_refresh*` and that's it. If any other client (or the server) is applying a modification to this "type", your local copy is automatically updated.

29.4 int raydium_network_propag_find (int type):

Lookups a "propag" by his `type`. Returns -1 is no propag is found.

29.5 void raydium_network_propag_recv (int type, char *buff):

Internal callback for "propag" receiving.

29.6 void raydium_network_propag_refresh_id (int i):

Will refresh a propag by his `id`.

29.7 void raydium_network_propag_refresh (int type):

Will refresh a propag by his `type`.

29.8 void raydium_network_propag_refresh_all (void):

Will refresh all propags

29.9 int raydium_network_propag_add (int type, void *data, int size):

This function will "register" a new propag. You need to provide the address of your variable/structure (`data`), ans its `size`. A dedicated `type` is also required (see at the top of this chapter).

29.10 void raydium_network_queue_element_init (raydium_network_Tcp * e):

Internal use. (TCP style packets)

29.11 unsigned short raydium_network_queue_tcpid_gen (void):

Internal use. (TCP style packets)

29.12 void raydium_network_queue_tcpid_known_add (int tcpid, int player):

Internal use. (TCP style packets)

29.13 signed char raydium_network_queue_tcpid_known (unsigned short tcpid, unsigned short player):

Internal use. (TCP style packets)

29.14 signed char raydium_network_queue_is_tcpid (int type):

Internal use. (TCP style packets)

29.15 void raydium_network_queue_element_add (char *packet, struct

sockaddr *to):
Internal use. (TCP style packets)

29.16 unsigned long *raydium_network_internal_find_delay_addr (int player):
Internal use. (TCP style packets)

29.17 void raydium_network_queue_check_time (void):
Internal use. (TCP style packets)

29.18 void raydium_network_queue_ack_send (unsigned short tcpid, struct sockaddr *to):
Internal use. (TCP style packets)

29.19 void raydium_network_queue_ack_recv (int type, char *buff):
Internal use. (TCP style packets)

29.20 void raydium_network_player_name (char *str):
This function will returns the current player name.
Raydium will ask the OS for "current logged user", but player name may
be provided thru `--name` command line argument.

29.21 signed char raydium_network_set_socket_block (int block):
This function will sets `block` (true or false) status to the network stack.
A blocking socket will wait indefinitely an incoming packet. A non blocking one
will return "no data" instead.
You’ve almost no reason to call this function by yourself.

29.22 signed char raydium_network_netcall_add (void *ptr, int type, signed char tcp):
This function will register a new Network Callback ("netcall").
With Raydium, you can read the main data stream with
`raydium_network_read_flushed()`, and configure netcalls on random
events (using packet type).

Netcalls signature is: `void(int type, char *buff)`

As you may configure the same callback function for multiples packet types,
this type is passed to your function, with the temporary `buff` buffer.
You can extract from field from packet if needed.

If you sets the `tcp` flag to true (1), your packet will use "TCP style"
network protocol (see a the top of this chapter).

29.23 void raydium_network_netcall_exec (int type, char *buff):
Internal callback for "netcall" receiving.

29.24 signed char raydium_network_timeout_check (void):
Internal use.

29.25 signed char raydium_network_init (void):

Nothing interesting unless you’re creating a console server (using the `RAYDIUM_NETWORK_ONLY` directive), since in this case you must do all inits by yourself...
example :

```
#define RAYDIUM_NETWORK_ONLY
#include "raydium/index.c"

...

int main(int argc, char **argv)
{
setbuf(stdout,NULL);
signal(SIGINT,quit);
raydium_php_init(); // only if you need PHP support
raydium_network_init();
raydium_network_server_create();
...
```

29.26 void raydium_network_write (struct sockaddr *to, int from, signed char type, char *buff):

Obviously, this function will send data.
If you’re a client, you don’t need to determine to field, as the only destination is the server, so you can use `NULL`, for example. If you’re a server, you can use `raydium_network_client_addr[]` array.

As a client, `from` argument is generally your own uid (`raydium_network_uid`), but you can use any other player number if needed.
As a server, `from` field is useless, since you are the only machine able to send data to clients.

As you may expect, `type` field is used to determine packet’s type.
You can use any (8 bits) value greater or equal to `RAYDIUM_NETWORK_PACKET_BASE`.

Finally, `buff` is a pointer to data’s buffer. This buffer must be `RAYDIUM_NETWORK_PACKET_SIZE` long, and can be cleared or re-used after this call.

29.27 void raydium_network_broadcast (signed char type, char *buff):

Sends data over network.
Obviously, from network point of vue, only a server can broadcast (to his clients).

When a client needs to broadcast (from the game point of vue) some informations (his own position, for example), he must send this information to server, and the server will broadcast it.

This function uses the same arguments as previous one, except `to` and `from`, not needed here.

29.28 signed char raydium_network_read (int *id, signed char *type, char *buff):

Reads next packet from network (FIFO) stack.
This function uses the same arguments as previous ones, and returns data availability: `RAYDIUM_NETWORK_DATA_OK`, `RAYDIUM_NETWORK_DATA_NONE` or `RAYDIUM_NETWORK_DATA_ERROR`.

29.29 signed char raydium_network_read_flushed (int *id, signed char *type, char *buff):

Reads last packet from network stack.
All previous packets will be ignored, only the newest packet will be read (if any).

As you may miss some important informations, you can use netcalls (see above) if you want to capture packets with a particular type, even with flushed reading.

29.30 signed char raydium_network_server_create (void):

Will transform you application into a server, accepting new clients instantaneously.
See also the `RAYDIUM_NETWORK_ONLY` directive if you want to create console servers.

29.31 signed char raydium_network_client_connect_to (char *server):

This function will try to connect your application to `server` (hostname or ip address).
WARNING: For now, this call could be endless ! (server failure while connecting).
This function will succed returning 1 or 0 otherwise.
You are connected instantaneously, and you must start sending data before server timeout (defined by `RAYDIUM_NETWORK_TIMEOUT`).
You player number can be found with `raydium_network_uid` variable, as said before.

29.32 signed char raydium_server_accept_new (struct sockaddr *from, char *name):

Internal server callback for new clients.

29.33 void raydium_network_close (void):

Obvious. Raydium will do it for you, anyway.

29.34 void raydium_network_internal_server_delays_dump (void):

Dumps "TCP Style" timeouts for all clients to console.

29.35 void raydium_network_internal_dump (void):

Dumps various stats about network stack to console.

29.36 signed char raydium_network_internet_test(void):

This function will test if direct internet connection is available,
using a DNS root server. Use with caution.

30 OSD (On Screen Display):

30.1 Introduction:

Raydium provides some high level function for "On Screen Display",
as string drawing (2D and 3D), application’s logo, mouse cursor, and other
various 2D displaying tools.

In most cases, these functions must be called after any other object
drawing function, to avoid overlapping problems.

Most functions will use a percentage system, and origin is at lower-left corner.

30.2 void raydium_osd_color_change (GLfloat r, GLfloat g, GLfloat b):

This function will change the font color for the next `raydium_osd_printf*`
calls.
As usual: 0 <= (`r`,g and b) <= 1.

30.3 void raydium_osd_alpha_change (GLfloat a):

Same as above, but will change font transparency.

30.4 void raydium_osd_color_rgba (GLfloat r, GLfloat g, GLfloat b, GLfloat a):

This is a mix of `raydium_osd_color_change` and `raydium_osd_alpha_change`.

30.5 void raydium_osd_color_ega (char hexa):

This function will change font color with the corresponding
`hexa`decimal code (as a char: ’0’ to ’F’) in the standard EGA palette.

Here is this palette:

Hexa	Color
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Purple
6	Brown
7	White
8	Grey

9	Light Blue
A	Light Green
B	Light Cyan
C	Light Red
D	Light Purple
E	Light Yellow
F	Light White

30.6 void raydium_osd_start (void):

Mostly for internal uses. (will configure screen for OSD operations)

30.7 void raydium_osd_stop (void):

Mostly for internal uses. (see above)

30.8 void raydium_osd_draw (int tex, GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2):

Will draw `tex` texture using (`x1`,`y1`) and (`x2`,`y2`) points.

30.9 void raydium_osd_draw_name (char *tex, GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2):

Same as above, but using texture filename.

30.10 void raydium_osd_printf (GLfloat x, GLfloat y, GLfloat size, GLfloat spacer, char *texture, unsigned char *format, ...):

This function is an OpenGL equivalent to the standard "printf" C function.

- (`x`,`y`) is the position of the text's beginning, as a screen
percentage, with origin at lower left.

- `size` is the font size, using an arbitrary unit. This size is always
proportionnal to frame size (font size will grow up with screen size,
in other words).

- `spacer` is the factor of spacing between 2 consecutive letters. With
standard fonts, 0.5 is a correct value (relatively condensed text).

- `texture` is obviously the texture filename to use (font*.tga are
often provided with Raydium distribution, and by R3S).

- `format` is the standard printf format string, followed by
corresponding arguments: `"^9Player ^Fname is: %10s", player_name`
This format can use '^' char to change color text, followed by a color,
indicated by a hexadecimal letter (EGA palette). See `raydium_osd_color_ega`

function, above.

Here you are a simple example:

```
strcpy(version,"^Ctest 0.1^F");
raydium_osd_printf(2,98,16,0.5,"font2.tga","- %3i FPS - tech demo %s for Raydium %s, CQFD
raydium_render_fps,version,raydium_version);
```

30.11 void raydium_osd_printf_3D (GLfloat x, GLfloat y, GLfloat z, GLfloat size, GLfloat spacer, char *texture, unsigned char *format, ...):

Same as above, but you can place your text in your application 3D space,
using `x`, `y` and `z` values.

30.12 void raydium_osd_logo (char *texture):

Will draw a logo for the current frame with texture filename.
For now, you’ve no control over rotation speed of the logo.

30.13 void raydium_osd_cursor_set (char *texture, GLfloat xsize, GLfloat ysize):

This function will set mouse cursor with texture filename and
with (`xsize`,`ysize`) size (percent of screen size).
You should use a RGBA texture for better results.
example:

```
raydium_osd_cursor_set("BOXcursor.tga",4,4);
```

30.14 void raydium_osd_cursor_draw (void):

Internal use.

30.15 void raydium_osd_internal_vertex (GLfloat x, GLfloat y, GLfloat top):

Internal use.

30.16 void raydium_osd_network_stat_draw (GLfloat px, GLfloat py, GLfloat size):

Will draw network stats (if available) in a box.

```
raydium_osd_network_stat_draw(5,30,20);
```

30.17 void raydium_osd_mask (GLfloat * color4):

Will draw a uniform mask using `color4` (RGBA color) for this frame.

30.18 void raydium_osd_mask_texture(int texture,GLfloat alpha):

Will draw a textured mask, with `alpha` opacity (1 is full opacity).

30.19 void raydium_osd_mask_texture_name(char *texture,GLfloat alpha):

Same as above, but resolving texture by name.

30.20 void raydium_osd_mask_texture_clip(int texture,GLfloat alpha, GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2):

Same as `raydium_osd_mask_texture`, but (x1,y1),(x2,y2) will be used as texture coords, in a [0,100] range.

30.21 void raydium_osd_mask_texture_clip_name(char *texture,GLfloat alpha, GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2):

Same as above, but resolving texture by name.

30.22 void raydium_osd_fade_callback (void):

Internal use.

30.23 void raydium_osd_fade_init (void):

Internal use.

30.24 void raydium_osd_fade_from (GLfloat * from4, GLfloat * to4, GLfloat time_len, void *OnFadeEnd?):

This function will configure a fading mask from `from4` color to `to4`.
This fade will last `time_len` seconds, and will call `OnFadeEnd?` callback
when finished.
This callback signature must be `void callback(void)`.

A standard fade-to-black-and-restore example:

```
// back to normal rendering
void restorefade(void)
{
GLfloat from[4]={0,0,0,2};
GLfloat to[4]={0,0,0,0};
raydium_osd_fade_from(from,to,1,NULL);
// do things (like moving camera to another place, for example).
}

...

// If space key : fade to black
if(raydium_key_last==1032)
{
GLfloat from[4]={0,0,0,0};
GLfloat to[4]={0,0,0,1};
raydium_osd_fade_from(from,to,0.3,restorefade);
}
```

31 In-game console:

31.1 Introduction:

This chapter introduce Raydium console, allowing applications to take user keyboard input (game commands, chat, ...) and to send informations to this console.
The end user can call the console using "the key below esc".

By default, if PHP support is enabled, all user commands will be redirected to PHP engine. Each command will get his own context, don't expect to create anything else than "single line PHP scripts" with the console. See PHP chapter for more informations.
The console allows the user to prefix command with the following characters:

- `/`: Non PHP command. The command will be sent to application (see `raydium_console_gets_callback`, below.

- `>`: Will launch argument as a PHP script (identical to `include("...")`)

- `!`: Will launch argument as a sequence script

Command history is saved to `raydium_history` file when application exits.

You can use a `void prompt(char *)` callback to get user commands. Your callback must be registered thru `raydium_console_gets_callback`:

```
raydium_console_gets_callback=prompt;
```

This console provides auto-completion of register functions and variables.
See the suitable chapter for more information.

31.2 void raydium_console_init (void):

Internal use.

31.3 void raydium_console_history_save (void):

Internal use (will flush console history to disk).
You can call it by yourself if needed.

31.4 int raydium_console_gets (char *where):

DISABLED.
Use `raydium_console_gets_callback` function pointer instead.

31.5 void raydium_console_history_previous (void):

Internal use.

31.6 void raydium_console_history_next (void):

Internal use.

31.7 void raydium_console_history_add (char *str):

Internal use.

31.8 void raydium_console_exec_script (char *file):

Internal use.

31.9 void raydium_console_exec_last_command (void):

Internal use.

31.10 void raydium_console_line_add (char *format, ...):

Mostly reserved for internal use, but unless `raydium_log`, this function will add the provided data only to ingame console, and not to "native" console.

31.11 void raydium_console_event (void):

Internal use. Will switch console up and down.

31.12 void raydium_console_draw (void):

Internal use.

31.13 int raydium_console_internal_isalphanumuscore (char c):

Internal use.

31.14 void raydium_console_complete (char *str):

Internal use.

32 Joysticks, pads and force feedback:

32.1 Introduction:

Raydium supports Joysticks, joypads, steering wheels, force feedback devices, keyboard emulation, for Linux only.

Since API could change during Win32 integration, there is no particular documentation about this subject.

Interesting variables:

```
signed char raydium_joy_button[RAYDIUM_BUTTONS_MAX_BUTTONS];
GLfloat raydium_joy_x;
GLfloat raydium_joy_y;
GLfloat raydium_joy_z;
int raydium_joy;
```

Buttons are booleans, joy x,y and z are -1 <= (x,y,z) <= 1 and 0 means "center".

32.2 void raydium_joy_key_emul (void):

Emulate keyboard (directional pad) with joy, if any.

32.3 void raydium_joy_ff_autocenter (int perc):

Set Force Feedback autocenter factor.

32.4 void raydium_joy_ff_tremble_set (GLfloat period, GLfloat force):

Send tremble effect to Force Feedback device for a determined period,
at a particular force. (no units yet).

33 Graphic User Interfaces:

33.1 Introduction:

Raydium provides a support for simple GUI definitions thru a set of
functions (RayPHP interface is available).
Raydium’s GUI are themable, using ".gui" theme text files. A default "full"
theme is provided as "theme-raydium2.gui" (and suitable ".tga" file) on the
data repository.
Complete informations about theme building are readable in this file.

33.2 Vocabulary:

This API will allow declaeation of:
- "widgets" (label, button, edit box, track bar, check box, combo box)
- "windows" (containers for widgets)

"Focus" is supported for windows and widgets. The final user will not have
any control on windows focus. "Tab" key is used for widget focus cycling.

Widgets and windows are identified by a name or by a unique numeric id.

33.3 Building:

The idea is simple: build a window (position and size), and create
widgets over this window.
All widgets are created using the current sizes (x,y and font). See
suitable function).
Buttons provides a simple callback, and all other widgets (but label)
provides an unified "read" function. Window deletion is also possible.

You must set current theme before any of this operations (see below).

33.4 void raydium_gui_window_init(int window):

Internal use. Will reset `window`.

33.5 void raydium_gui_init(void):

Internal use. Will init all GUI API. Called once by Raydium.

33.6 void raydium_gui_theme_init(void):

Internal use. Will init theme.

33.7 int raydium_gui_theme_load(char *filename):

This function will load and set current theme (".gui" files). You must load
a theme by yourself, since Raydium will never do it for you.
This function must be called before GUI building.

33.8 signed char raydium_gui_window_isvalid(int i):

Mostly internal. Will check if `i` window is valid.

33.9 int raydium_gui_window_find(char *name):

Will search `name` window's numeric id.

33.10 signed char raydium_gui_widget_isvalid(int i, int window):

Mostly internal. Will check if `i` widget of `window` is valid.

33.11 int raydium_gui_widget_find(char *name, int window):

Will search `name` widget numeric id (for `window`).

33.12 void raydium_gui_widget_next(void):

Mostly internal. Cycle focus.

33.13 void raydium_gui_widget_draw_internal(GLfloat *uv, GLfloat *xy):

Internal use. Generic drawing function.

33.14 void raydium_gui_button_draw(int w, int window):

Internal use.

33.15 void raydium_gui_track_draw(int w, int window):

Internal use.

33.16 void raydium_gui_label_draw(int w, int window):

Internal use.

33.17 void raydium_gui_edit_draw(int w, int window):

Internal use.

33.18 void raydium_gui_check_draw(int w, int window):

Internal use.

33.19 void raydium_gui_combo_draw(int w, int window):

Internal use.

33.20 void raydium_gui_window_draw(int window):

Internal use.

33.21 void raydium_gui_draw(void):

Internal use. GUI drawing callback.

33.22 int raydium_gui_button_read(int window, int widget, char *str):

Internal use. Button read accessor (dummy).

33.23 int raydium_gui_label_read(int window, int widget, char *str):

Internal use. Label read accessor (dummy).

33.24 int raydium_gui_track_read(int window, int widget, char *str):

Internal use. Track read accessor.

33.25 int raydium_gui_edit_read(int window, int widget, char *str):

Internal use. Edit read accessor.

33.26 int raydium_gui_check_read(int window, int widget, char *str):

Internal use. Check read accessor.

33.27 int raydium_gui_combo_read(int window, int widget, char *str):

Internal use. Combo read accessor.

33.28 void raydium_gui_show(void):

Will show current built GUI.

33.29 void raydium_gui_hide(void):

Will hide current built GUI. This is the default state.

33.30 signed char raydium_gui_isvisible(void):

Will return current visibility of GUI.

33.31 void raydium_gui_window_delete(int window):

Will delete `window`. No further access to widgets is possible.

33.32 void raydium_gui_window_delete_name(char *window):

Same as above, but using `window`'s name.

33.33 void raydium_gui_widget_sizes(GLfloat sizex, GLfloat sizey, GLfloat font_size):

Each widget is created using 3 size: X size, Y size and font size. This
function will allow you to set all sizes for a widget or a group of widget.
Unit: percents (screen)

33.34 int raydium_gui_window_create(char *name, GLfloat px, GLfloat py, GLfloat sizex, GLfloat sizey):

Obviously, this function will create a new window. This window will take focus
and overlap any previous window.
`px` and `py` for X and Y position on the screen, and `sizex` and `sizey`
for sizes, obviously.
Unit: percents (screen)

33.35 int raydium_gui_internal_object_create(char *name, int window, signed char type, GLfloat px, GLfloat py, GLfloat sizex, GLfloat sizey, GLfloat font_size):

Internal use.

33.36 int raydium_gui_button_create(char *name, int window, GLfloat px, GLfloat py, char *caption, void *OnClick?):

This function will create a new button, with `name` and with `window` for parent.
You need to provide a `caption` ("title") and a OnClick? callback function.
This callback must follow this prototype:

```
void btnButtonClick(raydium_gui_Object *w)
```

You can find `raydium_gui_Object` structure declaration in `raydium/gui.h`, if needed.

Unit for position (`px` and `py`): percents (window)

33.37 int raydium_gui_button_create_simple(char *name, int window, GLfloat px, GLfloat py, char *caption):

Same as above, but no OnClick? callback function is asked. This type of button is "readable" thru `raydium_gui_button_clicked`.

33.38 int raydium_gui_label_create(char *name, int window, GLfloat px, GLfloat py, char *caption, GLfloat r, GLfloat g, GLfloat b):

This function will create a new label, with `name` and with `window` for parent.
You need to provide a `caption` ("title") and an RGB color (0..1 interval)

Unit for position (`px` and `py`): percents (window)

33.39 int raydium_gui_track_create(char *name, int window, GLfloat px, GLfloat py, int min, int max, int current):

This function will create a new trackbar, with `name` and with `window` for parent.
You need to provide a `min` interger value, a `max` and `current` value.

Unit for position (`px` and `py`): percents (window)

33.40 int raydium_gui_edit_create(char *name, int window, GLfloat px, GLfloat py, char *default_text):

This function will create a new edit box, with `name` and with `window` for parent.
You may provide a default text (or an empty string), if needed. Unless all others Raydium's data, max string length is `RAYDIUM_GUI_DATASIZE` and not `RAYDIUM_MAX_NAME_LEN`, since this component may handle bigger strings.
See `raydium/gui.h` for more informations.

Unit for position (`px` and `py`): percents (window)

33.41 int raydium_gui_check_create(char *name, int window, GLfloat px, GLfloat py, char *caption, signed char checked):

This function will create a new check box, with `name` and with `window`
for parent.
You need to provide a `caption` ("title") and a boolean state (checked or not).

Unit for position (`px` and `py`): percents (window)

33.42 int raydium_gui_combo_create(char *name, int window, GLfloat px, GLfloat py, char *items, int current):

This function will create a new edit box, with `name` and with `window`
for parent.
`items` is a string, using '\n' as a separator. It's allowed to create an
empty item.
`current` is the default selected item in `items`. (first = 0)
Unless all others Raydium's data, max string length is `RAYDIUM_GUI_DATASIZE`
and not `RAYDIUM_MAX_NAME_LEN`, since this component may handle bigger
strings. See `raydium/gui.h` for more informations.

Unit for position (`px` and `py`): percents (window)

33.43 int raydium_gui_read(int window, int widget, char *str):

Use this function to get `widget`'s state (for `window`).
This function will always return this information thru two variable:
an integer (returned value) and a string (`str`).
This information is specific to `widget`'s type (checked or not for a
checkbox, current choice for a combo, current string for an edit box, ...)
Please, note `str` must be allocated before function call. This is also
the case for PHP scripts :

```
$str=str_pad("",256); // "pre-alloc"
$val=raydium_gui_read_name("main","track",$str);
echo "value=$val, string='$str'";
```

33.44 int raydium_gui_read_name(char *window, char *widget, char *str):

Same as above, but `window` and `widget` are resolved thru names, and
not numeric id.

33.45 int raydium_gui_button_clicked(void):

This function will return the id of the last clicked button,
or -1 if none were clicked.
The id is built like this : `window * 1000 + widget_id`
Usefull for PHP scripts, since it's not possible to create callback for
buttons with RayPHP.

34 Data registration:

34.1 Introduction:

Raydium supports scripting, for example using PHP in the current implementation.
All `raydium_register_*` functions are provided as a "bridge" between
your applications and PHP scripts, allowing you to "export" native variables
and functions to PHP scripts.
For more informations, see PHP chapters.

34.2 int raydium_register_find_name (char *name):

Lookups a variable by `name`. Search is not possible (yet) for
registered functions.
Mostly used internally.

34.3 signed char raydium_register_name_isvalid (char *name):

Tests `name`, and returns his viability as a boolean.
Accepted intervals for variables and functions: [a-z], [A-Z] and '_'
Numerics are not allowed.

34.4 int raydium_register_variable (void *addr, int type, char *name):

Will register a new variable. You must provide variable's address (`addr`),
`type` and `name`.
Current available types are: `RAYDIUM_REGISTER_INT`, `RAYDIUM_REGISTER_FLOAT`,
and `RAYDIUM_REGISTER_STR`.

34.5 int raydium_register_variable_const_f(float val, char *name):

Will register a new `float` constant.

34.6 int raydium_register_variable_const_i(int val, char *name):

Will register a new `int` constant.

34.7 void raydium_register_variable_unregister_last (void):

Variable are registered on a stack. As you may want to create "temporary"
variables (usefull for building script's arguments, for example), this function
allows you to unregister last registered variable. Multiple calls are possible.

34.8 int raydium_register_modifiy (char *var, char *args):

Deprecated.

34.9 void raydium_register_function (void *addr, char *name):

Will register a function. You only need to provide an address (`addr`)
and a name.

34.10 void raydium_register_dump (void):

Will dump to console all registered variables and functions.

35 Profiling (sort of ...):

35.1 Introduction:

You will find here a few functions for a very simple profiling.
For anything else than a quick time measure, use real profiling tools.
Note: Use only one "profiler" at a time.

35.2 void raydium_profile_start(void):

Starts measure.

35.3 void raydium_profile_end(char *tag):

Stops measure and displays result using `tag` string.

36 RayPHP (internals):

36.1 Introduction:

Raydium also use RayPHP (Raydium/PHP interface) for its own needs.
For PHP part of these functions, see "rayphp/" directory.
So far, RayPHP is dedicated to R3S (Raydium Server Side Scripts) access.
All this is mostly usefull for internal uses, since Raydium provides `fopen` wrappers, thru `raydium_file_fopen`.

36.2 int raydium_rayphp_repository_file_get (char *path):

Will contact R3S servers for downloading `path` file.

36.3 int raydium_rayphp_repository_file_put (char *path, int depends):

Will contact R3S servers for uploading `path` file. Set `depends` to true (1) if you also want to upload dependencies, false (0) otherwise.

36.4 int raydium_rayphp_repository_file_list(char *filter):

Will contact R3S servers to get file list, using `filter` (shell-like syntax). Default `filter` is `*`.

37 Text file parsing:

37.1 Introduction:

Raydium provides a set of functions dedicated to text files parsing. These files must follow a simple syntax:

```
// strings
variable_s="string value";

// float (or integer, i.e.)
variable_f=10.5;

// float array
variable_a={1,2,10.5,};

// raw data
variable_r=[
xxxxxxxx
#  oo  #
#      #
#  oo  #
xxxxxxxx
];
```

Semi-colon are purely esthetic.

37.2 void raydium_parser_trim (char *org):

Strip whitespace (or other characters) from the beginning and end of a string.
So far, ’ ’, ’\n’ and ’;’ are deleted.

37.3 signed char raydium_parser_isdata (char *str):

Returns true (1) if `str` contains data, false (0) otherwise (comments and blank lines).

37.4 signed char raydium_parser_cut (char *str, char *part1, char *part2, char separator):

This function will cut `str` in two parts (`part1` and `part2`) on `separator`. No memory allocation will be done by this functions.
First occurence of `separator` is used (left cut).
Return true (`i`+1) if `str` was cut, where `i` is the separator position.

37.5 void raydium_parser_replace (char *str, char what, char with):

Will replace all occurence of `what` with `with`.

37.6 int raydium_parser_read (char *var, char *val_s, GLfloat *val_f, int *size, FILE *fp):

Reads a new data line in `fp`.
`var` will contain variable name. You’ll find associated value in `val_s` if it’s a string, or `val_f` if it’s a float (or a float array). In this last case, `size` will return the number of elements if the array.

```
FILE *fp;
int ret;
char var[RAYDIUM_MAX_NAME_LEN];
char val_s[RAYDIUM_MAX_NAME_LEN];
GLfloat val_f[MY_ARRAY_SIZE];
int size;

fp=raydium_file_fopen("foobar.txt","rt");

while( (ret=raydium_parser_read(var,val_s,val_f,&size,fp))!=RAYDIUM_PARSER_TYPE_EOF)
{
if(!strcasecmp(var,"foobar_variable"))
{
if(ret!=RAYDIUM_PARSER_TYPE_FLOAT || size!=2)
{
raydium_log("error: foobar_variable is not float array");
continue;
}
memcpy(...);
}

...

}
```

38 Live textures and videos API:

38.1 Introduction:

Live API provides two distinct features:

1 - It provides an easy way to create and manage dynamic textures, since you just have to provide a pointer to your image data, and call suitable function each time this image is changing.

2 - This API also supports video4linux (aka V4L), as an extension of the Live API. The main goal is to link a video4linux device (webcam, tv card, ...) to a texture. A callback is also available if you want to get (and transform) data of every capture.

You'll find detailed informations for each domain below.

38.2 Color conversion:

Live API used to work with RGB and RGA color formats. Since some V4L devices use other patterns, Live API needs conversion functions. You've no need to do color conversion by yourself, consider all this as internal functions.

38.3 void v4l_copy_420_block (int yTL, int yTR, int yBL, int yBR, int u, int v, int rowPixels, unsigned char *rgb, int bits):

YUV420P block copy.
This code is not native.

38.4 int v4l_yuv420p2rgb (unsigned char *rgb_out, unsigned char *yuv_in, int width, int height, int bits):

YUV420P to RGB conversion.
This code is not native.

38.5 Live Video API:

This part of the Live API id dedicated to video devices. For now, the support is limited to Linux thru V4L API. Every V4L compatible device should work with Live Video, but for any advanced setup of your video device (tuner configuration, source, FPS, ...), you must use an external tool.
By default, Live API supports up to 4 simultaneous devices.

38.6 signed char raydium_live_video_isvalid(int i):

Internal use, but you can call this function if you want to verify if a live video device id is valid (in bounds, open, and ready to capture).

38.7 int raydium_live_video_find_free(void):

Internal use.
Finds a free live video device slot.

38.8 int raydium_live_video_open(char *device, int sizex, int sizey):

This is where you should start. This function opens `device` (something like "/dev/video0"), requestion `sizex` x `sizey` resolution.
If `device` is `RAYDIUM_LIVE_DEVICE_AUTO`, Raydium will use a default device, hardcoded or given thru commande line (`--video-device`).
Same story for sizes, with `RAYDIUM_LIVE_SIZE_AUTO`.
This function will try to detect a compatible palette (grayscale, rgb, yuv420p, with 4, 6, 8, 15, 16 and 24 bits per pixel) and capture method (`read()` or `mmap()`).
Returns -1 in case or error, device id otherwise.

38.9 int raydium_live_video_open_auto(void):

Same as above, but with full autodetection.

38.10 int raydium_live_video_read(raydium_live_Device *dev):

Internal V4L read function.

38.11 void raydium_internal_live_video_callback(void):

internal frame callback.

38.12 Live API Core:

the main goal of the Live API is to allow you to create your own dynamic textures. The first method is to provide your own picture data thru a pointer, the second method is to use a Live Video device (see above) as data source.

38.13 void raydium_internal_live_close(void):

Internal close function.

38.14 void raydium_live_init(void):

Internal init function.

38.15 signed char raydium_live_texture_isvalid(int i):

Internal use, but you can call this function if you want to verify if a live texture id is valid (in bounds, open, and ready to capture).

38.16 int raydium_live_texture_find_free(void):

Internal use.
Finds a free live texture slot.

38.17 int raydium_live_texture_find(int original_texture):

Resolvs `original_texture` id (native Raydium texture id) to a
live texture id, if any.

38.18 int raydium_live_texture_create(char *as, unsigned char *data_source, int tx, int ty, int bpp):

Create a new Live Texture with `as` name. You must provide a `data_source`
with RGB or RGBA format, with `tx` and `ty` size.
Possible bpp values are 24 (RGB) and 32 (RGBA).
Returns the live texture id, or -1 when it fails.

38.19 int raydium_live_texture_video(int device_id, char *as):

This is another way to create a Live Texture, but using a Live Video device
for data source. Provide texture name (`as`) and Live `device_id`.

38.20 void raydium_live_texture_refresh(int livetex):

When your data source have changed, call this function to refresh new
data to hardware. Obviously, this function is useless for Live Video textures
since Raydium will automatically refresh data.

38.21 void raydium_live_texture_refresh_name(char *texture):

Same as above, but using `texture` name.

38.22 void raydium_live_texture_refresh_callback_set(int livetex, void *callback):

You can create a "OnRefresh?" callback for any Live Texture (`livetex` is an
id to this texture). This is mostly usefull for Live Video texture.
Your callback must follow this prototype :
`int refresh_callback(unsigned char *data, int tx, int ty, int bpp)`
You have full write access to `data`, allowing you to draw over
the provided picture (warning: for non video Live textures, `data` pointer
is not owned by Raydium and may be "read only")
You must return 1 to confirm data flushing, or 0 to cancel this refresh.

38.23 void raydium_live_texture_refresh_callback_set_name(char *texture, void *callback):

Same as above, but using `texture` name.

38.24 void raydium_live_texture_mask(int livetex, GLfloat alpha):

This function will draw a fullscreen mask using `livetex` Live Texture id and
`alpha` opacity (0 means transparent, 1 means fully opaque, allowing any
intermediate value). Use this function at any place of your rendering
function AFTER camera call and obviously before `raydium_rendering_finish`.

38.25 void raydium_live_texture_mask_name(char *texture, GLfloat alpha):

Same as above, but using `texture` name.

39 Integrated Physics (ODE):

39.1 Introduction:

Raydium allows you to build applications with full physics, using ODE (Open Dynamics Engine). ODE is "an open source, high performance library for simulating rigid body dynamics", and is fully integrated into Raydium, with the usual abstraction. You can build cars, ragdolls, rockets, ... with only few lines of code. Physics are linked to sound API, particles engine, network layer, ... so you’ve almost nothing else to do but setting up objects.

Raydium’s website provides tutorials for building physics ready applications.

39.2 Vocabulary:

Raydium physics use a simple vocabulary, with a few entities :
- Objects:
Objects are containers, with no direct visual appearance. An object contains elements and joints (see below). By default, all elements in an object doesn’t collide each others. "Car", "Player", "Crane" are good object examples.

- Elements:
Elements are the main thing you will play with. An element is rendered using an associated 3D mesh, is configured with a geometry, density, a size, collides with others elements, ...
An element must be owned by an object.
For now, there is 3 element types (standard, satic, fixing). Static elements are unmovable, they just collide with other elements, usefull for very big elements, or externally controlled elements (motion capture, network, haptic interface, ...), for example.
Raydium supports boxes and spheres.

- Joints:
Joints are dedicated to elements linking. A joint must be linked with two elements or unwanted behaviors may happen.
For now, Raydium supports 4 joint types (hinge, hinge2, universal, fixed), and you will find more informations with suitable functions documentation, below.
On a joint, you could setup limits (min and max for all axes) and a maximum force before joint breaks, if needed.

- Motors:
A motor is linked to joints, and may powering an unlimited amount of joints.
For now, 3 motor types are available: engine, angular and rocket.

Engine type works the same way as a car’s engine: it will try to make "something" turning, at the desired speed. You can link a gearbox to this type (and only this one).

Angular type will try to rotate "something" to the desired angle, usefull for car’s front wheels, for example.

Rocket type is very simple: give a force and an orientation. Usefull for creating copters, rockets, and for elements "pushing", for example.
Special rocket is avaiblable for FPS style player controls.
Warning, a rocket is linked to an element ! (not a joint)

- Explosions:
Explosions are not directly related to rigid body physics, but consider it as a high level primitive.
With Raydium, you have two different ways to create an explosion.

First, you can create a "blowing explosion", generating a spherical blow. Any element in this growing sphere will be ejected.
Use this for very consequent explosions only !

Next, you can create an instantaneous explosion, with a degressive blowing effect. A force is applied to every body found inside the blowing radius, proportional to distance from the explosion’s center. Usefull for smaller explosions.

- Launchers:
Launchers are not real entities, but "only" tools. Obviously, they are allowing you to launch an element (you must provice force and orientation) from another element (relatively). More informations about launchers below.

39.3 Callbacks:

For advanced uses, you may want to enter into some "internal" parts of RayODE. Many callbacks are available for such needs.
To cancel any callback, set its value to `NULL` (default value).
Here is a quick list:

- `raydium_ode_StepCallback`
This callback is fired before every ODE callback. Since physcis callback frequency may change (see `raydium_ode_time_change`) during slow motion scenes, for example, this callback is quiet useful.
Callback prototype: `void f(void);`

- `raydium_ode_ObjectNearCollide`
When two objects are too near, before lauching "expensive" collision tests, Raydium is firing this event.

Callback prototype: `signed char f(int obj1, int obj2);`
`obj1` and `obj2` are the two objets, and you must return true (1) if you want to "validate" collision, or false (0) if you don’t want that two objects to collide.

- `raydium_ode_CollideCallback`

When two objects collides, Raydium will search all collisions between every elements. For each contact, this callback is fired. For complex objects, with a lot of elements, this callback may be fired a very large number of times during one ODE step ! Do only simple things here.

Callback prototype: `signed char f(int e1, int e2, dContact *c);`
`e1` and `e2` are the two colliding elements, and you must return true (1) if you want to "validate" this contact, or false (0) to cancel this contact (and only this one !)

See ODE documentation, chapter 7.3.7, for more informations about `dContact` structure.

- `raydium_ode_ExplosionCallback`

At every explosion, of any type, this event is fired. This is the best place to play suitable sound, create particles and such things.

Callback prototype: `void f(signed char type, dReal radius, dReal force_or_propag, dReal *pos);`
You can find in callback params:
explosion `type` (see above), `radius`, force or propag (depending on explosion type) and `pos`, an array of 3 dReal values for explosion position.

- `raydium_ode_BeforeElementDrawCallback`

When `raydium_ode_draw_all(0)` is called, for every element to draw, this callback is before element drawing.

Callback prototype: `signed char f(int elem);`
`elem` is the element'id. Return true (1) if you want to draw this element, or false (0) otherwise. This is also the best place to drawn team colors on cars, for example (see `raydium_rendering_rgb_force` for this use).

- `raydium_ode_AfterElementDrawCallback`

Same as the previous callback, but after element drawing.

Callback prototype: `void f(int elem);`
With the previous example (team colors), this is the place to restore default rendering state (see `raydium_rendering_rgb_normal`).

39.4 Miscallenous:

By default, ODE is called 400 times per second, allowing very accurate physics. You may change this in `ode.h` with `RAYDIUM_ODE_PHYSICS_FREQ` and `RAYDIUM_ODE_TIMESTEP`, but most ERP and CFM values must be changed in your applications. ODE use a lot of cache mechanisms, so 400 Hz is a reasonable value.

Please note RayODE interface is using `dReal` ODE type for variables.

For now, `dReal` is an alias to `float` type. But please use `sizeof()`.

Raydium provides some other functions for advanced uses, and you can access directly to ODE API for very experienced users.

See also the ODE documentation: http://opende.sourceforge.net/ode-latest-userguide.html

39.5 void raydium_ode_name_auto (char *prefix, char *dest):

This function will generate a single name, using `prefix`. The generated name is stored at `dest` address. No memory allocation is done.
Example : `raydium_ode_name_auto("prefix",str)` may generate something like `prefix_ode_0`.

39.6 void raydium_ode_init_object (int i):

Will initialize (or erase) object `i`. Mostly for internal uses.

39.7 void raydium_ode_init_element (int i):

Will initialize (or erase) element `i`. Mostly for internal uses.

39.8 void raydium_ode_init_joint (int i):

Will initialize (or erase) joint `i`. Mostly for internal uses.

39.9 void raydium_ode_init_motor (int i):

Will initialize (or erase) motor `i`. Mostly for internal uses.

39.10 void raydium_ode_init_explosion (int e):

Will initialize (or erase) spherical explosiion `i`. Mostly for internal uses.

39.11 void raydium_ode_init (void):

Will initialize all RayODE interface. Never call this function by yourself.

39.12 signed char raydium_ode_object_isvalid (int i):

Will return 0 (false) if object `i` is not valid (free slot or out of bounds) or 1 (true) otherwise.

39.13 signed char raydium_ode_element_isvalid (int i):

Will return 0 (false) if element `i` is not valid (free slot or out of bounds) or 1 (true) otherwise.

39.14 signed char raydium_ode_joint_isvalid (int i):

Will return 0 (false) if joint `i` is not valid (free slot or out of bounds) or 1 (true) otherwise.

39.15 signed char raydium_ode_motor_isvalid (int i):

Will return 0 (false) if motor `i` is not valid (free slot or out of bounds) or 1 (true) otherwise.

39.16 signed char raydium_ode_explosion_isvalid (int i):

Will return 0 (false) if explosion `i` is not valid (free slot or out of bounds)
or 1 (true) otherwise.

39.17 void raydium_ode_ground_dTriArrayCallback (dGeomID TriMesh?, dGeomID RefObject?, const int *TriIndices?, int TriCount?):

Internal. Unsupported.

39.18 int raydium_ode_ground_dTriCallback (dGeomID TriMesh?, dGeomID RefObject?, int TriangleIndex?):

Internal. Unsupported.

39.19 void raydium_ode_ground_set_name (char *name):

`ground` is a primitive for RayODE interface. You only have to set ground
mesh `name` (.tri file). You may call this function many times, if needed,
switching from one ground to another on the fly.
Warning: triangle normals are very important for ground models !

39.20 int raydium_ode_object_find (char *name):

Resolves object id from it’s `name`.

39.21 int raydium_ode_element_find (char *name):

Resolves element id from it’s `name`.

39.22 int raydium_ode_joint_find (char *name):

Resolves joint id from it’s `name`.

39.23 int raydium_ode_motor_find (char *name):

Resolves motor id from it’s `name`.

39.24 int raydium_ode_explosion_find (char *name):

Resolves explosion id from it’s `name`.

39.25 int raydium_ode_object_create (char *name):

Will build a new object with `name`. Returns new object id, or -1 when
it fails.

39.26 signed char raydium_ode_object_rename (int o, char *newname):

Will rename object o with a `newname`.

39.27 signed char raydium_ode_object_rename_name (char *o, char *newname):

Same as above, but from object’s name (o).

39.28 signed char raydium_ode_object_colliding (int o, signed char colliding):

By default, all elements from an object are not colliding each others.

The only exception is for `GLOBAL` object.
If you want to change this behaviour for o object, sets `colliding`
to 1 (true). 0 (false) sets back to default behaviour (no internal collisions).

39.29 signed char raydium_ode_object_colliding_name (char *o, signed char colliding):

Same as above, but using object's name.

39.30 void raydium_ode_object_linearvelocity_set (int o, dReal * vect):

Sets linear velocity for all elements of object o. Velocity is sets thru
`vect`, a 3 x dReal array.
Use with caution, setting an arbitrary linear velocity may cause unwanted
behaviours.

39.31 void raydium_ode_object_linearvelocity_set_name (char *o, dReal * vect):

Same as above, but using object's name.

39.32 void raydium_ode_object_linearvelocity_set_name_3f (char *o, dReal vx, dReal vy, dReal vz):

Same as above, but using 3 dReal values.

39.33 void raydium_ode_object_addforce (int o, dReal * vect):

Add force `vect` to all elements of object o.
Force is sets thru `vect`, a 3 x dReal array.
Prefer this method to `..._linearvelocity_set...` functions.

39.34 void raydium_ode_object_addforce_name (char *o, dReal * vect):

Same as above, but using object's name.

39.35 void raydium_ode_object_addforce_name_3f (char *o, dReal vx, dReal vy, dReal vz):

Same as above, but using 3 dReal values.

39.36 void raydium_ode_element_addforce (int e, dReal * vect):

Adds force `vect` to element e.
Force is sets thru `vect`, a 3 x dReal array.

39.37 void raydium_ode_element_addforce_name (char *e, dReal * vect):

Same as above, but using element's name.

39.38 void raydium_ode_element_addforce_name_3f (char *e, dReal vx, dReal vy, dReal vz):

Same as above, but using 3 dReal values.

39.39 void raydium_ode_element_addtorque (int e, dReal * vect):

Adds torque `vect` to element e.

Torque is sets thru `vect`, a 3 x dReal array.

39.40 void raydium_ode_element_addtorque_name (char *e, dReal * vect):

Same as above, but using element’s name.

39.41 void raydium_ode_element_addtorque_name_3f (char *e, dReal vx, dReal vy, dReal vz):

Same as above, but using 3 dReal values.

39.42 signed char raydium_ode_element_material (int e, dReal erp, dReal cfm):

When two elements collides, there’s two important parameters used for
contact point generation : ERP and CFM.
ERP means "Error Reduction Parameter", and its value is between 0 and 1 and
CFM means "Constraint Force Mixing".
Changing ERP and CFM values will change the way this element collides with
other elements, providing a "material" notion.
Raydium provides a few default values, see `RAYDIUM_ODE_MATERIAL_*` defines
in `raydium/ode.h` file (hard, medium, soft, soft2, default, ...).

For more informations, see ODE documentation, chapters 3.7 and 3.8.

39.43 signed char raydium_ode_element_material_name (char *name, dReal erp, dReal cfm):

Same as above, but using element’s name.

39.44 signed char raydium_ode_element_slip (int e, dReal slip):

Slip parameter is a complement of material values (ERP and CFM, see above).
Raydium provides a few default values, see `RAYDIUM_ODE_SLIP_*` defines
in `raydium/ode.h` file (ice, player, normal, default).

39.45 signed char raydium_ode_element_slip_name (char *e, dReal slip):

Same as above, but using element’s name.

39.46 signed char raydium_ode_element_rotfriction (int e, dReal rotfriction):

rotfriction stands for "Rotation Friction", "Rolling Friction",
"Damping Effect", ...
With RayODE, by default, when a sphere element is rolling over a flat ground,
it will roll forever. Applying a rotfriction factor will solve this.
A value of 0 will disable rotation friction.
Example:

```
#define ROTFRICTION     0.0005
raydium_ode_element_rotfriction(elem,ROTFRICTION);
```

39.47 signed char raydium_ode_element_rotfriction_name (char *e, dReal rotfriction):

Same as above, but using element’s name.

39.48 dReal *raydium_ode_element_linearvelocity_get (int e):

Returns a pointer to element’s linear velocity. Linear velocity is an array of 3 x dReal.

Example:

```
dReal *p;
p=raydium_ode_element_linearvelocity_get(elem);
raydium_log("%f %f %f",p[0],p[1],p[2]);
```

Returned data is available only for the current frame.

39.49 dReal *raydium_ode_element_linearvelocity_get_name (char *e):

Same as above, but using element’s name.

39.50 void raydium_ode_element_OnBlow (int e, void *OnBlow?):

During an instantaneous explosion, all elements inside the blow’s radius may fire an OnBlow? callback (event), if set.

`OnBlow?` callback must follow this prototype :

`void blow_touched(int elem, dReal force, dReal max_force)`

`elem` is the element id.

`force` is the amount of force received from explosion.

`max_force` is the amount of force at the core of the explosion.

Sets `OnBlow?` to `NULL` if you want to disable this callback.

39.51 void raydium_ode_element_OnBlow_name (char *e, void *OnBlow?):

Same as above, but using element’s name.

39.52 void raydium_ode_element_OnDelete (int e, void *OnDelete?):

OnDelete? callback is fired when someone or something tries to delete an element. This callback can cancel deletion, if needed.

`OnBlow?` callback must follow this prototype :

`int element_delete(int elem)`

`elem` is the element id. Return `1` (true) to confirm deletion, of `0` to cancel.

Sets `OnDelete?` to `NULL` if you want to disable this callback.

39.53 void raydium_ode_element_OnDelete_name (char *e, void *OnDelete?):

Same as above, but using element’s name.

39.54 void raydium_ode_element_gravity (int e, signed char enable):

By default, gravity applies to every element of the scene. If you want to disable gravity for element `e`, set `enable` to `0` (false).

You can restore gravity with `enable` sets to `1` (true).

39.55 void raydium_ode_element_gravity_name (char *e, signed char enable):

Same as above, but using element’s name.

39.56 void raydium_ode_element_ttl_set (int e, int ttl):

TTL means Time To Live. Setting a TTL on an element will automatically delete this element when TTL expires.

- TTL unit: `ttl` is given in ODE steps (see example, below).
- TTL deletion may be canceled by OnDelete? callback (see above).
- TTL may be changed on the fly, at anytime.
- a `ttl` value of -1 will disable TTL.

example:

```
raydium_ode_element_ttl_set(elem,RAYDIUM_ODE_PHYSICS_FREQ*5); // 5 seconds
```

39.57 void raydium_ode_element_ttl_set_name (char *e, int ttl):

Same as above, but using element’s name.

39.58 signed char raydium_ode_element_aabb_get (int element, dReal * aabb):

AABB means Axis-Aligned Bounding Box. This function will return element’s bounding box on X, Y and Z axis.

`aabb` is a pointer to an array of 6 x dReal.
No memory allocation is done.
Will return 0 (false) in case of failure.

39.59 signed char raydium_ode_element_aabb_get_name (char *element, dReal * aabb):

Same as above, but using element’s name.

39.60 int raydium_ode_element_touched_get (int e):

Every element provide a "touched" flag. If element e is touching anything, this function will return 1 (true).

39.61 int raydium_ode_element_touched_get_name (char *e):

Same as above, but using element’s name.

39.62 signed char raydium_ode_element_player_set (int e, signed char isplayer):

RayODE provides a special behaviour for FPS style players, also named "standing geoms". The idea is simple : a player element is always upright, and you can set an arbitrary rotation angle around Z axis anytime. Sets `isplayer` to 1 (true) to transform element e into a "player element".

39.63 signed char raydium_ode_element_player_set_name (char *name,

signed char isplayer):

Same as above, but using element’s name.

39.64 signed char raydium_ode_element_player_get (int e):

Returns if element e is a "player element" (1, true) or not (0, false).
See above for more informations about player elements.

39.65 signed char raydium_ode_element_player_get_name (char *name):

Same as above, but using element’s name.

39.66 signed char raydium_ode_element_player_angle (int e, dReal angle):

Sets "standing geom" Z rotation `angle` (radian) for element e.
See above for more informations about player elements.

39.67 signed char raydium_ode_element_player_angle_name (char *e, dReal angle):

Same as above, but using element’s name.

39.68 int raydium_ode_element_ground_texture_get (int e):

Unsupported. Do not use for now.

39.69 int raydium_ode_element_ground_texture_get_name (char *e):

Unsupported. Do not use for now.

39.70 int raydium_ode_element_object_get (int e):

Since every element is owned by an object, this function will return
the owner’s object id.

39.71 int raydium_ode_element_object_get_name (char *e):

Same as above, but using element’s name.

39.72 int raydium_ode_object_sphere_add (char *name, int group, dReal mass, dReal radius, signed char type, int tag, char *mesh):

This function will add an new "sphere" element to an object (`group`).
You must provide:
- `name`: single name for this new element.
- `group`: owner object id.
- `mass`: density of this new element. Mass will depend on radius.
- `radius`: radius of the element sphere geometry. Raydium is able to detect this value with `RAYDIUM_ODE_AUTODETECT`. Things like `RAYDIUM_ODE_AUTODETECT*2` are ok, meaning "twice the detected radius".
- `type`: `RAYDIUM_ODE_STANDARD` or `RAYDIUM_ODE_STATIC` (collide only, no physics).
- `tag`: use this integer value as you want. The important thing is that this value is sent to network, and will be available on every connected peer. This tag must be greater or equal to 0. Suitable functions are available to read back this value later on an element.

- `mesh`: 3D model used for rendering this element. Use an empty string to disable rendering (and not `NULL` !).

39.73 int raydium_ode_object_box_add (char *name, int group, dReal mass, dReal tx, dReal ty, dReal tz, signed char type, int tag, char *mesh):

This function will add an new "box" element to an object (`group`).
Arguments are the same as `raydium_ode_object_sphere_add` (see above) but `tx`, `ty` and `tz`, used for box sizes. As for spheres, you can use `RAYDIUM_ODE_AUTODETECT`. Give this value only for `tx`, this will automatically apply to `ty` and `tz`.
Again, Things like `RAYDIUM_ODE_AUTODETECT*2` are ok, meaning "twice the detected size".

39.74 int raydium_ode_element_fix (char *name, int *elem, int nelems, signed char keepgeoms):

Experimental code.

The idea here is to create a bounding single element for a group of elements.
You must provide:
- `name`: the new bounding element’s name.
- `elems`: an array of all elements to fix (id array).
- `nelems`: the number of elements in `elems` array.
- `keepgeoms`: set to `0`.

You can only fix standard elements (no statics) and all elements must be owned by the same object.

39.75 void raydium_ode_element_unfix (int e):

Experimental code. Unimplemented, yet.
Symmetric function, see `raydium_ode_element_fix`.

39.76 void raydium_ode_element_move (int elem, dReal * pos):

This function will move element `elem` to `pos`.
`pos` is a dReal array of 3 values (x,y,z).
Warning: arbitrary moves may lead to unwanted behaviours.

39.77 void raydium_ode_element_move_name (char *name, dReal * pos):

Same as above, but using element’s name.

39.78 void raydium_ode_element_move_3f(int elem, dReal x,dReal y, dReal z):

Same as `raydium_ode_element_move`, but using 3 dReal values.

39.79 void raydium_ode_element_move_name_3f (char *name, dReal x, dReal y, dReal z):

Same as above, but using element’s name.

39.80 void raydium_ode_element_rotate (int elem, dReal * rot):

This function will rotate element `elem` using `rot`.
`rot` is a dReal array of 3 values (rx,ry,rz), in radians.
Warning: arbitrary rotations may lead to unwanted behaviours.

39.81 void raydium_ode_element_rotate_3f (int elem, dReal rx, dReal ry, dReal rz):

Same as `raydium_ode_element_rotate`, but using 3 dReal values.

39.82 void raydium_ode_element_rotate_name (char *name, dReal * rot):

Same as `raydium_ode_element_rotate`, but using element's name.

39.83 void raydium_ode_element_rotateq (int elem, dReal * rot):

This function will rotate element `elem` using `rot` quaternion.
`rot` is a dReal array of 4 values.
Warning: arbitrary rotations may lead to unwanted behaviours.

39.84 void raydium_ode_element_rotateq_name (char *name, dReal * rot):

Same as `raydium_ode_element_rotateq`, but using element's name.

39.85 void raydium_ode_element_rotate_name_3f (char *name, dReal rx, dReal ry, dReal rz):

Same as `raydium_ode_element_rotate_name`, but using 3 dReal values.

39.86 void raydium_ode_object_rotate(int obj, dReal *rot):

This function will try to rotate object `obj`.
For now, rotation is done around the last element of the object.
`rot` is a dReal array of 3 values (rx,ry,rz), in radians.
Warning: arbitrary rotations may lead to unwanted behaviours.

39.87 void raydium_ode_object_rotate_name(char *obj, dReal *rot):

Same as above, but using object's name.

39.88 void raydium_ode_object_rotate_name_3f(char *obj, dReal rx, dReal ry, dReal rz):

Same as above, but using 3 dReal values.

39.89 void raydium_ode_object_move (int obj, dReal * pos):

This function will move object `obj` to `pos`.
Obviously, every element of object will be moved.
`pos` is a dReal array of 3 values (x,y,z).
Warning: arbitrary moves may lead to unwanted behaviours.

39.90 void raydium_ode_object_move_name (char *name, dReal * pos):

Same as above, but using object's name.

39.91 void raydium_ode_object_move_name_3f (char *name, dReal x, dReal y, dReal z):

Same as above, but using 3 dReal values.

39.92 void raydium_ode_object_rotateq (int obj, dReal * rot):

This function will try to rotate object `obj` using `rot` quaternion.
For now, rotation is done around the last element of the object.
`rot` is a dReal array of 4 values.
Warning: arbitrary rotations may lead to unwanted behaviours.

39.93 void raydium_ode_object_rotateq_name (char *obj, dReal * rot):

Same as above, but using object’s name.

39.94 void raydium_ode_element_rotate_direction (int elem, signed char Force0OrVel1?):

This function will rotate element `elem` from its force or velocity.
Sets `Force0OrVel1?` to `0` if you want to align element using its
force or `1` using its linear velocity.
Warning: arbitrary rotations may lead to unwanted behaviours.

39.95 void raydium_ode_element_rotate_direction_name (char *e, signed char Force0OrVel1?):

Same as above, but using element’s name.

39.96 void raydium_ode_element_data_set (int e, void *data):

You may want to link some data to any element. If so, use this function
and provide a pointer to your `data` for element e.

39.97 void raydium_ode_element_data_set_name (char *e, void *data):

Same as above, but using element’s name.

39.98 void *raydium_ode_element_data_get (int e):

This function will return a pointer to your linked data, if any (see above).

39.99 void *raydium_ode_element_data_get_name (char *e):

Same as above, but using element’s name.

39.100 int raydium_ode_element_tag_get (int e):

When you create a new element, you must provide a "tag". This function
allows you to get back the tag’s value, even on "distant" elements.

39.101 int raydium_ode_element_tag_get_name (char *e):

Same as above, but using element’s name.

39.102 void raydium_ode_joint_suspension (int j, dReal erp, dReal cfm):

ERP means "Error Reduction Parameter", and its value is between 0 and 1 and
CFM means "Constraint Force Mixing".
Changing ERP and CFM values will change joint energy absorption and restitution.

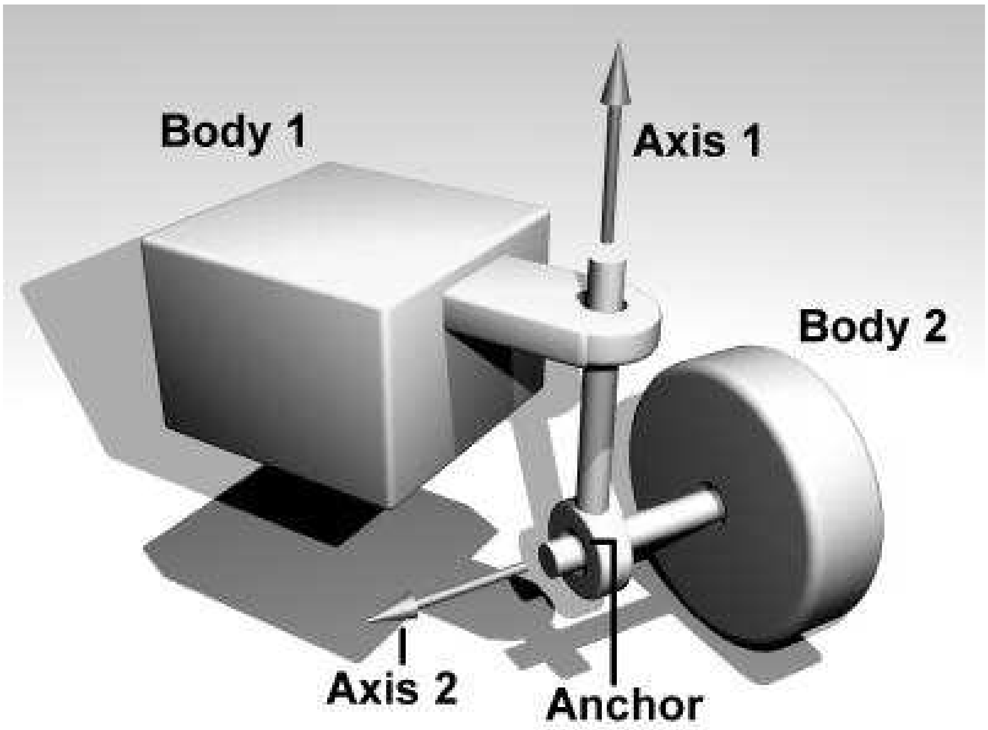
For more informations, see ODE documentation, chapters 3.7 and 3.8.

39.103 void raydium_ode_joint_suspension_name (char *j, dReal erp, dReal cfm):

Same as above, but using element's name.

39.104 int raydium_ode_joint_attach_hinge2 (char *name, int elem1, int elem2, dReal axe1x, dReal axe1y, dReal axe1z, dReal axe2x, dReal axe2y, dReal axe2z):

Will create a new joint between two elements (`elem1` and `elem2`).
Hinge2? is a very specialized joint, perfect for car wheel's for example.



"Axis 1 is specified relative to body 1 (this would be the steering axis if body 1 is the chassis). Axis 2 is specified relative to body 2 (this would be the wheel axis if body 2 is the wheel)."

You must also provide joint's `name`.

Raydium provides `RAYDIUM_ODE_JOINT_SUSP_DEFAULT_AXES` define, useful for most chassis-wheel joints, and `RAYDIUM_ODE_JOINT_AXE_X`, Y and Z for other cases.

You cannot attach a static element.

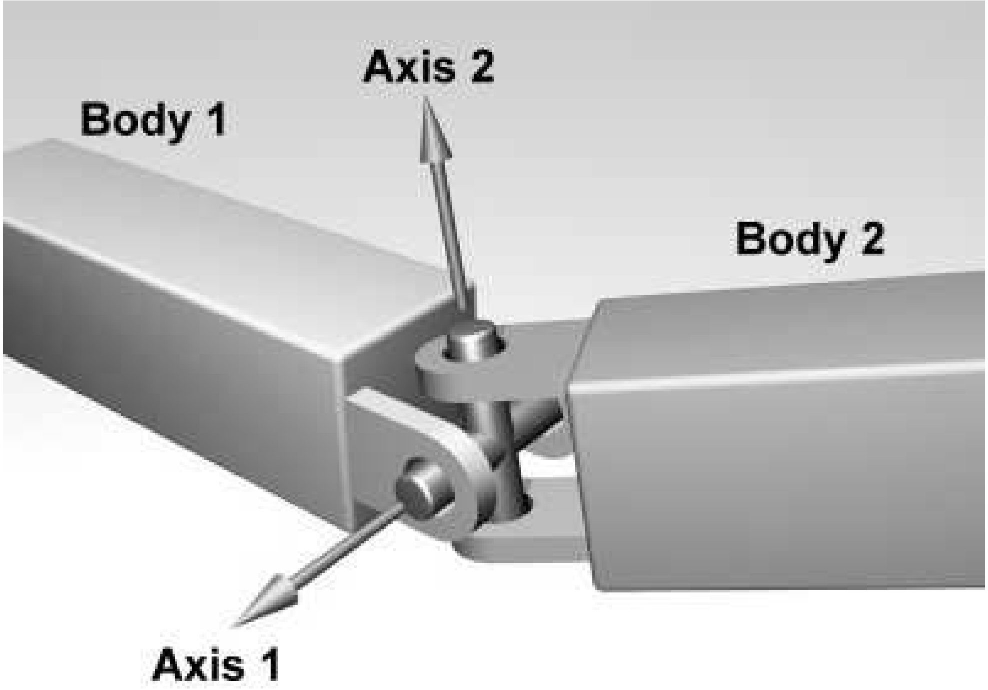
39.105 int raydium_ode_joint_attach_hinge2_name (char *name, char *elem1, char *elem2, dReal axe1x, dReal axe1y, dReal axe1z, dReal axe2x, dReal axe2y, dReal axe2z):

Same as above, but using elements's names.

39.106 int raydium_ode_joint_attach_universal (char *name, int elem1, int

elem2, dReal posx, dReal posy, dReal posz, dReal axe1x, dReal axe1y, dReal axe1z, dReal axe2x, dReal axe2y, dReal axe2z):

Will create a new joint between two elements (`elem1` and `elem2`).



"Given axis 1 on body 1, and axis 2 on body 2 that is perpendicular to axis 1, it keeps them perpendicular. In other words, rotation of the two bodies about the direction perpendicular to the two axes will be equal."

"Axis 1 and axis 2 should be perpendicular to each other."

You must also provide joint's `name`, and joint position (`posx`, `posy`, `posz`) in world coordinates.

Raydium provides `RAYDIUM_ODE_JOINT_AXE_X`, `RAYDIUM_ODE_JOINT_AXE_Y` and `RAYDIUM_ODE_JOINT_AXE_Z` defines, that may help.

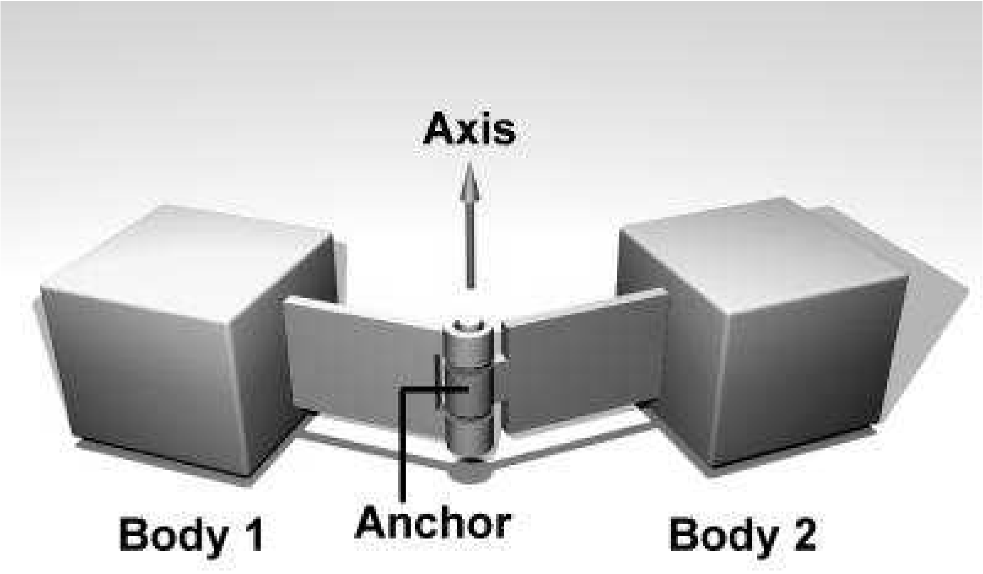
You cannot attach a static element.

39.107 int raydium_ode_joint_attach_universal_name (char *name, char *elem1, char *elem2, dReal posx, dReal posy, dReal posz, dReal axe1x, dReal axe1y, dReal axe1z, dReal axe2x, dReal axe2y, dReal axe2z):

Same as above, but using elements's names.

39.108 int raydium_ode_joint_attach_hinge (char *name, int elem1, int elem2, dReal posx, dReal posy, dReal posz, dReal axe1x, dReal axe1y, dReal axe1z):

Will create a new joint between two elements (`elem1` and `elem2`).



You must provide joint's `name`, and joint position (`posx`, `posy`, `posz`) in world coordinates.

Raydium provides `RAYDIUM_ODE_JOINT_AXE_X`, `RAYDIUM_ODE_JOINT_AXE_Y` and `RAYDIUM_ODE_JOINT_AXE_Z` defines, that may help for setting axis.

You cannot attach a static element.

39.109 int raydium_ode_joint_attach_hinge_name (char *name, char *elem1, char *elem2, dReal posx, dReal posy, dReal posz, dReal axe1x, dReal axe1y, dReal axe1z):

Same as above, but using elements's names.

39.110 int raydium_ode_joint_attach_fixed (char *name, int elem1, int elem2):

Will create a new joint between two elements (`elem1` and `elem2`).

Fixed joint is more a hack than a real joint. Use only when it's absolutely necessary, and have a look to `raydium_ode_element_fix`.

You must provide joint's `name`
You cannot attach a static element.

39.111 int raydium_ode_joint_attach_fixed_name (char *name, char *elem1, char *elem2):

Same as above, but using elements's names.

39.112 void raydium_ode_joint_hinge_limits (int j, dReal lo, dReal hi):

Sets low (`lo`) and high (`hi`) limits for hinge joint `j`.

39.113 void raydium_ode_joint_hinge_limits_name (char *j, dReal lo, dReal hi):

Same as above, but using joint's name.

39.114 void raydium_ode_joint_universal_limits (int j, dReal lo1, dReal hi1, dReal lo2, dReal hi2):

Sets low and hight limits for axe 1 (`lo1`, `hi1`) and axe 2 (`lo2`, `hi2`) for universal joint `j`. See `raydium_ode_joint_attach_universal` for more informations about universal joint axes.

39.115 void raydium_ode_joint_universal_limits_name (char *j, dReal lo1, dReal hi1, dReal lo2, dReal hi2):

Same as above, but using joint's name.

39.116 void raydium_ode_joint_hinge2_block (int j, signed char block):

Sometime, you may need to block rotation for first axe of hinge2 joints, for example with rear wheels of a car. If so, set `block` to `1` (true). Setting `block` back to `0` (false) will restore standard rotation behaviour.

39.117 void raydium_ode_joint_hinge2_block_name (char *name, signed char block):

Same as above, but using joint's name.

39.118 void raydium_ode_joint_delete_callback (int j, void (*f) (int)):

Since joints may break (see `raydium_ode_joint_break_force`), it may be useful to get a callback on joint deletion.
This callback must this prototype:
`void joint_delete(int jid)`
`jid` is the deleted joint id. You can't cancel joint deletion (yet).

39.119 void raydium_ode_joint_delete_callback_name (char *name, void (*f) (int)):

Same as above, but using joint's name.

39.120 void raydium_ode_joint_break_force (int j, dReal maxforce):

Setting a non-zero `maxforce` on a joint will transform this joint into a "breakable joint". There's no unit for `maxforce`, you'll probably have to find the suitable value empirically.

39.121 void raydium_ode_joint_break_force_name (char *name, dReal maxforce):

Same as above, but using joint's name.

39.122 void raydium_ode_joint_elements_get (int j, int *e1, int *e2):

Will return elements (`e1` and `e2`) linked to joint `j`.

39.123 void raydium_ode_joint_elements_get_name (char *j, int *e1, int *e2):

Same as above, but using joint's name.

39.124 void raydium_ode_motor_update_joints_data_internal (int j):

Internal function.

39.125 void raydium_ode_motor_speed (int j, dReal force):

Sets motor `j` speed parameter. This is only suitable for "engine"
and "rocket" type motors. There's no special unit for `force`.

39.126 void raydium_ode_motor_speed_name (char *name, dReal force):

Same as above, but using motor's name.

39.127 void raydium_ode_motor_power_max (int j, dReal power):

Sets motor `j` max power parameter. This is only suitable for "engine"
and "angular" motors. There's no special unit for `power`.

39.128 void raydium_ode_motor_power_max_name (char *name, dReal power):

Same as above, but using motor's name.

39.129 void raydium_ode_motor_angle (int j, dReal angle):

Sets motor `j` angle parameter. This is only suitable for "angular" motors.
`angle` has the units of radians.

39.130 void raydium_ode_motor_angle_name (char *motor, dReal angle):

Same as above, but using motor's name.

39.131 void raydium_ode_motor_gears_set (int m, dReal * gears, int n_gears):

Sets a gearbox for motor `m` (only suitable for "engine" motors).
`gears` is an array of dReal values, with all gears factors).
`n_gears` is the array length (total number of gears for this gearbox).
example:

```
// rear,1,2,3,4,5
dReal gears[]={-0.4,0.4,0.6,0.8,0.9,1.0};
...
raydium_ode_motor_gears_set(main_engine,gears,6);
```

If you want to cancel a gearbox, set a gearbox with only one gear with 1.0
factor value.

Raydium gearboxes implementation is very naive, with 100% output.
For example, a 0.5 gear factor will divide maximum speed by two, but will
provide twice the normal torque.

39.132 void raydium_ode_motor_gears_set_name (char *m, dReal * gears, int n_gears):

Same as above, but using motor's name.

39.133 void raydium_ode_motor_gear_change (int m, int gear):

Switch motor `m` to `gear`.

39.134 void raydium_ode_motor_gear_change_name (char *m, int gear):

Same as above, but using motor's name.

39.135 dReal *raydium_ode_element_pos_get (int j):

This function will return element `j`'s current position, as an array of
3 dReal values.
example:

```
dReal *pos;
dReal pos_copy;
...
pos=raydium_ode_element_pos_get(my_element);
raydium_log("%f %f %f",pos[0],pos[1],pos[2]);
memcpy(pos_copy,pos,sizeof(dReal)*3);
...
```

Returned data is available only for the current frame.

39.136 dReal *raydium_ode_element_pos_get_name (char *name):

Same as above, but using element's name.

39.137 signed char raydium_ode_element_rotq_get (int j, dReal * res):

This function will return element `j`'s current rotation, as an array of
4 dReal values (quaternion), thru `res`.
No memory allocation will be done.

39.138 signed char raydium_ode_element_rotq_get_name (char *name, dReal * res):

Same as above, but using element's name.

39.139 signed char raydium_ode_element_rot_get (int e, dReal * rx, dReal * ry, dReal * rz):

This code is experimental. It should returns element e's current rotation
using 3 dReal angles, in radians. Do not apply back values to the
element since there're not "ODE formated".

39.140 signed char raydium_ode_element_rot_get_name (char *e, dReal * rx, dReal * ry, dReal * rz):

Same as above, but using element's name.

39.141 void raydium_ode_element_sound_update (int e, int source):

This function is a small bridge between RayODE and sound API, updating sound
`source` using element e's position.

39.142 void raydium_ode_element_sound_update_name (char *e, int source):

Same as above, but using element’s name.

39.143 void raydium_ode_element_RelPointPos (int e, dReal px, dReal py, dReal pz, dReal * res):

Give a point (`px`, `py` and `pz`) on element `e` to this function,
and il will return this point in global coordinates (`res`).
Returned data is available only for the current frame.

39.144 void raydium_ode_element_RelPointPos_name (char *e, dReal px, dReal py, dReal pz, dReal * res):

Same as above, but using element’s name.

39.145 int raydium_ode_motor_create (char *name, int obj, signed char type):

This function will create a new motor, using `name` (single), for
object `obj`, with `type`. As said before, available types are
`RAYDIUM_ODE_MOTOR_ENGINE`, `RAYDIUM_ODE_MOTOR_ANGULAR` and
`RAYDIUM_ODE_MOTOR_ROCKET`. See the first part of this chapter for more
informations about motor types.

39.146 void raydium_ode_motor_attach (int motor, int joint, int joint_axe):

This function will link `motor` to `joint`, on axe `joint_axe` (first axe
is axe `0` and so on ...). This is only suitable for engine and angular motors.

39.147 void raydium_ode_motor_attach_name (char *motor, char *joint, int joint_axe):

Same as above, but using motor’s name and joint’s name.

39.148 dReal raydium_ode_motor_speed_get (int m, int gears):

Will return current motor speed.
For engine style motors, if `gears` is sets to 1 (true), returned speed
will be relative to current motor’s gear. Useless for other types.

39.149 dReal raydium_ode_motor_speed_get_name (char *name, int gears):

same as above, but using motor’s name.

39.150 void raydium_ode_motor_rocket_set (int m, int element, dReal x, dReal y, dReal z):

This function will configure rocket motor `m` on `element` at position
(x,y,z). Rocket motors are unusable until this function is called.

39.151 void raydium_ode_motor_rocket_set_name (char *motor, char *element, dReal x, dReal y, dReal z):

same as above, but using motor’s name.

39.152 void raydium_ode_motor_rocket_orientation (int m, dReal rx, dReal ry, dReal rz):

This function will rotate rocket `m` using `rx`,`ry` and `rz` angles in degrees. Base orientation is `z` up.

39.153 void raydium_ode_motor_rocket_orientation_name (char *name, dReal rx, dReal ry, dReal rz):

same as above, but using motor's name.

39.154 void raydium_ode_motor_rocket_playermovement (int m, signed char isplayermovement):

Will configure rocket `m` for player movements. This type of rocket will be automatically disabled when linked element is not touched by anything (ground in most cases).

39.155 void raydium_ode_motor_rocket_playermovement_name (char *m, signed char isplayermovement):

same as above, but using motor's name.

39.156 signed char raydium_ode_motor_delete (int e):

Will obviously delete motor e.

39.157 signed char raydium_ode_motor_delete_name (char *name):

same as above, but using motor's name.

39.158 signed char raydium_ode_joint_delete (int joint):

Will obviously delete `joint`.

39.159 signed char raydium_ode_joint_delete_name (char *name):

same as above, but using joint's name.

39.160 signed char raydium_ode_element_delete (int e, signed char deletejoints):

Will obviously delete element e. Deletion may me queued for some reason, for a very short time (current collide loop). For now, you must set `deletejoints` to 1 (true), since joints without 2 linked elements are invalid.
Linked rocket engines will be deleted, too.

39.161 signed char raydium_ode_element_delete_name (char *name, signed char deletejoints):

Same as above, but using element's name.

39.162 signed char raydium_ode_object_delete (int obj):

Will obviously delete object `obj`. All elements, joints and motors will be deleted with object.

39.163 signed char raydium_ode_object_delete_name (char *name):

Same as above, but using object's name.

39.164 signed char raydium_ode_explosion_delete (int e):

Will delete `RAYDIUM_ODE_NETWORK_EXPLOSION_EXPL` type explosion e.

39.165 signed char raydium_ode_element_moveto (int element, int object, signed char deletejoints):

This function will move `element` from his owner object to another `object`.
This "migration" will not be completed until `element` is not touching
anymore his previous owner.
For now, you must set `deletejoints` to `1` (true), deleting linked joints.

39.166 signed char raydium_ode_element_moveto_name (char *element, char *object, signed char deletejoints):

Same as above, but using element's name and object's name.

39.167 void raydium_ode_joint_break (int j):

Internal joint testing function.

39.168 signed char raydium_ode_launcher (int element, int from_element, dReal * rot, dReal force):

This function will launch an `element` from `from_element`.
You must provide `rot`, an array of 3 dReal angles in degreees, relative
to `from_element` current orientation.
You must also provide a `force`, with no particular unit.

39.169 signed char raydium_ode_launcher_name (char *element, char *from_element, dReal * rot, dReal force):

Same as above, using `element` and `from_element` names.

39.170 signed char raydium_ode_launcher_name_3f (char *element, char *from_element, dReal rx, dReal ry, dReal rz, dReal force):

Same as above, but using 3 dReal values for rotation.

39.171 signed char raydium_ode_launcher_simple (int element, int from_element, dReal * lrot, dReal force):

This function will act the same as previous ones, adding a few things:
- `element` will be aligned with `from_element` (position and rotation).
- `element` will be "migrated" to GLOBAL object during launch.

39.172 signed char raydium_ode_launcher_simple_name (char *element, char *from_element, dReal * rot, dReal force):

Same as above, using `element` and `from_element` names.

39.173 signed char raydium_ode_launcher_simple_name_3f (char *element, char *from_element, dReal rx, dReal ry, dReal rz, dReal force):

Same as above, but using 3 dReal values for rotation.

39.174 void raydium_ode_explosion_blow (dReal radius, dReal max_force, dReal * pos):

This function will create an instantaneous explosion, generating a degressive blowing effect.
You must provide a `radius` (normal world units), a maximum force (`max_force`), and a position (`pos`, 3 x dReal array).

39.175 void raydium_ode_explosion_blow_3f (dReal radius, dReal max_force, dReal px, dReal py, dReal pz):

Same as above, but using 3 dReal values for position.

39.176 int raydium_ode_explosion_create (char *name, dReal final_radius, dReal propag, dReal * pos):

This function will create an spherical growing explosion. Any element in the explosion will be ejected.
As said before: "Use this for very consequent explosions only !".
You must provide `final_radius`, `propag` (growing size) and a position (`pos`, 3 x dReal array).
When an explosion reach its final radius, it will be deleted.

39.177 void raydium_ode_element_camera_inboard (int e, dReal px, dReal py, dReal pz, dReal lookx, dReal looky, dReal lookz):

RayODE to camera API bridge.
Sets the camera on element e at relative position (`px`,`py`,`pz`), and looking at (`lookx`,`looky`,`lookz`) relative point.

39.178 void raydium_ode_element_camera_inboard_name (char *name, dReal px, dReal py, dReal pz, dReal lookx, dReal looky, dReal lookz):

Same as above, but using element’s name.

39.179 void raydium_ode_draw_all (signed char names):

This function will draw all RayODE scene. You must call this function by yourself.
Sets `names` to false (0) for normal rendering.
Other `names` values will:
- draw elements, joints and motors names and elements bounding boxes with `1`.
- draw objets AABB (Axis-Aligned Bounding Box).

39.180 void raydium_ode_near_callback (void *data, dGeomID o1, dGeomID o2):

Internal callback.

39.181 void raydium_ode_callback (void):

Internal frame callback.

39.182 void raydium_ode_time_change (GLfloat perc):

This function will change RayODE timecall frequency, allowing slow motion effects, for example. This function will automatically adjust particle engine time base.
`perc` is the percentage of the normal time base.
Since this function obviously do not change physics accuracy, be careful with `perc` > 100, wich will generate a big load for the CPU.

39.183 void raydium_ode_element_particle (int elem, char *filename):

This function will "fix" a particle generator on element `elem`. You must provide particle generator's `filename`.

39.184 void raydium_ode_element_particle_name (char *elem, char *filename):

Same as above, using element's name.

39.185 void raydium_ode_element_particle_offset (int elem, char *filename, dReal * offset):

Same as `raydium_ode_element_particle`, but with an `offset`, relative to element. `offset` is an array of 3 dReal values.

39.186 void raydium_ode_element_particle_offset_name (char *elem, char *filename, dReal * offset):

Same as above, using element's name.

39.187 void raydium_ode_element_particle_offset_name_3f (char *elem, char *filename, dReal ox, dReal oy, dReal oz):

Same as above, but using 3 dReal values for offset.

39.188 void raydium_ode_element_particle_point (int elem, char *filename):

Same as `raydium_ode_element_particle`, but generator will not be linked with element, only positioned at current element's position.

39.189 void raydium_ode_element_particle_point_name (char *elem, char *filename):

Same as above, using element's name.

39.190 void raydium_camera_smooth_path_to_element (char *path, int element, GLfloat path_step, GLfloat smooth_step):

This function is a clone of `raydium_camera_smooth_path_to_pos` dedicated to RayODE, looking at `element` from path.
You may look at suitable chapter for more informations about `path`, `path_step` and `smooth_step`.

39.191 void raydium_camera_smooth_path_to_element_name (char *path, char *element, GLfloat path_step, GLfloat smooth_step):

Same as above, using element's name.

39.192 void raydium_camera_smooth_element_to_path_name (char *element,

char *path, GLfloat path_step, GLfloat smooth_step):

This function is a clone of `raydium_camera_smooth_pos_to_path` dedicated to RayODE, looking at path, from `element`.
Here, you must provide element's name.
You may look at suitable chapter for more informations about `path`, `path_step` and `smooth_step`.

39.193 void raydium_camera_smooth_element_to_path_offset (int element, GLfloat offset_x, GLfloat offset_y, GLfloat offset_z, char *path, GLfloat path_step, GLfloat smooth_step):

This function is a clone of `raydium_camera_smooth_pos_to_path` dedicated to RayODE and providing an offset (for `element`), looking at path, from `element`.
You may look at suitable chapter for more informations about `path`, `path_step` and `smooth_step`.

39.194 void raydium_camera_smooth_element_to_path_offset_name (char *element, GLfloat offset_x, GLfloat offset_y, GLfloat offset_z, char *path, GLfloat path_step, GLfloat smooth_step):

Same as above, using element's name.

39.195 int raydium_ode_capture_3d(char *filename):

This function is provided "for fun" only. The main idea is to dump all scene to a .tri file (`filename`). A .sprt file will also be created, wich is a special file format with all particles found during the dump. You can reload .sprt files with `raydium_particle_state_restore`.
Note from source code:

```
// This function is provided "for fun" only. Not all effects are dumped:
// Missing : shadows, forced colors, before/after callbacks,
// fixed elements, ...
// Some code is pasted from file.c (and this is BAD ! :)
```

39.196 int raydium_ode_orphans_check(void):

Search orphans in all objects. An orphan is a geometry that exists into ODE but is not managed by RayODE.
This function will print object with orphans and return total orphan count.

40 RayODE network layer:

40.1 Introduction:

Physics engines are extremely powerful tools, but it turns to nightmares when the application must be networked. RayODE API provides its own network layer, using Raydium lower level network API. And the great thing is that you've almost anything to do !

Just choose the best "send" function and let Raydium do the rest.

RayODE Net will use udp streams, netcall (RPC), smart timeouts, predictions, dead reckoning, and many others voodoo things. Just trust.

A few things about internals:
- NID: Network ID. Every networked element have a NID.
- Distant elements are localy created using static elements, owned by an object called "`DISTANT`".
- `raydium_ode_network_maxfreq` defines the paquet sending frequency. By default, this value is `RAYDIUM_ODE_NETWORK_MAXFREQ`, but you can use `--ode-rate` command line switch.
- No rotation prediction is done.
- See `config.h` if you want to disable prediction (`ODE_PREDICTION`) or to debug RayODE Net (`DEBUG_ODENET`, very verbose !).
- Explosions are also automatically managed by RayODE Net.
- Do NOT use Raydium lower level network API when using RayODE Net. Use netcalls, propags and so on.

Nothing is said here about how to create a RayODE Net server. There’s only a few more things to do if you already have a standard server, but since it’s unsupported for now, you must have a look to existing RayODE Net servers.

40.2 int raydium_ode_network_MaxElementsPerPacket (void):

This function will return how many elements may be sent with current packet size (see `common.h`).

40.3 int raydium_network_nid_element_find (int nid):

Internal. Find wich element have `nid`.

40.4 void raydium_ode_network_newdel_event (int type, char *buff):

Internal. NEWDEL netcall event.
NEWDEL is fired when a new element is created or deleted somewhere in the network.

40.5 void raydium_ode_network_nidwho_event (int type, char *buff):

Internal. NIDWHO netcall event.
NIDWHO is sent when someone received some "update" informations about a nid, but didn’t received previous NEWDEL informations for this nid.
The nid owner will send a reply.

Most reasons for this are:
- We are a new client and we dont known anything about the whole scene.
- The NEWDEL packet was lost ("TCP style" packets may be lost too ...)

NIDWHO answer will be used by every peer to refresh its own copy of the element informations (geometry type, mesh, size and tag).

40.6 void raydium_ode_network_explosion_event (int type, char *buff):

Internal explosion netcall event.(`RAYDIUM_ODE_NETWORK_EXPLOSION_EXPL` and `RAYDIUM_ODE_NETWORK_EXPLOSION_BLOW`).

40.7 void raydium_ode_network_init (void):

Internal. Will initialize all RayODE Net layer and register netcalls.

40.8 signed char raydium_ode_network_TimeToSend (void):

Almost internal. Will return 1 (true) if it's time to send a new packet, using `raydium_ode_network_maxfreq` value.

40.9 void raydium_ode_network_element_send (short nelems, int *e):

Will send all elements of e array to network. You must provide array lenght using `nelems`.

To "time to send ?" test is done, you'll probably have to do it yourself.

40.10 void raydium_ode_network_element_send_all (void):

Will try to send all elements to network. Warning, packet size may be to small to send all elements !..

40.11 void raydium_ode_network_element_send_random (int nelems):

Will send randomly chosen elements to network. You must provide how many elements you want with `nelems`, but RAYDIUM_ODE_NETWORK_OPTIMAL is available.

40.12 void raydium_ode_network_element_send_iterative (int nelems):

Will send elements to network, iteratively chose. You must provide how many elements you want with `nelems`, but RAYDIUM_ODE_NETWORK_OPTIMAL is available.

40.13 void raydium_ode_network_nidwho (int nid):

Internal. Will ask for informations for `nid` (see above).

NID sending frequency is now limited, since a lot of overhead was generated when new clients were joining a "big" network.

40.14 void raydium_ode_network_apply (raydium_ode_network_Event * ev):

Internal. This callback is fired when new data is received. A lot of things are done here (timeouts, dead reckoning, ...)

40.15 void raydium_ode_network_read (void):

Internal. Reads new packets, if any.

40.16 void raydium_ode_network_element_new (int e):

Internal. Send a new element to network.

40.17 void raydium_ode_network_element_delete (int e):

Internal. Send "delete event" to network, since we're deleting one of "our" elements.

40.18 void raydium_ode_network_explosion_send (raydium_ode_network_Explosion * exp):

Internal. Send a new explosion event.

40.19 signed char raydium_ode_network_element_isdistant (int elem):

Will return true (1) if element `elem` is "distant", or false (0) if it's one of "our" elements.

40.20 signed char raydium_ode_network_element_isdistant_name (char *elem):

Same as above, but using element's name.

40.21 signed char raydium_ode_network_element_distantowner(int elem):

Returns UID (peer "user" ID) for the distant element owner. See `network.c` documentation for more informations about UID.

40.22 signed char raydium_ode_network_element_distantowner_name(char *elem):

Same as above, but using element's name.

40.23 void raydium_ode_network_element_trajectory_correct (int elem):

Internal. Applies dead reckoning values to element.

40.24 void raydium_ode_network_elment_next_local(void):

Call this function when you don't want that the next created element is sent to network ("local only" element).

41 RegAPI:

41.1 Introduction:

RegAPI is an internal system that exports some Raydium's API functions to scripting engine, creating bindings.
See RayPHP chapter for more informations anout scripting.

41.2 void raydium_register_api(void):

Internal. Will register Raydium API.

42 Video playback:

42.1 Introduction:

Raydium supports simple video playback, thru a special video codec (JPGS), useful for menus enhancements, "speaking" thumbnails, ...
This codec only supports video, use sound API if needed.
You will find an small utility, `mk_jpgs` in Raydium source tree, didacted to movie creation.

42.2 How to create a movie ?:

First, compile `mk_jpgs`: example: `gcc mk_jpgs -o mk_jpgs` or any other standard build command.
Then, generate JPEG pictures:
`mplayer movie.avi -vo jpeg:quality=50 -vf scale=256:256`, where you may change quality factor and output size. Use "hardware firendly" sizes (64, 128,256,...) !
You can now build JPGS file:
`./mk_jpgs 25 256 256 video.jpgs` (fps, size x, size y, output file)

42.3 void raydium_video_init(void):

Internal use.

42.4 signed char raydium_video_isvalid(int i):

Internal use, but you can call this function if you want to verify if a video id is valid (in bounds and open).

42.5 int raydium_video_find_free(void):

Internal use.
Finds a free video slot.

42.6 int raydium_video_find(char *name):

Resolvs video `name`, returning suitable texture id (native Raydium texture id).

42.7 void raydium_video_jpeg_decompress(FILE *fp,unsigned char *to):

Internal.

42.8 int raydium_video_open(char *filename, char *as):

This function will open and prepare video `filename`, and will attach this video to a "live texture" (see Live API chapter, if needed).

42.9 void raydium_video_callback_video(int id):

Internal use.

42.10 void raydium_video_callback(void):

Internal use. Frame callback.

42.11 void raydium_video_delete(int id):

Will delete video `id`. Warning: this function will not delete associated Live texture, so you may open a new video with the same texture name, but video size must be the same a the previous one.

42.12 void raydium_video_delete_name(char *name):

Same as above, using video name.

43 PHP scripting engine:

43.1 Introduction:

This is the internal part of the RayPHP API, where Raydium
deals with Zend engine.

All this is for internal use, so no documentation is provided.

44 Miscalleneous:

44.1 License:

Raydium engine and provided applications are released under GPL version 2.
You can found the original text of this license here :
http://www.gnu.org/licenses/gpl.txt

44.2 About CQFD Corp Raydium Team:

Alphabetical order:
batcox, Blue Prawn, Cocorobix, FlexH, Jimbo, manproc, Mildred, neub, RyLe?,
whisky, willou, Xfennec, Yoltie

44.3 Todo:

No particular order:
- rendering core rewrite
- self-running demo
- (idea from RyLe?) ’rayphp/’ scripts integration into the binary (and why
not, a "PACK style" support).
- more network stack optimisations (UDP reads, mainly)
- better organisation of comp.sh and ocomp.sh files (separate options
and build process)

See also my todo: http://raydium.yoopla.org/wiki/XfenneC

Please, if you start working on a feature, say it on the Wiki.

44.4 Links:

http://raydium.cqfd-corp.org (Raydium home)
svn://cqfd-corp.org/raydium/trunk (SVN trunk)
http://raydium.cqfd-corp.org/svn.php (SVN "live" changelog)
http://memak.cqfd-corp.org (MeMak forum: "a game using Raydium", french)
http://www.cqfd-corp.org (CQFD homesite)
mailto:xfennec -AT- cqfd-corp.org

44.5 Greets:

RyLe?: original implementation of sound.c (OpenAL core sound API)

BatcoX: export of RayODE functions into RayPHP (reg_api.c)
and additional PHP wrappers (wrappers.c)

Mildred: header and Makefile generator, dynamic version of

Raydium (.so and .a) for Linux.

Chapters:

Index:

char *path, GLfloat path_step, GLfloat smooth_step)
raydium_camera_vectors (GLfloat * res3)
raydium_capture_frame(char *filename)
raydium_capture_frame_auto(void)
raydium_capture_frame_jpeg(char *filename)
raydium_clear_color_update (void)
raydium_clear_frame (void)
raydium_console_complete (char *str)
raydium_console_draw (void)
raydium_console_event (void)
raydium_console_exec_last_command (void)
raydium_console_exec_script (char *file)
raydium_console_gets (char *where)
raydium_console_history_add (char *str)
raydium_console_history_next (void)
raydium_console_history_previous (void)
raydium_console_history_save (void)
raydium_console_init (void)
raydium_console_internal_isalphanumuscore (char c)
raydium_console_line_add (char *format, ...)
raydium_file_dirname(char *dest,char *from)
raydium_file_fopen(char *file, char *mode)
raydium_file_log_fopen_display(void)
raydium_file_set_textures (char *name)
raydium_fog_color_update (void)
raydium_fog_disable (void)
raydium_fog_enable (void)
raydium_fog_mode (void)
raydium_gui_button_clicked(void)
raydium_gui_button_create(char *name, int window, GLfloat px, GLfloat py, char *caption, void *OnClick)
raydium_gui_button_create_simple(char *name, int window, GLfloat px, GLfloat py, char *caption)
raydium_gui_button_draw(int w, int window)
raydium_gui_button_read(int window, int widget, char *str)
raydium_gui_check_create(char *name, int window, GLfloat px, GLfloat py, char *caption, signed char checked)
raydium_gui_check_draw(int w, int window)
raydium_gui_check_read(int window, int widget, char *str)
raydium_gui_combo_create(char *name, int window, GLfloat px, GLfloat py, char *items, int current)
raydium_gui_combo_draw(int w, int window)
raydium_gui_combo_read(int window, int widget, char *str)
raydium_gui_draw(void)
raydium_gui_edit_create(char *name, int window, GLfloat px, GLfloat py, char *default_text)
raydium_gui_edit_draw(int w, int window)
raydium_gui_edit_read(int window, int widget, char *str)
raydium_gui_hide(void)
raydium_gui_init(void)

raydium_gui_internal_object_create(char *name, int window, signed char type, GLfloat px, GLfloat py, GLfloat sizex, GLfloat sizey, GLfloat font_size)
raydium_gui_isvisible(void)
raydium_gui_label_create(char *name, int window, GLfloat px, GLfloat py, char *caption, GLfloat r, GLfloat g, GLfloat b)
raydium_gui_label_draw(int w, int window)
raydium_gui_label_read(int window, int widget, char *str)
raydium_gui_read(int window, int widget, char *str)
raydium_gui_read_name(char *window, char *widget, char *str)
raydium_gui_show(void)
raydium_gui_theme_init(void)
raydium_gui_theme_load(char *filename)
raydium_gui_track_create(char *name, int window, GLfloat px, GLfloat py, int min, int max, int current)
raydium_gui_track_draw(int w, int window)
raydium_gui_track_read(int window, int widget, char *str)
raydium_gui_widget_draw_internal(GLfloat *uv, GLfloat *xy)
raydium_gui_widget_find(char *name, int window)
raydium_gui_widget_isvalid(int i, int window)
raydium_gui_widget_next(void)
raydium_gui_widget_sizes(GLfloat sizex, GLfloat sizey, GLfloat font_size)
raydium_gui_window_create(char *name, GLfloat px, GLfloat py, GLfloat sizex, GLfloat sizey)
raydium_gui_window_delete(int window)
raydium_gui_window_delete_name(char *window)
raydium_gui_window_draw(int window)
raydium_gui_window_find(char *name)
raydium_gui_window_init(int window)
raydium_gui_window_isvalid(int i)
raydium_init_args (int argc, char * *argv)
raydium_init_cli_option (char *option, char *value)
raydium_init_cli_option_default (char *option, char *value, char *default_value)
raydium_init_engine (void)
raydium_init_key (void)
raydium_init_lights (void)
raydium_init_objects (void)
raydium_init_reset (void)
raydium_internal_dump (void)
raydium_internal_dump_matrix (int n)
raydium_internal_live_close(void)
raydium_internal_live_video_callback(void)
raydium_joy_ff_autocenter (int perc)
raydium_joy_ff_tremble_set (GLfloat period, GLfloat force)
raydium_joy_key_emul (void)
raydium_key_normal_callback (GLuint key, int x, int y)
raydium_key_pressed (GLuint key)
raydium_key_special_callback (GLuint key, int x, int y)
raydium_key_special_up_callback (GLuint key, int x, int y)

raydium_light_blink_internal_update (GLuint l)
raydium_light_blink_start (GLuint l, int fpc)
raydium_light_callback (void)
raydium_light_disable (void)
raydium_light_enable (void)
raydium_light_move (GLuint l, GLfloat * vect)
raydium_light_off (GLuint l)
raydium_light_on (GLuint l)
raydium_light_reset (GLuint l)
raydium_light_switch (GLuint l)
raydium_light_to_GL_light (GLuint l)
raydium_light_update_all (GLuint l)
raydium_light_update_intensity (GLuint l)
raydium_light_update_position (GLuint l)
raydium_light_update_position_all (void)
raydium_live_init(void)
raydium_live_texture_create(char *as, unsigned char *data_source, int tx, int ty, int bpp)
raydium_live_texture_find(int original_texture)
raydium_live_texture_find_free(void)
raydium_live_texture_isvalid(int i)
raydium_live_texture_mask(int livetex, GLfloat alpha)
raydium_live_texture_mask_name(char *texture, GLfloat alpha)
raydium_live_texture_refresh(int livetex)
raydium_live_texture_refresh_callback_set(int livetex, void *callback)
raydium_live_texture_refresh_callback_set_name(char *texture, void *callback)
raydium_live_texture_refresh_name(char *texture)
raydium_live_texture_video(int device_id, char *as)
raydium_live_video_find_free(void)
raydium_live_video_isvalid(int i)
raydium_live_video_open(char *device, int sizex, int sizey)
raydium_live_video_open_auto(void)
raydium_live_video_read(raydium_live_Device *dev)
raydium_log (char *format, ...)
raydium_mouse_button_pressed (int button)
raydium_mouse_click_callback (int but, int state, int x, int y)
raydium_mouse_hide() (macro)
raydium_mouse_init (void)
raydium_mouse_isvisible(void)
raydium_mouse_move(x,y) (macro)
raydium_mouse_move_callback (int x, int y)
raydium_mouse_show() (macro)
raydium_network_broadcast (signed char type, char *buff)
raydium_network_client_connect_to (char *server)
raydium_network_close (void)
raydium_network_init (void)
raydium_network_internal_dump (void)
raydium_network_internal_find_delay_addr (int player)
raydium_network_internal_server_delays_dump (void)

raydium_network_internet_test(void)
raydium_network_netcall_add (void *ptr, int type, signed char tcp)
raydium_network_netcall_exec (int type, char *buff)
raydium_network_nid_element_find (int nid)
raydium_network_player_name (char *str)
raydium_network_propag_add (int type, void *data, int size)
raydium_network_propag_find (int type)
raydium_network_propag_recv (int type, char *buff)
raydium_network_propag_refresh (int type)
raydium_network_propag_refresh_all (void)
raydium_network_propag_refresh_id (int i)
raydium_network_queue_ack_recv (int type, char *buff)
raydium_network_queue_ack_send (unsigned short tcpid, struct sockaddr *to)
raydium_network_queue_check_time (void)
raydium_network_queue_element_add (char *packet, struct sockaddr *to)
raydium_network_queue_element_init (raydium_network_Tcp * e)
raydium_network_queue_is_tcpid (int type)
raydium_network_queue_tcpid_gen (void)
raydium_network_queue_tcpid_known (unsigned short tcpid, unsigned short player)
raydium_network_queue_tcpid_known_add (int tcpid, int player)
raydium_network_read (int *id, signed char *type, char *buff)
raydium_network_read_flushed (int *id, signed char *type, char *buff)
raydium_network_server_create (void)
raydium_network_set_socket_block (int block)
raydium_network_timeout_check (void)
raydium_network_write (struct sockaddr *to, int from, signed char type, char *buff)
raydium_normal_generate_lastest_triangle (int default_visu)
raydium_normal_regenerate_all (void)
raydium_normal_restore_all (void)
raydium_normal_smooth_all (void)
raydium_object_deform (GLuint obj, GLfloat ampl)
raydium_object_deform_name (char *name, GLfloat ampl)
raydium_object_draw (GLuint o)
raydium_object_draw_name (char *name)
raydium_object_find (char *name)
raydium_object_find_axes_max (GLuint obj, GLfloat * tx, GLfloat * ty, GLfloat * tz)
raydium_object_find_dist_max (GLuint obj)
raydium_object_find_load (char *name)
raydium_object_load (char *filename)
raydium_object_reset (GLuint o)
raydium_ode_callback (void)
raydium_ode_capture_3d(char *filename)
raydium_ode_draw_all (signed char names)
raydium_ode_element_OnBlow (int e, void *OnBlow)
raydium_ode_element_OnBlow_name (char *e, void *OnBlow)
raydium_ode_element_OnDelete (int e, void *OnDelete)

raydium_ode_element_OnDelete_name (char *e, void *OnDelete)
raydium_ode_element_RelPointPos (int e, dReal px, dReal py, dReal pz, dReal * res)
raydium_ode_element_RelPointPos_name (char *e, dReal px, dReal py, dReal pz, dReal * res)
raydium_ode_element_aabb_get (int element, dReal * aabb)
raydium_ode_element_aabb_get_name (char *element, dReal * aabb)
raydium_ode_element_addforce (int e, dReal * vect)
raydium_ode_element_addforce_name (char *e, dReal * vect)
raydium_ode_element_addforce_name_3f (char *e, dReal vx, dReal vy, dReal vz)
raydium_ode_element_addtorque (int e, dReal * vect)
raydium_ode_element_addtorque_name (char *e, dReal * vect)
raydium_ode_element_addtorque_name_3f (char *e, dReal vx, dReal vy, dReal vz)
raydium_ode_element_camera_inboard (int e, dReal px, dReal py, dReal pz, dReal lookx, dReal looky, dReal lookz)
raydium_ode_element_camera_inboard_name (char *name, dReal px, dReal py, dReal pz, dReal lookx, dReal looky, dReal lookz)
raydium_ode_element_data_get (int e)
raydium_ode_element_data_get_name (char *e)
raydium_ode_element_data_set (int e, void *data)
raydium_ode_element_data_set_name (char *e, void *data)
raydium_ode_element_delete (int e, signed char deletejoints)
raydium_ode_element_delete_name (char *name, signed char deletejoints)
raydium_ode_element_find (char *name)
raydium_ode_element_fix (char *name, int *elem, int nelems, signed char keepgeoms)
raydium_ode_element_gravity (int e, signed char enable)
raydium_ode_element_gravity_name (char *e, signed char enable)
raydium_ode_element_ground_texture_get (int e)
raydium_ode_element_ground_texture_get_name (char *e)
raydium_ode_element_isvalid (int i)
raydium_ode_element_linearvelocity_get (int e)
raydium_ode_element_linearvelocity_get_name (char *e)
raydium_ode_element_material (int e, dReal erp, dReal cfm)
raydium_ode_element_material_name (char *name, dReal erp, dReal cfm)
raydium_ode_element_move (int elem, dReal * pos)
raydium_ode_element_move_3f(int elem, dReal x,dReal y, dReal z)
raydium_ode_element_move_name (char *name, dReal * pos)
raydium_ode_element_move_name_3f (char *name, dReal x, dReal y, dReal z)
raydium_ode_element_moveto (int element, int object, signed char deletejoints)
raydium_ode_element_moveto_name (char *element, char *object, signed char deletejoints)
raydium_ode_element_object_get (int e)
raydium_ode_element_object_get_name (char *e)
raydium_ode_element_particle (int elem, char *filename)
raydium_ode_element_particle_name (char *elem, char *filename)
raydium_ode_element_particle_offset (int elem, char *filename, dReal *

offset)
raydium_ode_element_particle_offset_name (char *elem, char *filename, dReal * offset)
raydium_ode_element_particle_offset_name_3f (char *elem, char *filename, dReal ox, dReal oy, dReal oz)
raydium_ode_element_particle_point (int elem, char *filename)
raydium_ode_element_particle_point_name (char *elem, char *filename)
raydium_ode_element_player_angle (int e, dReal angle)
raydium_ode_element_player_angle_name (char *e, dReal angle)
raydium_ode_element_player_get (int e)
raydium_ode_element_player_get_name (char *name)
raydium_ode_element_player_set (int e, signed char isplayer)
raydium_ode_element_player_set_name (char *name, signed char isplayer)
raydium_ode_element_pos_get (int j)
raydium_ode_element_pos_get_name (char *name)
raydium_ode_element_rot_get (int e, dReal * rx, dReal * ry, dReal * rz)
raydium_ode_element_rot_get_name (char *e, dReal * rx, dReal * ry, dReal * rz)
raydium_ode_element_rotate (int elem, dReal * rot)
raydium_ode_element_rotate_3f (int elem, dReal rx, dReal ry, dReal rz)
raydium_ode_element_rotate_direction (int elem, signed char Force0OrVel1)
raydium_ode_element_rotate_direction_name (char *e, signed char Force0OrVel1)
raydium_ode_element_rotate_name (char *name, dReal * rot)
raydium_ode_element_rotate_name_3f (char *name, dReal rx, dReal ry, dReal rz)
raydium_ode_element_rotateq (int elem, dReal * rot)
raydium_ode_element_rotateq_name (char *name, dReal * rot)
raydium_ode_element_rotfriction (int e, dReal rotfriction)
raydium_ode_element_rotfriction_name (char *e, dReal rotfriction)
raydium_ode_element_rotq_get (int j, dReal * res)
raydium_ode_element_rotq_get_name (char *name, dReal * res)
raydium_ode_element_slip (int e, dReal slip)
raydium_ode_element_slip_name (char *e, dReal slip)
raydium_ode_element_sound_update (int e, int source)
raydium_ode_element_sound_update_name (char *e, int source)
raydium_ode_element_tag_get (int e)
raydium_ode_element_tag_get_name (char *e)
raydium_ode_element_touched_get (int e)
raydium_ode_element_touched_get_name (char *e)
raydium_ode_element_ttl_set (int e, int ttl)
raydium_ode_element_ttl_set_name (char *e, int ttl)
raydium_ode_element_unfix (int e)
raydium_ode_explosion_blow (dReal radius, dReal max_force, dReal * pos)
raydium_ode_explosion_blow_3f (dReal radius, dReal max_force, dReal px, dReal py, dReal pz)
raydium_ode_explosion_create (char *name, dReal final_radius, dReal propag, dReal * pos)
raydium_ode_explosion_delete (int e)
raydium_ode_explosion_find (char *name)

raydium_ode_explosion_isvalid (int i)
raydium_ode_ground_dTriArrayCallback (dGeomID TriMesh, dGeomID RefObject, const int *TriIndices, int TriCount)
raydium_ode_ground_dTriCallback (dGeomID TriMesh, dGeomID RefObject, int TriangleIndex)
raydium_ode_ground_set_name (char *name)
raydium_ode_init (void)
raydium_ode_init_element (int i)
raydium_ode_init_explosion (int e)
raydium_ode_init_joint (int i)
raydium_ode_init_motor (int i)
raydium_ode_init_object (int i)
raydium_ode_joint_attach_fixed (char *name, int elem1, int elem2)
raydium_ode_joint_attach_fixed_name (char *name, char *elem1, char *elem2)
raydium_ode_joint_attach_hinge (char *name, int elem1, int elem2, dReal posx, dReal posy, dReal posz, dReal axe1x, dReal axe1y, dReal axe1z)
raydium_ode_joint_attach_hinge2 (char *name, int elem1, int elem2, dReal axe1x, dReal axe1y, dReal axe1z, dReal axe2x, dReal axe2y, dReal axe2z)
raydium_ode_joint_attach_hinge2_name (char *name, char *elem1, char *elem2, dReal axe1x, dReal axe1y, dReal axe1z, dReal axe2x, dReal axe2y, dReal axe2z)
raydium_ode_joint_attach_hinge_name (char *name, char *elem1, char *elem2, dReal posx, dReal posy, dReal posz, dReal axe1x, dReal axe1y, dReal axe1z)
raydium_ode_joint_attach_universal (char *name, int elem1, int elem2, dReal posx, dReal posy, dReal posz, dReal axe1x, dReal axe1y, dReal axe1z, dReal axe2x, dReal axe2y, dReal axe2z)
raydium_ode_joint_attach_universal_name (char *name, char *elem1, char *elem2, dReal posx, dReal posy, dReal posz, dReal axe1x, dReal axe1y, dReal axe1z, dReal axe2x, dReal axe2y, dReal axe2z)
raydium_ode_joint_break (int j)
raydium_ode_joint_break_force (int j, dReal maxforce)
raydium_ode_joint_break_force_name (char *name, dReal maxforce)
raydium_ode_joint_delete (int joint)
raydium_ode_joint_delete_callback (int j, void (*f) (int))
raydium_ode_joint_delete_callback_name (char *name, void (*f) (int))
raydium_ode_joint_delete_name (char *name)
raydium_ode_joint_elements_get (int j, int *e1, int *e2)
raydium_ode_joint_elements_get_name (char *j, int *e1, int *e2)
raydium_ode_joint_find (char *name)
raydium_ode_joint_hinge2_block (int j, signed char block)
raydium_ode_joint_hinge2_block_name (char *name, signed char block)
raydium_ode_joint_hinge_limits (int j, dReal lo, dReal hi)
raydium_ode_joint_hinge_limits_name (char *j, dReal lo, dReal hi)
raydium_ode_joint_isvalid (int i)
raydium_ode_joint_suspension (int j, dReal erp, dReal cfm)
raydium_ode_joint_suspension_name (char *j, dReal erp, dReal cfm)
raydium_ode_joint_universal_limits (int j, dReal lo1, dReal hi1, dReal lo2, dReal hi2)

raydium_ode_joint_universal_limits_name (char *j, dReal lo1, dReal hi1, dReal lo2, dReal hi2)
raydium_ode_launcher (int element, int from_element, dReal * rot, dReal force)
raydium_ode_launcher_name (char *element, char *from_element, dReal * rot, dReal force)
raydium_ode_launcher_name_3f (char *element, char *from_element, dReal rx, dReal ry, dReal rz, dReal force)
raydium_ode_launcher_simple (int element, int from_element, dReal * lrot, dReal force)
raydium_ode_launcher_simple_name (char *element, char *from_element, dReal * rot, dReal force)
raydium_ode_launcher_simple_name_3f (char *element, char *from_element, dReal rx, dReal ry, dReal rz, dReal force)
raydium_ode_motor_angle (int j, dReal angle)
raydium_ode_motor_angle_name (char *motor, dReal angle)
raydium_ode_motor_attach (int motor, int joint, int joint_axe)
raydium_ode_motor_attach_name (char *motor, char *joint, int joint_axe)
raydium_ode_motor_create (char *name, int obj, signed char type)
raydium_ode_motor_delete (int e)
raydium_ode_motor_delete_name (char *name)
raydium_ode_motor_find (char *name)
raydium_ode_motor_gear_change (int m, int gear)
raydium_ode_motor_gear_change_name (char *m, int gear)
raydium_ode_motor_gears_set (int m, dReal * gears, int n_gears)
raydium_ode_motor_gears_set_name (char *m, dReal * gears, int n_gears)
raydium_ode_motor_isvalid (int i)
raydium_ode_motor_power_max (int j, dReal power)
raydium_ode_motor_power_max_name (char *name, dReal power)
raydium_ode_motor_rocket_orientation (int m, dReal rx, dReal ry, dReal rz)
raydium_ode_motor_rocket_orientation_name (char *name, dReal rx, dReal ry, dReal rz)
raydium_ode_motor_rocket_playermovement (int m, signed char isplayermovement)
raydium_ode_motor_rocket_playermovement_name (char *m, signed char isplayermovement)
raydium_ode_motor_rocket_set (int m, int element, dReal x, dReal y, dReal z)
raydium_ode_motor_rocket_set_name (char *motor, char *element, dReal x, dReal y, dReal z)
raydium_ode_motor_speed (int j, dReal force)
raydium_ode_motor_speed_get (int m, int gears)
raydium_ode_motor_speed_get_name (char *name, int gears)
raydium_ode_motor_speed_name (char *name, dReal force)
raydium_ode_motor_update_joints_data_internal (int j)
raydium_ode_name_auto (char *prefix, char *dest)
raydium_ode_near_callback (void *data, dGeomID o1, dGeomID o2)
raydium_ode_network_MaxElementsPerPacket (void)
raydium_ode_network_TimeToSend (void)

raydium_ode_network_apply (raydium_ode_network_Event * ev)
raydium_ode_network_element_delete (int e)
raydium_ode_network_element_distantowner(int elem)
raydium_ode_network_element_distantowner_name(char *elem)
raydium_ode_network_element_isdistant (int elem)
raydium_ode_network_element_isdistant_name (char *elem)
raydium_ode_network_element_new (int e)
raydium_ode_network_element_send (short nelems, int *e)
raydium_ode_network_element_send_all (void)
raydium_ode_network_element_send_iterative (int nelems)
raydium_ode_network_element_send_random (int nelems)
raydium_ode_network_element_trajectory_correct (int elem)
raydium_ode_network_elment_next_local(void)
raydium_ode_network_explosion_event (int type, char *buff)
raydium_ode_network_explosion_send (raydium_ode_network_Explosion * exp)
raydium_ode_network_init (void)
raydium_ode_network_newdel_event (int type, char *buff)
raydium_ode_network_nidwho (int nid)
raydium_ode_network_nidwho_event (int type, char *buff)
raydium_ode_network_read (void)
raydium_ode_object_addforce (int o, dReal * vect)
raydium_ode_object_addforce_name (char *o, dReal * vect)
raydium_ode_object_addforce_name_3f (char *o, dReal vx, dReal vy, dReal vz)
raydium_ode_object_box_add (char *name, int group, dReal mass, dReal tx, dReal ty, dReal tz, signed char type, int tag, char *mesh)
raydium_ode_object_colliding (int o, signed char colliding)
raydium_ode_object_colliding_name (char *o, signed char colliding)
raydium_ode_object_create (char *name)
raydium_ode_object_delete (int obj)
raydium_ode_object_delete_name (char *name)
raydium_ode_object_find (char *name)
raydium_ode_object_isvalid (int i)
raydium_ode_object_linearvelocity_set (int o, dReal * vect)
raydium_ode_object_linearvelocity_set_name (char *o, dReal * vect)
raydium_ode_object_linearvelocity_set_name_3f (char *o, dReal vx, dReal vy, dReal vz)
raydium_ode_object_move (int obj, dReal * pos)
raydium_ode_object_move_name (char *name, dReal * pos)
raydium_ode_object_move_name_3f (char *name, dReal x, dReal y, dReal z)
raydium_ode_object_rename (int o, char *newname)
raydium_ode_object_rename_name (char *o, char *newname)
raydium_ode_object_rotate(int obj, dReal *rot)
raydium_ode_object_rotate_name(char *obj, dReal *rot)
raydium_ode_object_rotate_name_3f(char *obj, dReal rx, dReal ry, dReal rz)
raydium_ode_object_rotateq (int obj, dReal * rot)
raydium_ode_object_rotateq_name (char *obj, dReal * rot)
raydium_ode_object_sphere_add (char *name, int group, dReal mass, dReal radius, signed char type, int tag, char *mesh)

raydium_ode_orphans_check(void)
raydium_ode_time_change (GLfloat perc)
raydium_osd_alpha_change (GLfloat a)
raydium_osd_color_change (GLfloat r, GLfloat g, GLfloat b)
raydium_osd_color_ega (char hexa)
raydium_osd_color_rgba (GLfloat r, GLfloat g, GLfloat b, GLfloat a)
raydium_osd_cursor_draw (void)
raydium_osd_cursor_set (char *texture, GLfloat xsize, GLfloat ysize)
raydium_osd_draw (int tex, GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2)
raydium_osd_draw_name (char *tex, GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2)
raydium_osd_fade_callback (void)
raydium_osd_fade_from (GLfloat * from4, GLfloat * to4, GLfloat time_len, void *OnFadeEnd)
raydium_osd_fade_init (void)
raydium_osd_internal_vertex (GLfloat x, GLfloat y, GLfloat top)
raydium_osd_logo (char *texture)
raydium_osd_mask (GLfloat * color4)
raydium_osd_mask_texture(int texture,GLfloat alpha)
raydium_osd_mask_texture_clip(int texture,GLfloat alpha, GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2)
raydium_osd_mask_texture_clip_name(char *texture,GLfloat alpha, GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2)
raydium_osd_mask_texture_name(char *texture,GLfloat alpha)
raydium_osd_network_stat_draw (GLfloat px, GLfloat py, GLfloat size)
raydium_osd_printf (GLfloat x, GLfloat y, GLfloat size, GLfloat spacer, char *texture, unsigned char *format, ...)
raydium_osd_printf_3D (GLfloat x, GLfloat y, GLfloat z, GLfloat size, GLfloat spacer, char *texture, unsigned char *format, ...)
raydium_osd_start (void)
raydium_osd_stop (void)
raydium_parser_cut (char *str, char *part1, char *part2, char separator)
raydium_parser_isdata (char *str)
raydium_parser_read (char *var, char *val_s, GLfloat *val_f, int *size, FILE *fp)
raydium_parser_replace (char *str, char what, char with)
raydium_parser_trim (char *org)
raydium_particle_callback (void)
raydium_particle_draw (raydium_particle_Particle * p, GLfloat ux, GLfloat uy, GLfloat uz, GLfloat rx, GLfloat ry, GLfloat rz)
raydium_particle_draw_all (void)
raydium_particle_find_free (void)
raydium_particle_generator_delete (int gen)
raydium_particle_generator_delete_name (char *gen)
raydium_particle_generator_enable (int gen, signed char enabled)
raydium_particle_generator_enable_name (char *gen, signed char enable)
raydium_particle_generator_find (char *name)
raydium_particle_generator_isvalid (int g)
raydium_particle_generator_load (char *filename, char *name)

raydium_particle_generator_load_internal (int generator, FILE * fp, char *filename)
raydium_particle_generator_move (int gen, GLfloat * pos)
raydium_particle_generator_move_name (char *gen, GLfloat * pos)
raydium_particle_generator_move_name_3f (char *gen, GLfloat x, GLfloat y, GLfloat z)
raydium_particle_generator_particles_OnDelete (int gen, void *OnDelete)
raydium_particle_generator_particles_OnDelete_name (char *gen, void *OnDelete)
raydium_particle_generator_update (int g, GLfloat step)
raydium_particle_init (void)
raydium_particle_name_auto (char *prefix, char *dest)
raydium_particle_preload (char *filename)
raydium_particle_scale_all(GLfloat scale)
raydium_particle_state_dump(char *filename)
raydium_particle_state_restore(char *filename)
raydium_particle_update (int part, GLfloat step)
raydium_profile_end(char *tag)
raydium_profile_start(void)
raydium_random_0_x (GLfloat i)
raydium_random_f (GLfloat min, GLfloat max)
raydium_random_i (int min, int max)
raydium_random_neg_pos_1 (void)
raydium_random_pos_1 (void)
raydium_random_proba (GLfloat proba)
raydium_random_randomize (void)
raydium_rayphp_repository_file_get (char *path)
raydium_rayphp_repository_file_list(char *filter)
raydium_rayphp_repository_file_put (char *path, int depends)
raydium_register_api(void)
raydium_register_dump (void)
raydium_register_find_name (char *name)
raydium_register_function (void *addr, char *name)
raydium_register_modifiy (char *var, char *args)
raydium_register_name_isvalid (char *name)
raydium_register_variable (void *addr, int type, char *name)
raydium_register_variable_const_f(float val, char *name)
raydium_register_variable_const_i(int val, char *name)
raydium_register_variable_unregister_last (void)
raydium_render_lightmap_color(GLfloat *color)
raydium_render_lightmap_color_4f(GLfloat r, GLfloat g, GLfloat b, GLfloat a)
raydium_rendering (void)
raydium_rendering_displaylists_disable(void)
raydium_rendering_displaylists_enable(void)
raydium_rendering_finish (void)
raydium_rendering_from_to (GLuint from, GLuint to)
raydium_rendering_internal_prepare_texture_render (GLuint tex)
raydium_rendering_internal_restore_render_state (void)
raydium_rendering_normal (void)

raydium_rendering_prepare_texture_unit (GLenum tu, GLuint tex)
raydium_rendering_rgb_force (GLfloat r, GLfloat g, GLfloat b)
raydium_rendering_rgb_normal (void)
raydium_rendering_wireframe (void)
raydium_server_accept_new (struct sockaddr *from, char *name)
raydium_sky_box_cache (void)
raydium_sky_box_render (GLfloat x, GLfloat y, GLfloat z)
raydium_sound_Array3IsValid(ALfloat *a)
raydium_sound_GetListenerOr (ALfloat * Or[])
raydium_sound_GetListenerPos (ALfloat * Pos[])
raydium_sound_GetListenerVel (ALfloat * Vel[])
raydium_sound_GetSourceDir (int src, ALfloat * Dir[])
raydium_sound_GetSourceGain (int src, ALfloat * g)
raydium_sound_GetSourcePitch (int src, ALfloat * p)
raydium_sound_GetSourcePos (int src, ALfloat * Pos[])
raydium_sound_GetSourceVel (int src, ALfloat * Vel[])
raydium_sound_InitSource (int src)
raydium_sound_LoadWav (const char *fname)
raydium_sound_SetListenerOr (ALfloat Or[])
raydium_sound_SetListenerPos (ALfloat Pos[])
raydium_sound_SetListenerVel (ALfloat Vel[])
raydium_sound_SetSourceDir (int src, ALfloat Dir[])
raydium_sound_SetSourceGain (int src, ALfloat g)
raydium_sound_SetSourceLoop (int src, signed char loop)
raydium_sound_SetSourcePitch (int src, ALfloat p)
raydium_sound_SetSourcePos (int src, ALfloat Pos[])
raydium_sound_SetSourcePosCamera(int src)
raydium_sound_SetSourceVel (int src, ALfloat Vel[])
raydium_sound_SourcePause (int src)
raydium_sound_SourcePlay (int src)
raydium_sound_SourceStop (int src)
raydium_sound_SourceUnpause (int src)
raydium_sound_SourceVerify (int src)
raydium_sound_callback (void)
raydium_sound_close (void)
raydium_sound_init (void)
raydium_sound_load_music (char *fname)
raydium_sound_music_callback (void)
raydium_sound_source_fade(int src, ALfloat len)
raydium_sound_verify (char *caller)
raydium_texture_current_set (GLuint current)
raydium_texture_current_set_name (char *name)
raydium_texture_filter_change (GLuint filter)
raydium_texture_find_by_name (char *name)
raydium_texture_load (char *filename)
raydium_texture_load_erase (char *filename, GLuint to_replace)
raydium_texture_load_internal(char *filename, char *as, signed char faked, int live_id_fake)
raydium_texture_size_is_correct (GLuint size)
raydium_timecall_add (void *funct, GLint hz)

raydium_timecall_callback (void)
raydium_timecall_clock (void)
raydium_timecall_detect_frequency (void)
raydium_timecall_devrtc_clock (void)
raydium_timecall_devrtc_close (void)
raydium_timecall_devrtc_init (void)
raydium_timecall_devrtc_rate_change (unsigned long new)
raydium_timecall_freq_change (int callback, GLint hz)
raydium_timecall_init (void)
raydium_timecall_internal_w32_detect_modulo(int div)
raydium_timecall_internal_w32_divmodulo_find(void)
raydium_timecall_raydium (GLfloat step)
raydium_trigo_abs(a) (macro)
raydium_trigo_cos (GLfloat i)
raydium_trigo_cos_inv (GLfloat i)
raydium_trigo_isfloat(a) (macro)
raydium_trigo_max(a,b) (macro)
raydium_trigo_min(a,b) (macro)
raydium_trigo_pos_get_modelview (GLfloat * res)
raydium_trigo_pos_to_matrix (GLfloat * pos, GLfloat * m)
raydium_trigo_pow2_next(int value)
raydium_trigo_rotate (GLfloat * p, GLfloat rx, GLfloat ry, GLfloat rz, GLfloat * res)
raydium_trigo_sin (GLfloat i)
raydium_trigo_sin_inv (GLfloat i)
raydium_vertex_add (GLfloat x, GLfloat y, GLfloat z)
raydium_vertex_uv_add (GLfloat x, GLfloat y, GLfloat z, GLfloat u, GLfloat v)
raydium_vertex_uv_normals_add (GLfloat x, GLfloat y, GLfloat z, GLfloat nx, GLfloat ny, GLfloat nz, GLfloat u, GLfloat v)
raydium_video_callback(void)
raydium_video_callback_video(int id)
raydium_video_delete(int id)
raydium_video_delete_name(char *name)
raydium_video_find(char *name)
raydium_video_find_free(void)
raydium_video_init(void)
raydium_video_isvalid(int i)
raydium_video_jpeg_decompress(FILE *fp,unsigned char *to)
raydium_video_open(char *filename, char *as)
raydium_window_close (void)
raydium_window_create (GLuint tx, GLuint ty, signed char rendering, char *name)
raydium_window_resize_callback (GLsizei Width, GLsizei Height)
raydium_window_view_perspective(GLfloat fov, GLfloat fnear, GLfloat ffar)
raydium_window_view_update (void)
unsupported - int v4l_yuv420p2rgb (unsigned char *rgb_out, unsigned char *yuv_in, int width, int height, int bits)
unsupported - void dump_vertex_to (char *filename)
unsupported - void dump_vertex_to_alpha (char *filename)

`unsupported - void read_vertex_from (char *filename)`
`unsupported - void v4l_copy_420_block (int yTL, int yTR, int yBL, int yBR, int u, int v, int rowPixels, unsigned char *rgb, int bits)`

Éditer cette page :: 2005-09-21 20:55:15 :: Propriétaire : JeanFran? :: Références :: Recherche : RSS

XHTML 1.0 valide ? :: CSS valide ? :: -- Fonctionne avec WikiNi 0.4.2 (interwiki) modifié par Flex