REI: An Online Video Gaming platform

Mickaël Rémond, Thierry Mallard

Erlang User Conference – 18 november 2003

Contents

1 History: Genesis of the REI project

1.1 Worldforge: Our first attempt

Worldforge (`http://www.worldforge.org/`) is a project aiming at building a platform for online role playing game. It has been launch in 1998. The project manage to produce an interesting proof of concept called Acorn. Acorn was a prototype game composed of several parts:

- *UClient:* An isometric game client.
- *Stage:* The game server supporting the Atlas protocol. The Atlas protocol is a complex interoperability protocol defined by the Worldforge project members. Another implementation of the server, called *Cyphesis* was more functionnal at this time.

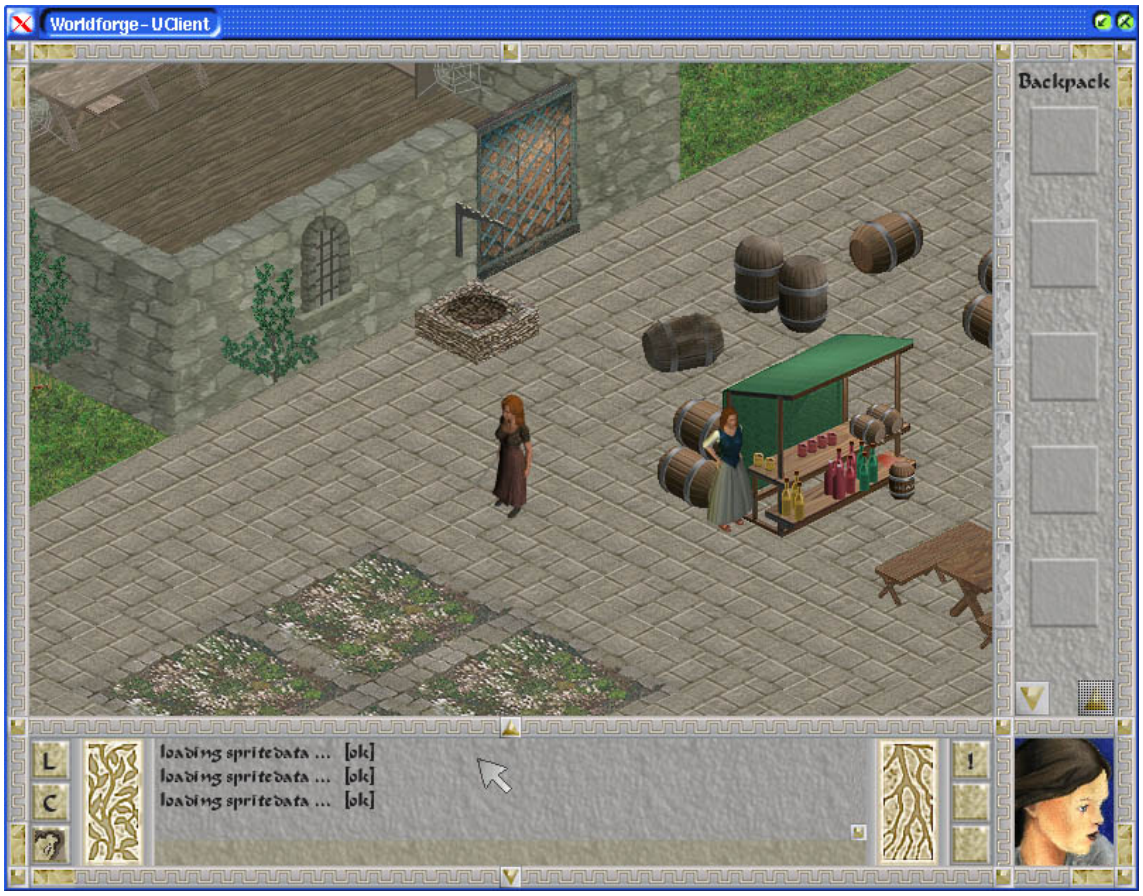

Figure 1: Worldforge Acorn screenshot - UClient

- *Libatlas:* An implementation of the Atlas protocol.

Figure 1 presents a screenshot of an online session with UClient.

In 2000, we realized that Erlang could be a middleware of choice to develop the server for the Worldforge platform. We decided to write a server called Erly-Stage to show that Erlang was a viable solution.to develop such an application. We progressed quickly and manage to run the standard UClient working on our Erlang based server. To do that, we reimplemented a small subset of the Atlas protocol, and a server to manage the game (player registration, objects synchronisation, ...). We mimic the original Stage server architecture but made turned into something that could be spawned onto a cluster of machine. The topology of the application was configurable. This was something that was not at all possible with the original Stage prototype. The application could run on one or several physical machine. Figure 2 describes the clustered Erly-Stage client-server architecture. Our server was a strict drop-down replacement of the original

server.

The result was very encouraging as our server was performing better than the original one. The main point was that, thank to Erlang soft real-time characteristics, the players synchronisation was much more accurate, leading to smoother moves management in the game.

However, we also realized that the focus of the Worldforge project was somewhat wrong. For example, the Atlas library was a huge work that is difficult to implement completly. However the value of the effort is not obvious when you start seriously working with it. There is plenty of interoperability protocol available. Atlas has no particular advantages without offering benefits for gaming, as it is both high-level and mix the transportation layer, with the data encoding. The choice of XML as a default is also not very efficient. The online gaming challenge lies mainly in being able to provide a robust and scalable platform. The architecture of such a platform is really difficult and the validation of the architecture as a whole should be the first priority of the project. On another level, the modular design was too complex regarding the actual tasks that where accomplished by the components.

At this moment we had to give up the video gaming development for the time, due to professional constraints. However, we kept on thinking on how we could do something similar more easily.

1.2 REI: An online video gaming platform

When we decided to restart a video gaming development project. Erlang was still an obvious choice for the server. We started working on a new project in december 2002: REI aims at providing a framework for online video game development. The name[1] implies that it is a testbed to validate our conception ideas.

1.2.1 Ogre

We first did some tests based on Ogre (`http://ogre.sourceforge.net/`). Ogre stands for Object-oriented Graphics Rendering Engine.

We made a client/server game prototype based on this engine. Figure 3 shows the architecture of this prototype. The expected evolutions are in the yellow area. The server code and architecture is closely described in the French Erlang book[2]. Figure 4 presents a screenshot of the REI-Ogre client prototype. The design was much simpler that the Worldforge design. We concentrated on messages dispatching with optional validation[3].

[1] Rei means prototype in Japanese.

[2] Erlang programmation, Mickaël Rémond, Editions Eyrolles, Collection Coming Next, 2003

[3] We did not get further on this aspects but the most interesting part here is finding a proper algorithm to do validation that does not prevent the game for running smoothly. The client should assume that its action are validated by the server without any explicit correction notification. If everything goes well the game is thus smooth. The game is only disturbed if, for some reasons, the server does not agree on the client actions.

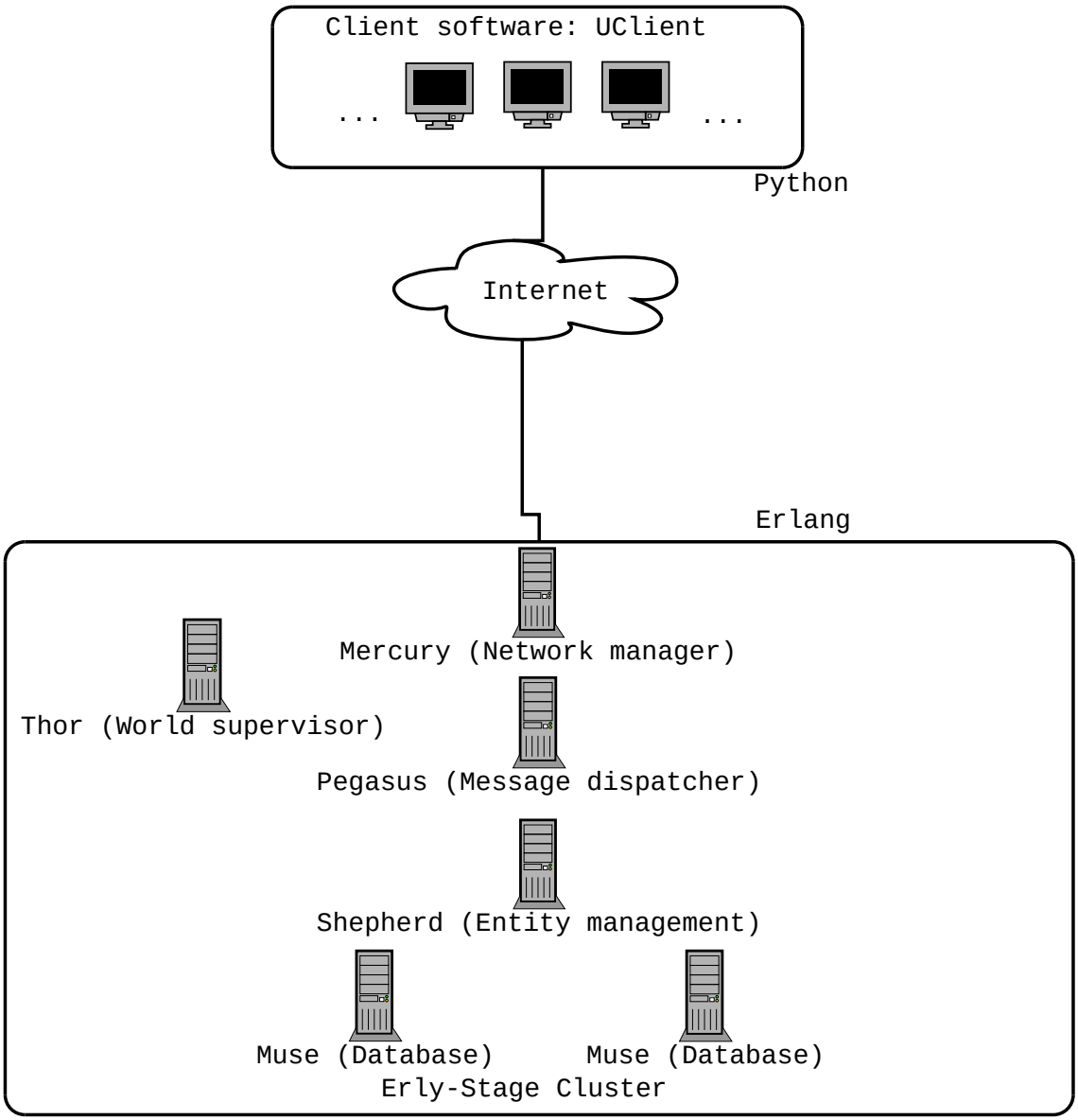

Figure 2: Erly-Stage client-server architecture

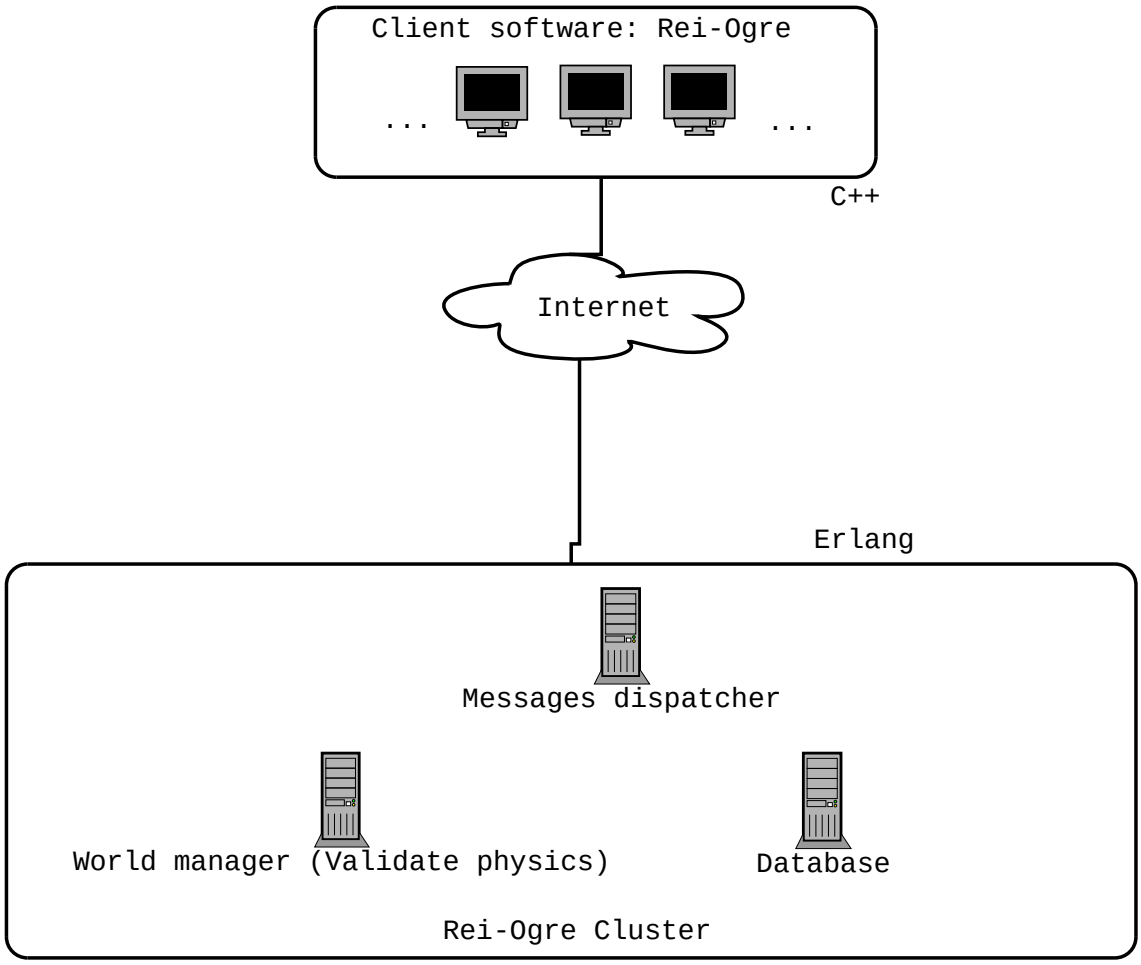

Figure 3: REI-Ogre architecture

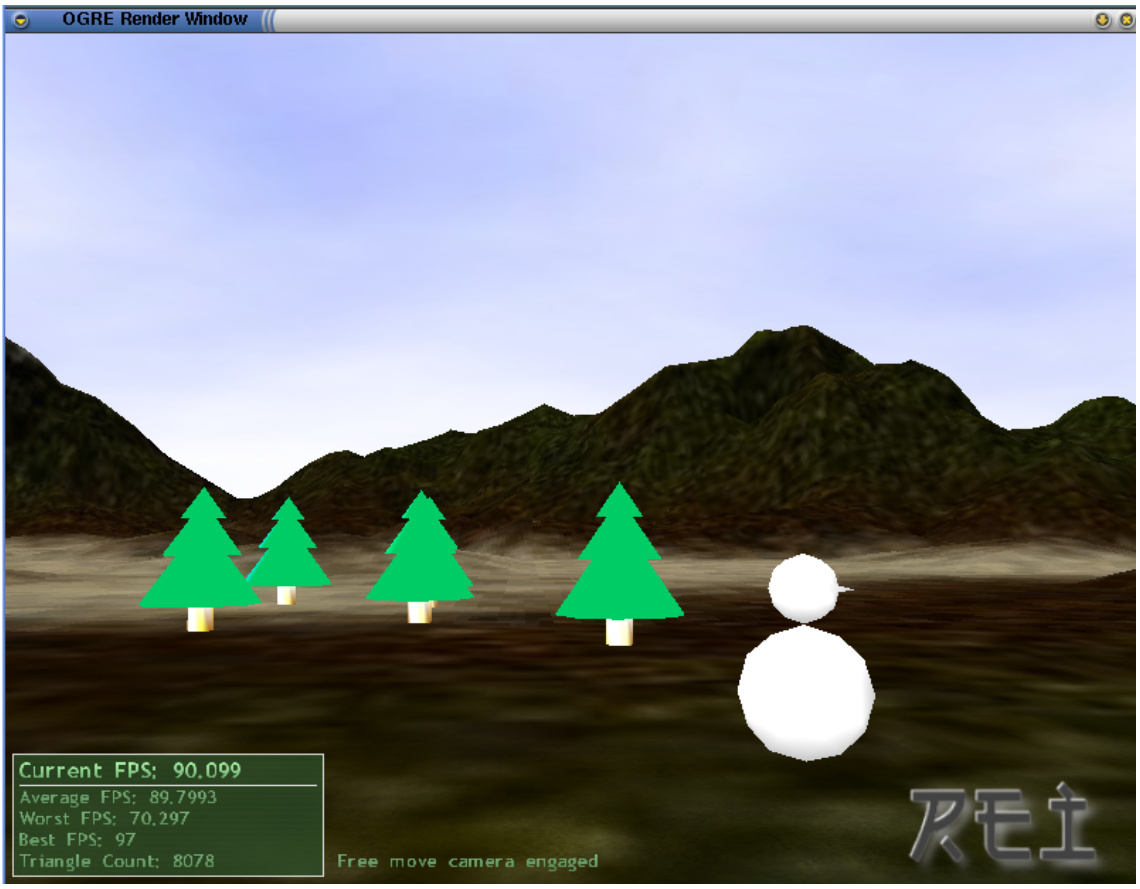

Figure 4: REI-Ogre screenshot

Everything was at the end working correctly. However, there was two main problems:

- It was difficult to keep in touch with the development of the software. Many extensions were not well maintained prototypes.
- Mechanism for writing extensions was not very modular.
- It was impossible to launch the engine without the rendering, for doing only calculation or collision detection.

The last argument was in itself a no-go, because it became obvious that the engine would be needed on the server too, mainly to handle heavy duty calculation.

Ogre is a promising 3D engine, but its main drawback is being developed in a really messy way and we found it hard to manage to found our way through the code and make everything working. Components and plugins often evolve at different pace and it is difficult to make everything stand together. The toolset is also hard to build, because Ogre is not using standard format for meshes for example. Ogre can produce some really amazing render, but proved a little bit to experimental for us.

Figure 5 shows a screenshot of VTrike, a brazilian commercial game using Ogre engine.

1.2.2 Nebula Device

Nebula Device is a very professional engine. It is used for real life work and it shows. It is very modular and very well thought. Its architecture is service oriented. You can decide to start some services without using some others. The rendering is one of this service and you can use the engine without asking for any rendering. This means that this is adequat for running on a server as a physics engine. Having the same engine on the server and the client is also a good way to be sure that physics algorithms used are the same.

We decided to give it a try. We first wrote a prototype equivalent to Rei-Ogre. The server was the same. The client was however not written in C++. The Nebula Engine itself was made to be leverage by scripting language such as TCL, Python and LUA. It made sense then to use a high level language to build the prototype. The performance bottleneck where already taken into account in the engine: The engine was indeed implementing the CPU power hungry calculation. That is why the client was written in Python, using the Nebula Device Engine.

We were disappointing by the result of the client, at the time when we introduced the networking aspects. Multithreading was not properly handled by the Python binding and we did not manage to make the prototype work smoothly with server.

We then realized that our target architecture was the following:

It was thus necessary to build an Erlang Nebula binding to compute physics on the server (collision detection, trajectory calculation). The server is supposed to detect any

Figure 5: VTrike (Ogre 3D engine)

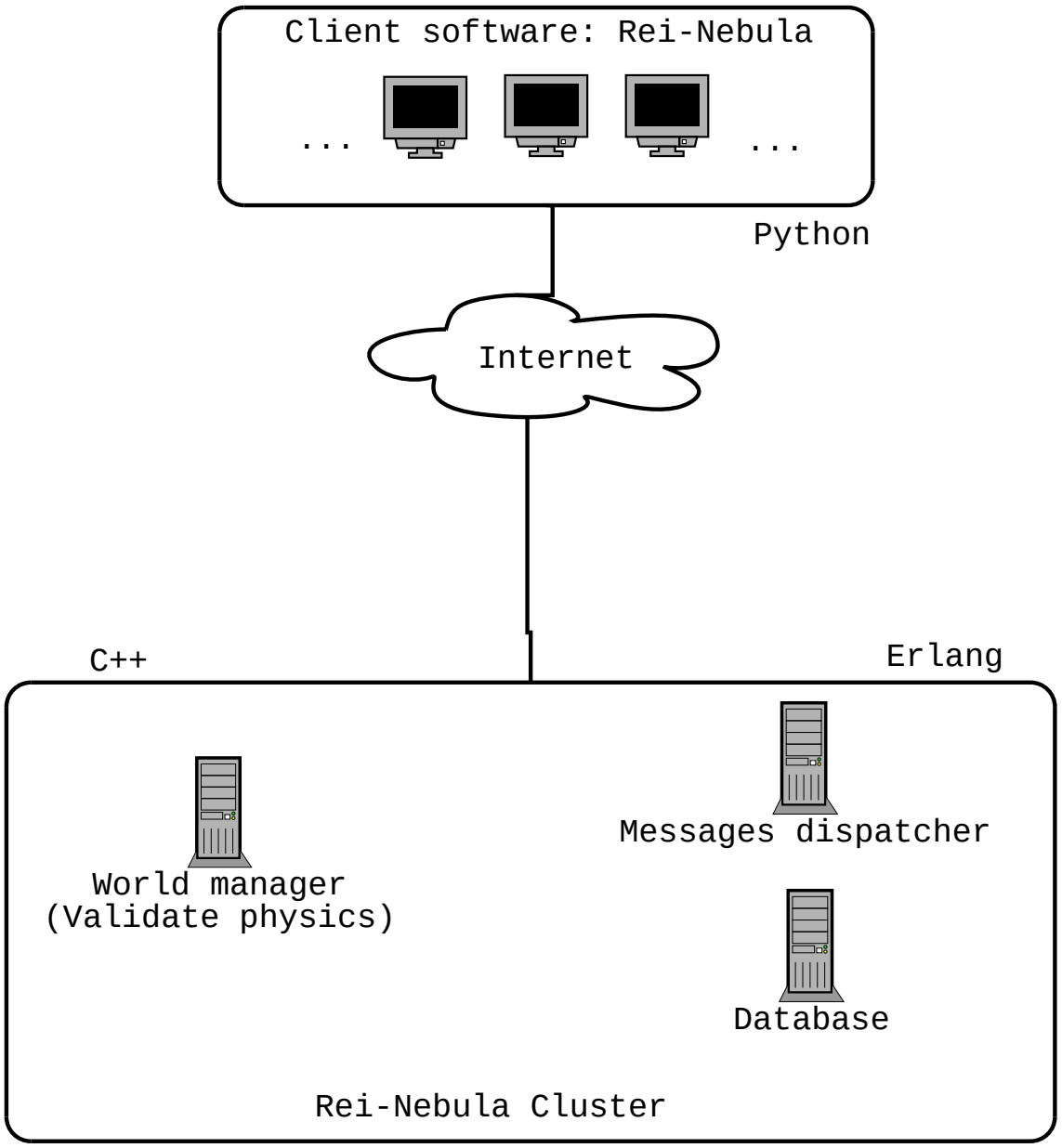

Figure 6: Rei-Nebula first architecture

mistake or “fraud” on the client. It is then tempting to try putting Erlang on the client side to benefits from its concurrency and networking capability. We were getting to such an architecture:

That is what we did. We wrote an Erlang-Nebula binding. The result is very positive with several lesson:

- Nebula Device has been thought to make it easy to add new scripting language support. When you have implemented the basic functions in the framework, you take advantage of the whole thing (Debug console, ...), complete API access.
- You can benefits from Erlang concurrency model on the client. We made some try with many concurrent Erlang process managing several objects in the 3D world and this is working properly. The networking concurrent problems are thus solved by the use of Erlang concurrency aspects.

1.3 REI: Big picture and next steps

REI is intended to be much more than a 3D engine combined with a powerful concurrent language. We aims at building a complete Online Game conception and hosting platform. The online Game Development framework includes:

- *3D world creation framework:* The framework needs to enable huge world and take into account the fact that players are numerous and might need to be distributed into world areas.
- *3D game administration platform:* One of the main problem when running a 3D gaming platform are evolution of the software (server and clients) to fixe rules and add new player ability to support a continuously evolving scenario. The software and platform behaviour has to be upgraded while minimizing the system downtime.
- *Game back-office:* Online games and especially massively online multiplayer games are managing a lot of contents that should be produce in a collaborative but controled way. To do that we need a back-office that can manage the workflow of:
  - Game media (Image, models, ...)
  - Rules (as scripts)

  The back-office will handle upgrade and diffusion of the various game components.

2 Focus on the Nebula-Erlang binding

The heart of the project is currently the Nebula Erlang binding. It allow to develop 3D programs in Erlang and more specifically games.

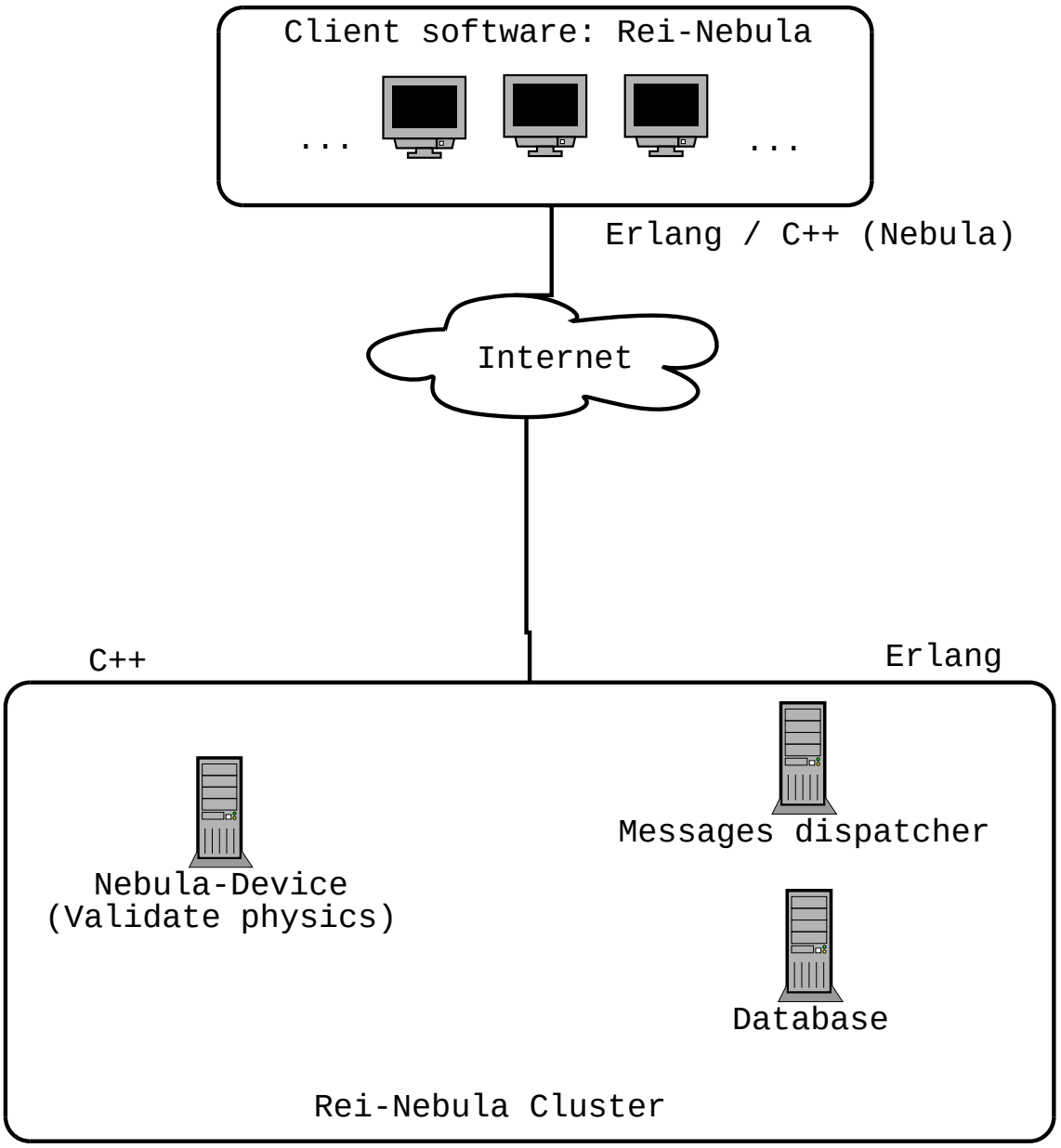

Figure 7: Rei-Nebula second architecture

2.1 Installing Nebula-Erlang

You need the following things to play with the Nebula-Erlang binding:

- Nebula-Device,
- Nebula-Device contributions (basically for the game framework),
- Erlang/OTP,
- The Nebula-Erlang binding itself.

Nebula itself depends on several libraries, such as DEVIL (Imaging Library) or TCL/TK 8.4 for example. You should first make sure that you can compile Nebula[4] before trying to compile the Nebula Erlang binding[5].

2.2 Basics concepts

There are two things to know to get started with Nebula-Device:

- Nebula-Device API relies on a string-based protocol,
- Nebula-Device objects are organized as a hierarchy where an object identifier is a path to the object in the hierarchy.

2.2.1 String based protocol

Nebula-Device is essentially based on a string based protocol. You have very few functions call to understand to get started. The parameters are coded as strings that you pass as parameters to the function.

- `new(TypeOfObject, ObjectId)`: This function create an object of a given type with a user defined object Identifier.
- `cmd(ObjectId, FunctionName, ParametersList)`: This function call a function on the given object, identified by its ObjectId, with a list of parameters.

2.2.2 Object hierarchy

Nebula-Device is build around the idea of hierarchy. Objects are place in the system hierarchy. Some paths are conventional place-holder for typical objects (like servers), but your are most of the time free to organise the object hierarchy to your liking.

[4] In `nebula/code/src`, do a `tclsh8.4 updsrc.tcl` and then a `make`.
[5] Launch the `rei.sh` command from the root of the Nebula-Erlang binding archive.

2.2.3 Code example

Here is an example piece of code that shows the creation of three objects and function call on it:

```
...
new("n3dnode", "/usr/object1"),
new("nmeshnode", "/usr/object1/mesh"),
cmd("/usr/object1/mesh", "setfilename", [ "torus.n3d" ]),
cmd("/usr/object1", "txyz", [ "10", "10", "10" ]),
...
```

2.3 Nebula-Device service architecture

Nebula's architecture is based on services (known as *servers*) which are plugged above a kernel. Every services are shown as part of the Nebula Object Hierarchy. This is similar to a filesystem. For instance, servers are often placed in the `/sys/servers` directory.

Most services are available on every plateform, but there are some exceptions.

For example, the sound server isn't available on Linux at the moment.

2.3.1 Primary servers

On most Nebula programs, the following servers are loaded :

- nglserver: `/sys/servers/gfx`

  This is an OpenGL graphic server. There's also a Direct3D server, which can be placed at the same place instead of the OpenGL one.

  The graphic server controls several important attributes such as:

  - Graphic mode (windowed or fullscreen),
  - Screen or window resolution.

- ninputserver: `/sys/servers/input`

  This server also keymapping and mouse handling.

- nscenegraph2: `/sys/servers/sgraph2`

  The core scene graph handler.

- nfileserver2: `/sys/servers/file2`

  A file server. This allows some abstraction between the real filesystem and the application.

  For instance, you can register a symbolic directory (`mysystem: ==> /usr/local/mygame`) and then use it later on (i.e. loading `mysystem:/img/logo.jpeg`)

2.3.2 Secondary servers

Some other secondary servers may be loaded :

- nsbufshadowserver: `/sys/servers/shadow`
  Shadowing service
- nchannelserver: `/sys/servers/channel`
  Channel server, which allows timestamping and flow control. This service is for example used by interpolators. This is mainly a synchronization service.
- nconserver: `/sys/servers/console`
  A console server, which exports an abstraction to the scripting language
- nmathserver: `/sys/servers/math`
  The mathematic server, for specific computing
- nparticleserver: `/sys/servers/particle`
  A particle server, for smoke effect for instance
- nspecialfxserver: `/sys/servers/specialfx`
  Some S/FX server. Lens flare for example

In a small default framework (called `ndefault`), we have created an observer libraries helper that helps pushing building simple scene.

2.4 Simple examples

2.4.1 Nebula Erlang: Hello World!

Here is an example of the most simple Nebula program you can do:

```
-module(simple).
-export([start/0]).
start() ->
  nebula:start(), % Start the Nebula engine
  ndefault:start(), % Setup a default "world"
  Object = "/usr/scene/object",
  MeshFileName = "torus",
  %% Create object, and add mesh and shader
  nebula:new( "n3dnode", Object ),
  ndefault:mesh( Object, MeshFileName ),
  ndefault:shader( Object ),
  ndefault:eventloop().
```

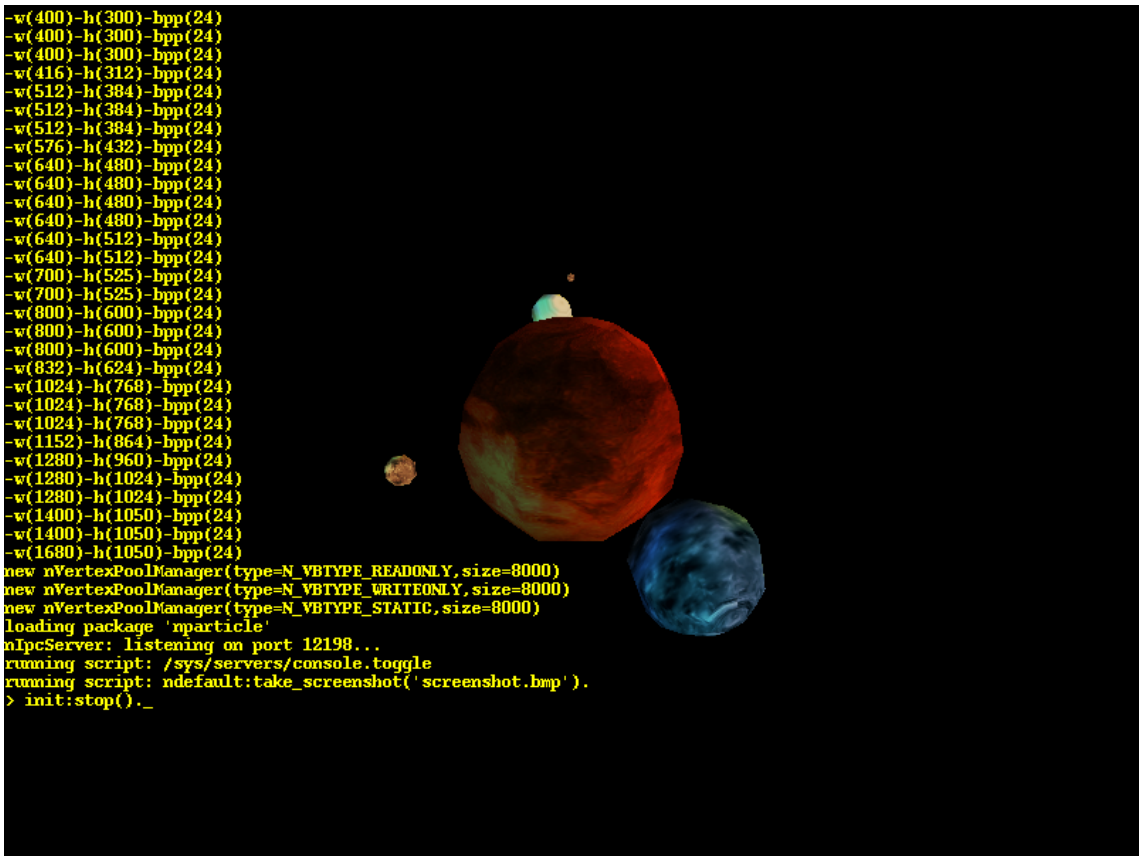

Figure 8: Using the in-game Nebula console

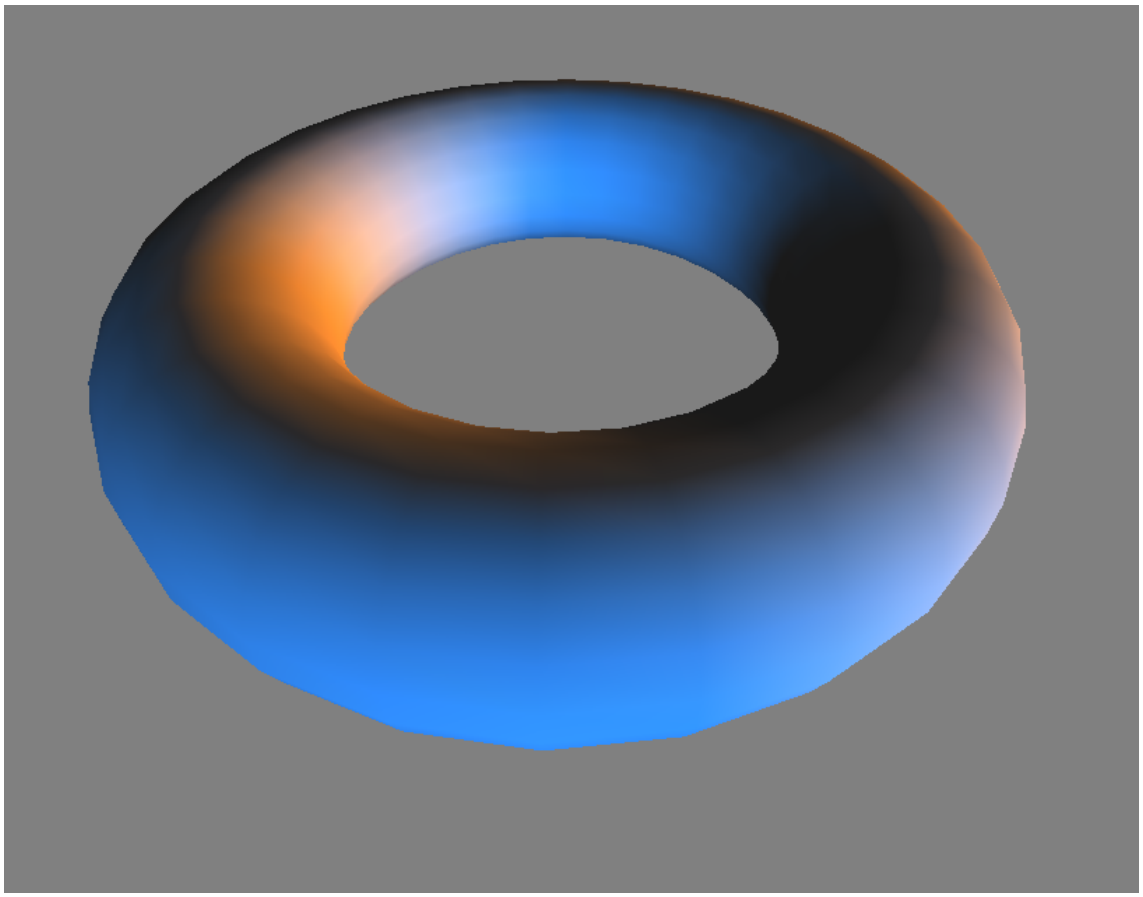

Figure 9: `simple:start/0` execution result

Most of the low-level work is done by the `ndefault` module. You should ultimately have a look to this module code to get a better understanding of the Nebula engine. This is however not necessary to start doing fun things with the engine.

2.4.2 Objects creation

Objects are composed of meshes and shaders. Meshes are ALIAS WAVEFRONT file formatted data. Shaders are pieces of code that describe the rendering setup of the object in the Nebula Engine formalism.

- Mesh: This is the 3D description of the object. The mesh defines the triangles of the object. The mesh is usually defined in .n3d files, which are simple ALIASWAVEFRONT (`.obj`) mesh definition. You can optimize the mesh for loading in the 3D engine thanks to various command line tools provided with the Nebula framework.
- Shader: This is how the object should be represented. It defines the material of the object: is it a wooded object or a metal object ? How does the object reflects lights ? ...

  Optionnaly, an object can reference texture (*ntexarray*) that can in be used in the shader definition.

To create an object, you need to define both a mesh and a shader. The `ndefault` module provide two helper functions:

- mesh(ObjectId, MeshFile)
- shader(ObjectId)

The following code is suffisant to minimally define an object:

```
...
Object = "/usr/scene/object",
MeshFileName = "torus",
nebula:new( "n3dnode", Object ),
ndefault:mesh( Object, MeshFileName ),
ndefault:shader( Object ),
...
```

2.4.3 Mesh creation: using Wings3D to generate Nebula Objects

Wings3D is a polygon mesh modeler written in Erlang. This is a perfect tool to generate models that can be imported into the Nebula Device. It is well suited for low-polygons object creation.

You have to follow the following steps if you want to import Wings3D models into the engine:

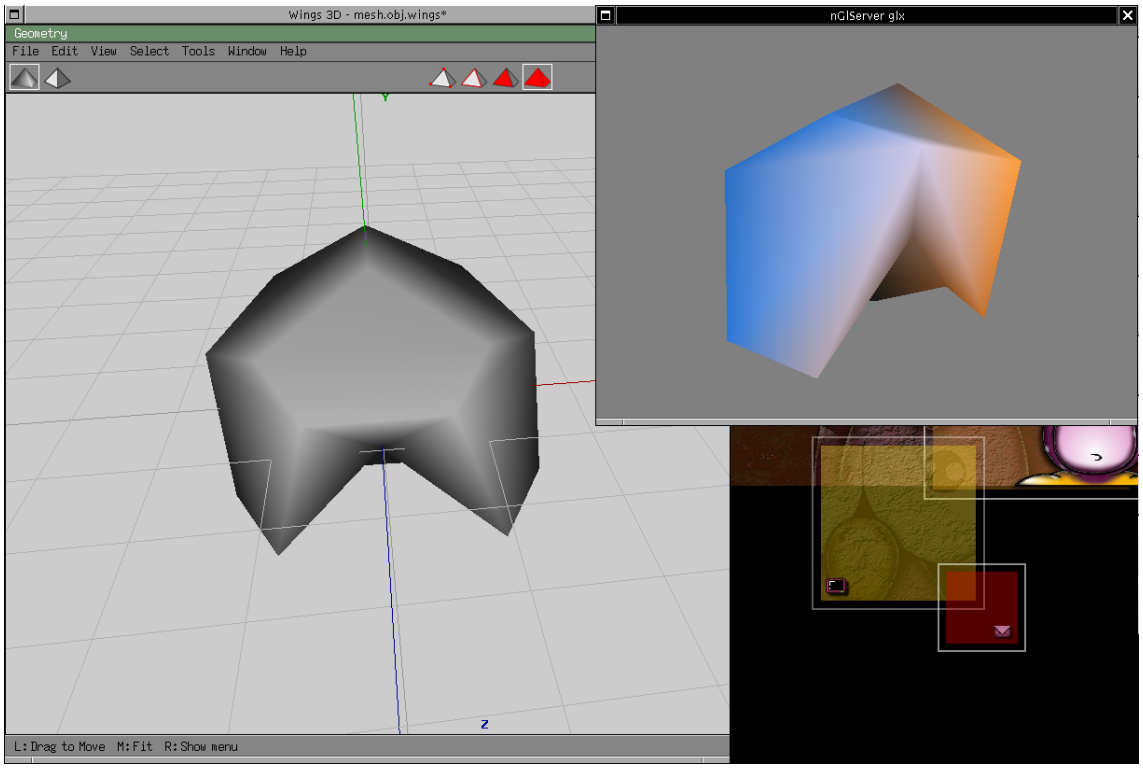

Figure 10: An object model in Wings3D and the same model used in an Erlang Nebula program

1. Save your Wings3D model into the ALIAS WAVEFRONT file format (i.e.: `mesh.obj`).
2. Clean-up the model using wftools provided with Nebula:

   ```
   wfclean <mesh.obj | wftriang | wfflatten >mesh.n3d
   ```

3. Check that your object can be properly loaded into the Erlang Nebula world with Erlang mesh viewer:

   ```
   nmesh_viewer mesh.n3d
   ```

It is still needed to write a Nebula exporter for Wings3D that will generate Nebula shaders based on how materials are defined in the modeler. This is an important feature on the project todo list.

2.4.4 Complex shader definition

In the following example, the `ndefault` shader function has been replaced by custom shaders definition, whose name is passed as parameters to the start function:

```
-module(shader).
-export([start/1]).
-export([default/1, envir/1]).
-import( nebula, [new/2, cmd/3, mcmd/2] ).

start(ShaderFunction) ->
  nebula:start(), % Start the Nebula engine
  ndefault:start(), % Setup a default "world"

  Object = "/usr/scene/object",
  MeshFileName = "torus",

  %% Create object, and add mesh and shader
  nebula:new( "n3dnode", Object ),
  ndefault:mesh( Object, MeshFileName ),
  ?MODULE:ShaderFunction( Object ),

  ndefault:eventloop().
```

Shader defines materials behaviour regarding light. Here is an example setting object shader:

```
default( Object ) ->
  Shader = Object ++ "/shd",
  new( "nshadernode", Shader),
  mcmd( Shader,
   [
    { "setnumstages", [ "0" ] },
    %% Global color / light setting
    { "setdiffuse", [ "1.0", "1.0", "1.0", "1.0"]},
    { "setemissive", [ "0.0", "0.0", "0.0", "0.0"]},
    { "setambient", [ "0.5", "0.5", "0.5", "0.5"]},
    { "setlightenable", [ "1" ]},
    { "setalphaenable", [ "0" ]}
   ]).
```

Figure 9 presents the default shader rendering.

Here are some explaination regarding shader setting:

- `setlightenable` takes “1” or “0” as parameter. It defines if the object shadow dark side are influenced by scene light position.
- `setalphaenable` relates to object transparency. It takes “1” or “0” as parameter.

- `setdiffuse`, `setemissive` and `setambient` define the object diffuse, emissive and ambient reflection lighting. They take for string parameters (RGB colors from 0 to 1, plus an alpha channel).
- `setnumstages` is used to specify the number of color texture layers (or rendering stages) the object has. In very simple cases, it is “0”. In complex cases, it can be “3” of more.

You can of course use raw bitmap files as texture layer for the object. The following code use two operation to define the object color:

```
envir( Object ) ->
  Shader = Object ++ "/shd",
  new( "nshadernode", Shader),
  mcmd( Shader,
   [
    { "setnumstages", [ "2" ] },

    { "setcolorop", [ "0", "mul tex prev" ]},
    { "setcolorop", [ "1", "ipol tex prev const.a" ]},

    { "begintunit", [ "0" ]},
    { "setaddress", [ "wrap", "wrap"] },
    { "setminmagfilter", ["linear", "linear"] },
    { "settexcoordsrc", [ "uv0" ] },
    { "setenabletransform", [ "1" ] },
    { "endtunit", [] },
    { "sxyz0", [ "2", "2", "2"]},

    { "begintunit", [ "1" ]},
    { "setaddress", [ "wrap", "wrap"] },
    { "setminmagfilter", ["linear", "linear"] },
    { "settexcoordsrc", [ "spheremap" ] },
    { "setenabletransform", [ "1" ] },
    { "endtunit", [] },
    { "setconst0", [ "0.2", "0.2", "0.2", "0.2" ] },
    % Global color / light setting
    { "setdiffuse", [ "1.0", "1.0", "1.0", "1.0"]},
    { "setemissive", [ "0.0", "0.0", "0.0", "0.0"]},
    { "setambient", [ "1.0", "1.0", "1.0", "1.0"]},
    { "setlightenable", [ "1" ]},
    { "setalphaenable", [ "0" ]}
]),

  Texture = Object ++ "/tex",
  new( "ntexarraynode", Texture ),
```

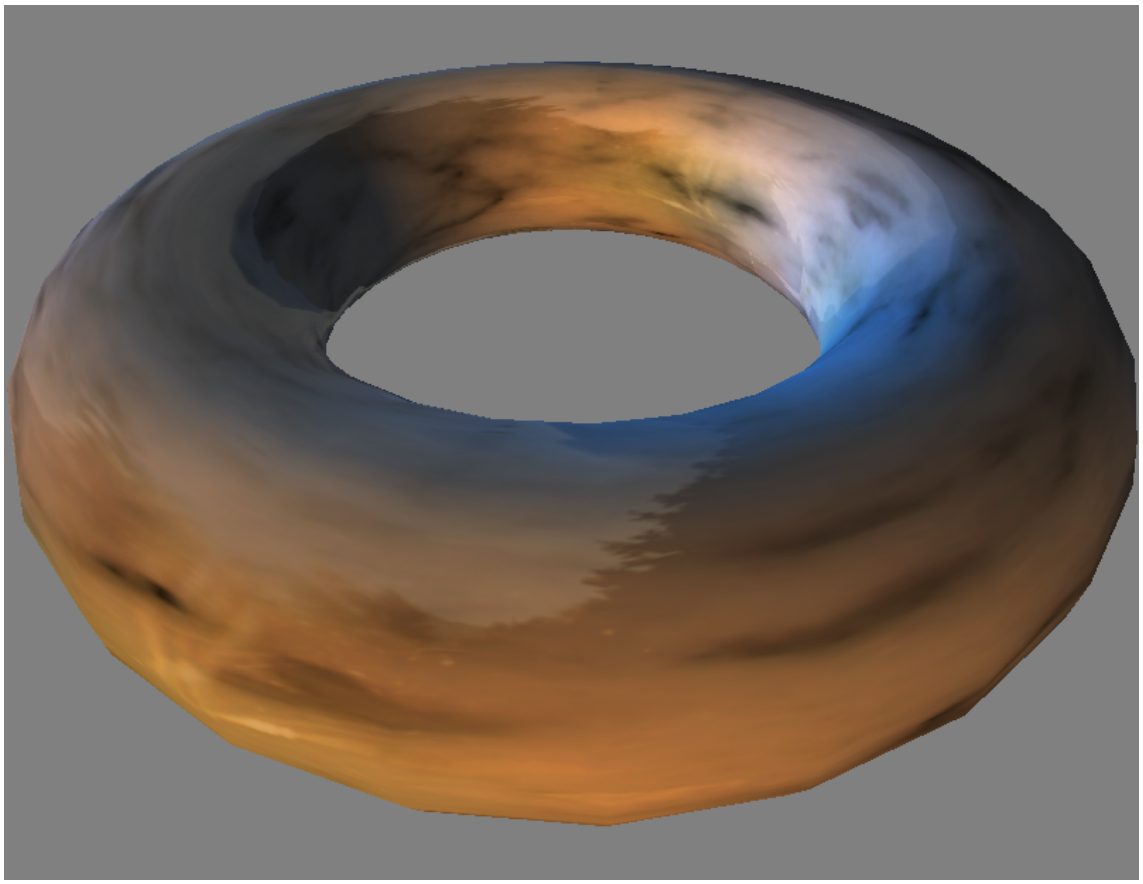

Figure 11: `shader:start(envir).`

```
mcmd( Texture,
 [
  { "settexture", [ "0", "bmp/marble.bmp", "none" ]},
  { "settexture", [ "1", "bmp/autobahn.bmp", "none" ]}]).
```

Nebula documentation describes how shader rendering stages are working:

> *`setcolorop` specifies a pixel shader operation which works with the rgb component of a color (for each stage of the shader, a separate pixel and alpha operation may be defined). The syntax for a valid instruction is: op[.1,.2,.4] [-]arg0[.a,.c] [[-]arg1[.a,.c]] [[-]arg1[.a,.c]] The following operations are defined:*
>
> – *replace -> replace color with arg0*

– mul -> multiply (modulate)
– add -> add unsigned
– adds -> add signed
– ipol -> interpolate
– dot -> dot3 (matrix operation)

The dot operation may not be available on all older hardware. Where it isn't available, it is replaced by a multiply.

The following optional postfixes may be appended to the opcodes to manipulate the final result:
– .2 -> multiply the result by 2
– .4 -> multiply the result by 4

Please note that those postfixes are not supported by all host systems.

Valid values for 'Arg' are:
– tex -> the result of the texture read operation of this stage
– prev -> the result of the previous pixel shader stage
– const -> the color constant defined for this stage (see 'setconst')
– prim -> the untextured "base pixel" from the lighting equation

Argument strings can be prefixed by a '-' (minus sign) to invert the argument before it goes into the operation.

Argument strings can be postfixed by a '.c' or '.a' to explicitly select the rgb component (.c) or alpha component (.a) of the pixel. The default is '.c' for .setcolorop and '.a' for .setalphaop.

`settexcoordsrc` *sets the source where this texture unit should get its uv coordinates from. Valid values are:*
– uv0..uv3 - the vertex buffer uv coordinate streams 0..3
– objectspace - generate on the fly from object space coordinates
– eyespace - generate on the fly from eye space coordinates
– spheremap - generate on the fly for spherical env mapping

Please note that texture bitmap definition are made, by object layer, in a `ntexarraynode` object.

Many other parameters can be controled, such as:

- zbuffer configuration,
- wireframe object rendering,
- cullmode for controlling hidden faces,
- ...

Please refer to Nebula Device documentation for more details.

2.5 More advanced topics

2.5.1 Simple objects movements with interpolators

Nebula Device provides a straight-forward way to animate the object in the scene. You can attach interpolators to 3D nodes to define their movement, rotation, scale evolution, ... You can attach as many interpolator as you want on Nebula nodes.

Several kinds of interpolators are available:

- linear: Does a linear interpolation between keyframes.
- step: Steps between keyframes, no interpolation.
- quaternion: Spherical quaternion interpolation. Only makes sense when connected to a command that takes quaternions.
- cubic: Cubic interpolation.
- spline: Catmull-Rom spline interpolation.

Interpolation is usually made according to the time channel information[6]. Interpolator can manage one to four dimensions, depending on the function you want it to connect to. For example, one axis rotations rely on unidimensional interpolator.

The following code illustrate interpolators use:

```
-module(ipol).
-export([start/0, start/1]).
-export([ipol/1, ipolx/1, ipoly/1, ipolz/1]).
-import( nebula, [new/2, cmd/3, mcmd/2] ).
start() ->
  start(ipol).
start(IpolFunction) ->
  %% Setup a default "world"
  scene_setup(),
  Object = "/usr/scene/object",
  Meshname = "torus",
  %% Create object, and add mesh and shader
  nebula:new( "n3dnode", Object ),
  ndefault:mesh( Object, Meshname ),
  ndefault:shader( Object ),
  ?MODULE:IpolFunction( Object ),
  ndefault:eventloop().
%% Setup default environment
scene_setup() ->
```

[6] You could however write your own channel to tweak the interpolator behaviour.

```
  nebula:start(),
  ndefault:start().
%% Plays with object interpolation engine
%%% 3D Rotation
ipol( Object ) ->
  Ip = Object ++ "/ip",
  new( "nipol", Ip),
  mcmd( Ip,
  [
   { "connect", [ "rxyz" ] },
   { "addkey3f", [ "0.0", "0.0", "0.0", "0.0"]},
   { "addkey3f", [ "10.0", "360.0", "1080.0", "720.0"]}
]).
ipolx( Object ) -> ipol1( Object, "rx" ).
ipoly( Object ) -> ipol1( Object, "ry" ).
ipolz( Object ) -> ipol1( Object, "rz" ).
ipol1( Object, RotationAxis ) ->
  Ip = Object ++ "/ip",
  new( "nipol", Ip),
  mcmd( Ip,
  [
   { "connect", [ RotationAxis ] },
   { "addkey1f", [ "0.0", "0.0"]},
   { "addkey1f", [ "10.0", "1080.0"]}
  ]).
```

Depending on the atom parameters you pass to the start function (`ipol`, `ipolx`, `ipoly`, `ipolz`), the torus, will rotate on three or only one rotation axis.

2.5.2 Hierarchical objects management

Interpolators are attached to 3D nodes. This means that you can use anyware in your 3D object hierarchy. This makes it easy to define movement for a whole hierarchy of 3D objects.

Objects managed in the Nebula world are placed in a pseudo file system that compose a tree of objects. The tree is an handy way to order your objects in the world but it is also a way to apply modification to group of objects. “Transformations” can be applied to objects that are placed at deepest levels in the world hierarchy.

The following example shows complex moves of objects into several referencials. This example presents the rotation of several planets around several axis.

```
-module(planets).
-export([start/0]).
-import( nebula, [new/2, cmd/3, mcmd/2] ).
```

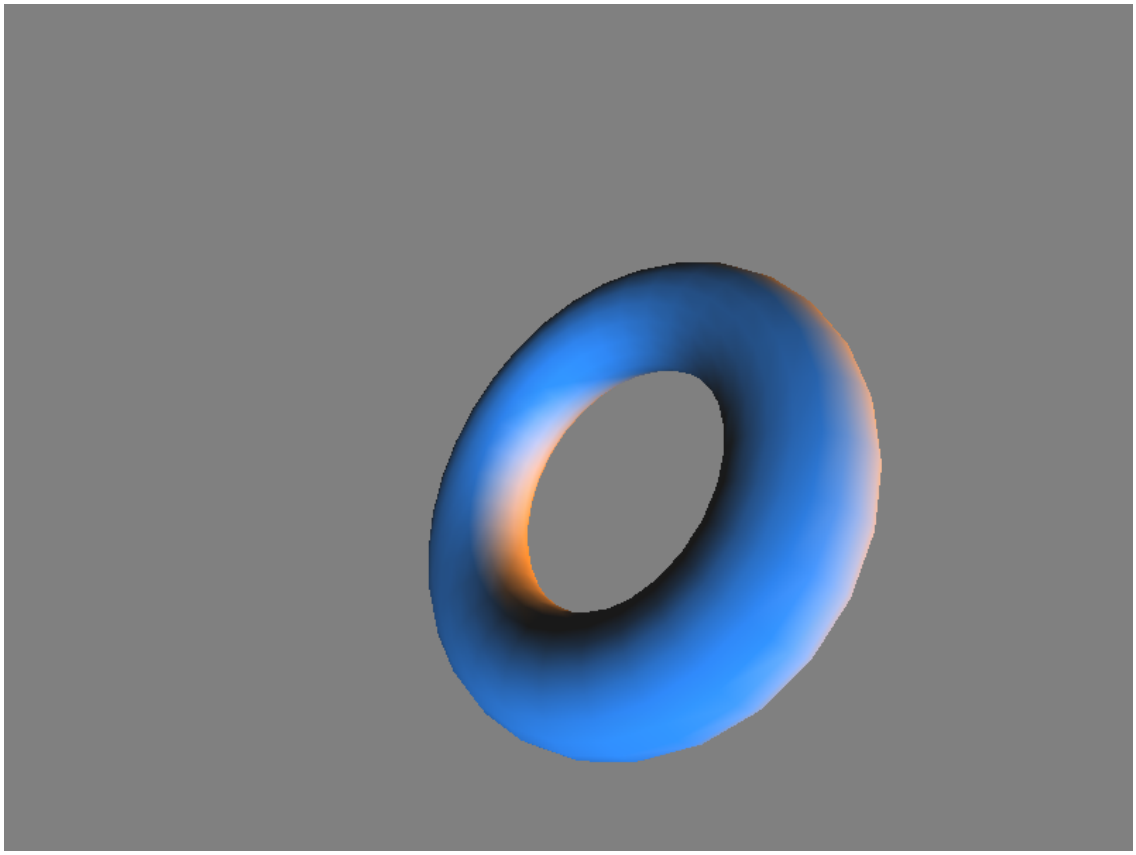

Figure 12: Rotation of the torus object on three axis (`ipol`)

```
start() ->
    %% Setup a default "world"
    scene_setup(),
    cmd( "/sys/servers/gfx", "setclearcolor", [ "0",  "0", "0", "0" ] ),
    Scene = "/usr/scene",
    Sun = Scene ++ "/sun",
    plainobject( Sun, "meshes/smooth.n3d", "bmp/lava.bmp" ),
    rot( Sun, "20", "ry" ),
    Planet1 = Sun ++ "/planet1",
    plainobject( Planet1, "meshes/smooth.n3d", "bmp/pla2.bmp" ),
    mcmd( Planet1,
      [
       {"tx",   ["2"]},
       {"sxyz", ["0.4", "0.4", "0.4"]}]),
    rot( Planet1, "10", "ry" ),
    rot( Planet1, "5", "rz" ),
    ExtraRot1 = Sun ++ "/extrarot1",
    new( "n3dnode",  ExtraRot1 ),
    cmd( ExtraRot1, "rx", ["-5"] ),
    rot( ExtraRot1, "25", "ry" ),
    Planet2 = ExtraRot1 ++ "/planet2",
    plainobject( Planet2, "meshes/smooth.n3d", "bmp/pla3.bmp" ),
    mcmd( Planet2,
      [
       {"tx",   ["3"]},
       {"tz",   ["2"]},
       {"sxyz", ["0.3", "0.3", "0.3"]}]),
    rot( Planet2, "6", "ry"),
    rot( Planet2, "10", "rx"),
    Moon1 = Planet2 ++ "/moon1",
    plainobject( Moon1, "meshes/smooth.n3d", "bmp/pla1.bmp" ),
    mcmd( Moon1,
      [
       {"tx",   ["1.5"]},
       {"tz",   ["2"]},
       {"sxyz", ["0.3", "0.3", "0.3"]}]),
    ExtraRot3 = Planet2 ++ "/extrarot3",
    new( "n3dnode", ExtraRot3 ),
    rot(ExtraRot3, "4", "ry"),
    Moon2 = ExtraRot3 ++ "/moon2",
    plainobject(Moon2, "meshes/smooth.n3d", "bmp/pla1.bmp"),
    mcmd( Moon2,
      [
       {"tx",   ["3"]},
       {"sxyz", ["0.2", "0.2", "0.2"]}]),
    ExtraRot2 = ExtraRot1 ++ "/extrarot2",
```

```
    new( "n3dnode", ExtraRot2 ),
    cmd( ExtraRot2, "rx", ["25"] ),
    rot(ExtraRot2, "15", "ry"),
    Planet3 = ExtraRot2 ++ "/planet3",
    plainobject( Planet3, "meshes/smooth.n3d", "bmp/pla1.bmp" ),
    mcmd( Planet3,
      [
       {"tx",   ["-1.3"]},
       {"tz",   ["-1.3"]},
       {"sxyz", ["0.1", "0.1", "0.1"]}]),
    rot(Planet3, "1", "rz"),
    ndefault:eventloop().
%% Setup default environment
scene_setup() ->
    nebula:start(),
    ndefault:start().
%% Interpolator factory
rot( Father, Time, Channel ) ->
    Ipol = Father ++ "/" ++ Channel,
    new( "nipol", Ipol),
    mcmd( Ipol,
          [
           { "connect",  [ Channel ] },
           { "addkey1f", [ "0.0", "0.0" ]},
           { "addkey1f", [ Time, "360.0" ]}
          ]).
% Customed object creation function
plainobject( ObjectId, MeshFile,  TextureFile ) ->
    new( "n3dnode", ObjectId ),
    Mesh = ObjectId ++ "/mesh",
    new( "nmeshnode", Mesh ),
    cmd( Mesh, "setfilename", [ MeshFile ] ),
    Shader = ObjectId ++ "/shader",
    new( "nshadernode",  Shader ),
    mcmd( Shader,
      [
       { "setnumstages", ["1"] },
       { "setcolorop", ["0", "mul tex prev"] },
       { "begintunit", [ "0" ]},
        { "setaddress", [ "wrap", "wrap"] },
        { "setminmagfilter", ["linear", "linear"] },
        { "settexcoordsrc", [ "uv0" ] },
        { "setenabletransform", [ "0" ] },
       { "endtunit", [] },
       { "setlightenable", [ "1" ]},
       { "setdiffuse",  [ "1.0", "1.0", "1.0", "1.0"]},
```

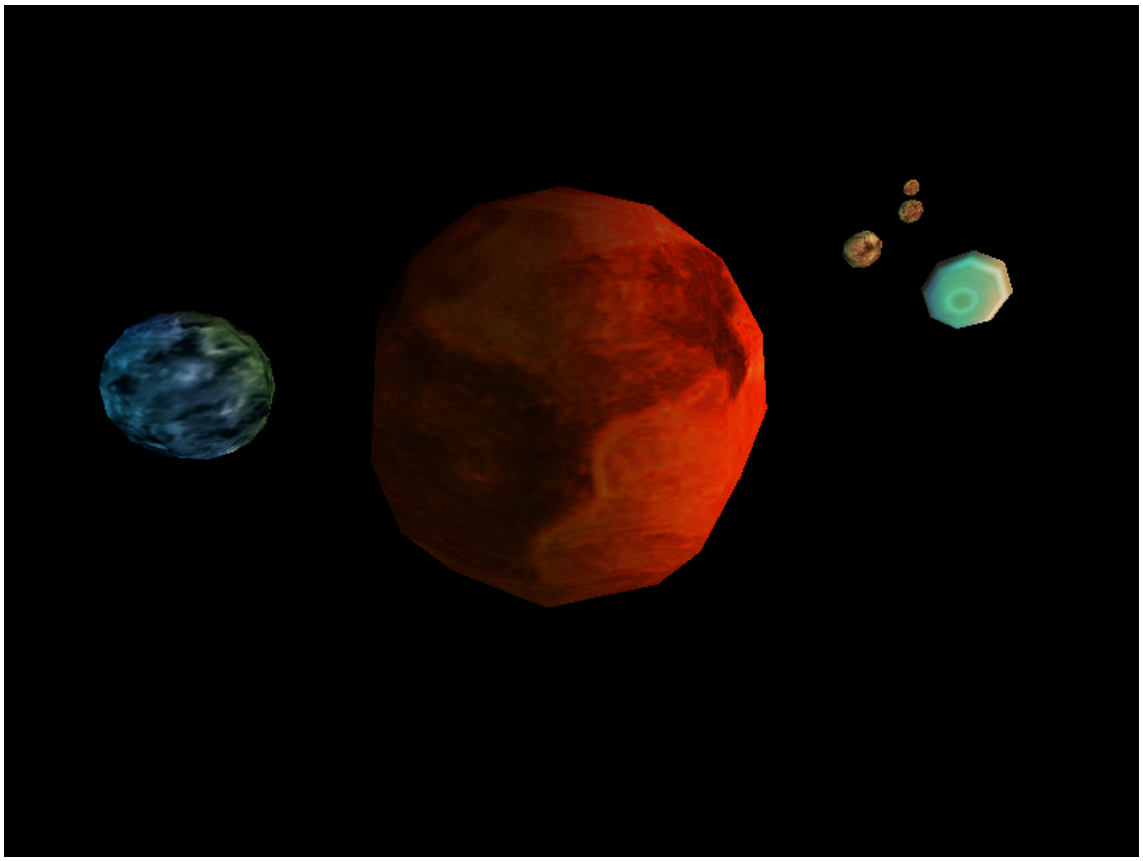

Figure 13: Planets' revolutions

```
 { "setemissive", [ "0.0", "0.0", "0.0", "0.0"]},
 { "setambient",  [ "1.0", "1.0", "1.0", "1.0"]},
 { "setalphaenable", [ "0" ]}
]),
Tex = ObjectId ++ "/tex",
new( "ntexarraynode", Tex ),
cmd( Tex, "settexture", ["0", TextureFile, "none"] ).
```

Figure 13 shows how the planets are organised in the 3d space.

In the following example the object tree is organised as follow:

```
/usr/scene
/usr/scene/sun
/usr/scene/sun/planet1
/usr/scene/sun/extrarot1
```

```
/usr/scene/sun/extrarot1/planet2
/usr/scene/sun/extrarot1/planet2/moon1
/usr/scene/sun/extrarot1/planet2/extrarot3
/usr/scene/sun/extrarot1/planet2/extrarot3/moon2
/usr/scene/sun/extrarot1/extrarot2
/usr/scene/sun/extrarot1/extrarot2/planet3
```

Rotations applied locally also applies to child objects. In our example, the rotation applied to the sun refers to all objects. Interpolators' rotations does not applies to grand-father objects.

Note that rotations are not all in the main 3d planes. Rotation axis are transformed through extra rotations.

2.5.3 Grasping the power of interpolators

To connect an interpolator to a parent node, you need to specify a function, that will serve as a connector. You can use interpolation on size (`sxyz`), position (`txyz`), orientation (`rzyx`) but also lights or object color, texture position, and so on.

In the following example we build a blinking lamp by connecting an interpolator to its `setcolor` function. The object passed as parameter is a `nlightnode` type object. A 4D interpolator has been used, because the setcolor Nebula function takes four parameters:

```
ipol( ObjectLightnode, IpolType  ) ->
  Ip = ObjectLightnode ++ "/setcolor",
  new( "nipol", Ip),
  mcmd( Ip,
  [
   { "connect", [ "setcolor" ] },
   { "setipoltype", [ IpolType ] },
   { "addkey4f", [ "0.0", "1.0", "1.0", "1.0", "1.0" ]},
   { "addkey4f", [ "1.0", "0.0", "0.0", "0.0", "0.0" ]},
   { "addkey4f", [ "2.0", "1.0", "1.0", "1.0", "1.0" ]}
]).
```

Figure 14 shows a screenshot of the smoothly blinking light example. You could get another effect by using a different type of interpolator, such as `step`. In this case, blink would not be smooth at all.

2.5.4 Processing input events

You can assign functions to user input events. This allows you to interact with the 3 dimensional world.

Here is an example of the ndefault input setup:

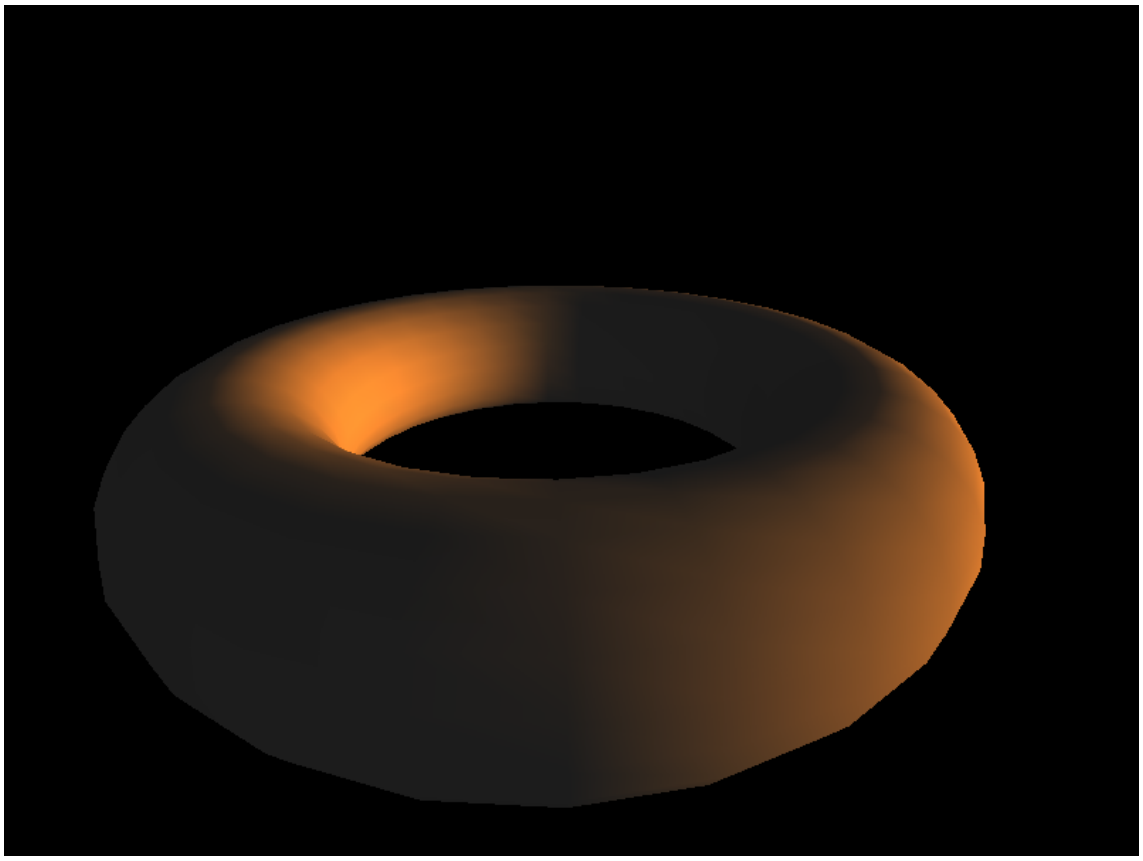

Figure 14: Smoothly blinking light

```
input() ->
  mcmd( "/sys/servers/input",
  [
   { "beginmap", [] },
   { "map", [ "keyb0:shift.pressed", "pan" ] },
   { "map", [ "keyb0:ctrl.pressed", "orbit" ] },
   { "map", [ "mouse0:btn0.pressed", "pan" ] },
   { "map", [ "mouse0:btn1.pressed", "orbit" ] },
   { "map", [ "mouse0:btn2.pressed", "dolly" ] },
   { "map", [ "keyb0:space.down" , "script:ndefault:orig()." ] },
   { "map", [ "keyb0:j.down" , "script:ndefault:janaview()." ] },
   { "map", [ "keyb0:b.down" , "script:ndefault:berndview()." ] },
   { "map", [ "keyb0:e.down" , "script:ndefault:eriview()." ] },
   { "map", [ "keyb0:k.down" , "script:ndefault:katiview()." ] },
   { "map", [ "keyb0:f.down" , "script:ndefault:fullscreen()." ] },
   { "map", [ "keyb0:w.down" , "script:ndefault:window()." ] },
   { "map", [ "keyb0:q.down" , "script:ndefault:quit()." ] },
   { "map", [ "keyb0:esc.down" , "script:/sys/servers/console.toggle" ] },
   { "map", [ "keyb0:f2.down" , "script:/sys/servers/console.watch gfx*" ] }
   { "map", [ "keyb0:f2.up" , "script:/sys/servers/console.unwatch" ] },
   { "map", [ "keyb0:f10.down", "script:ndefault:take_screenshot('screenshot
." ] },
   { "endmap", [] }
  ]),
  ok.
```

Note that when you do the input events map several time in your code, previous settings are not kept.

2.6 Other features overview

2.6.1 Lights

Lights are Nebula objects primitive. There are four types of light in Nebula:

- ambient: There should exist only one per scene, since the ambient lightsource can't be combined (they overwrite each other). To an ambient lightsource only "setcolor" is a usefull operation.
- point: Evaluates the actual position. "setcolor" sets color and intensity, "setattenuation" the distance related fade of light.
- spot: Evaluates the actual position and orientation (the ray of light goes along the negative z-axis of the parent n3dnode). setcolor, setattenuation and setspot work like expected.

- directional: The orientation of the parent n3dnode defines the direction (along the negative z-axis) of the light.setcolor sets color and intensity. Position of the n3dnode, setattenuation, setspot have no meaning in this context.

The following code is taken from the ndefault module and present two point lights (orange and blue) and defines the ambient lighting:

```
%% ==== Scene lights ====
%% Setup 2 point-lights and ambient
light() ->
  DLight = "/usr/scene/dlight",
  new( "n3dnode", DLight ),
  pointlight1(DLight),
  pointlight2(DLight),
  ambientlight(DLight).
%% Create the first point light
pointlight1(DLight) ->
  Light1 = DLight ++ "/light1",
  new( "n3dnode", Light1 ),
  L1Light = Light1 ++ "/light",
  new( "nlightnode", L1Light),
  mcmd( L1Light,
  [
   { "setattenuation", [ "1", "0", "0" ] },
   { "settype", [ "point" ] },
   { "setcolor", [ "0.1", "0.5", "1.0", "1.0" ] }] ),
  cmd( Light1, "txyz", [ "-50", "5", "50"] ).
%% Create the second point light
pointlight2(DLight) ->
  Light2 = DLight ++ "/light2",
  new( "n3dnode", Light2 ),
  L2Light = Light2 ++ "/light",
  new( "nlightnode", L2Light ),
  mcmd( L2Light,
  [
   { "setattenuation", [ "1", "0", "0" ] },
   { "settype", [ "point" ] },
   { "setcolor", [ "1.0", "0.5", "0.1", "1.0" ] }] ),
  cmd( Light2, "txyz", [ "50", "5", "50"] ).
%% Setup ambient light
ambientlight(DLight) ->
  Ambient = DLight ++ "/amb",
  new( "nlightnode", Ambient ),
  cmd( Ambient, "setcolor", [ "0.2", "0.2", "0.2", "0.0"] ).
```

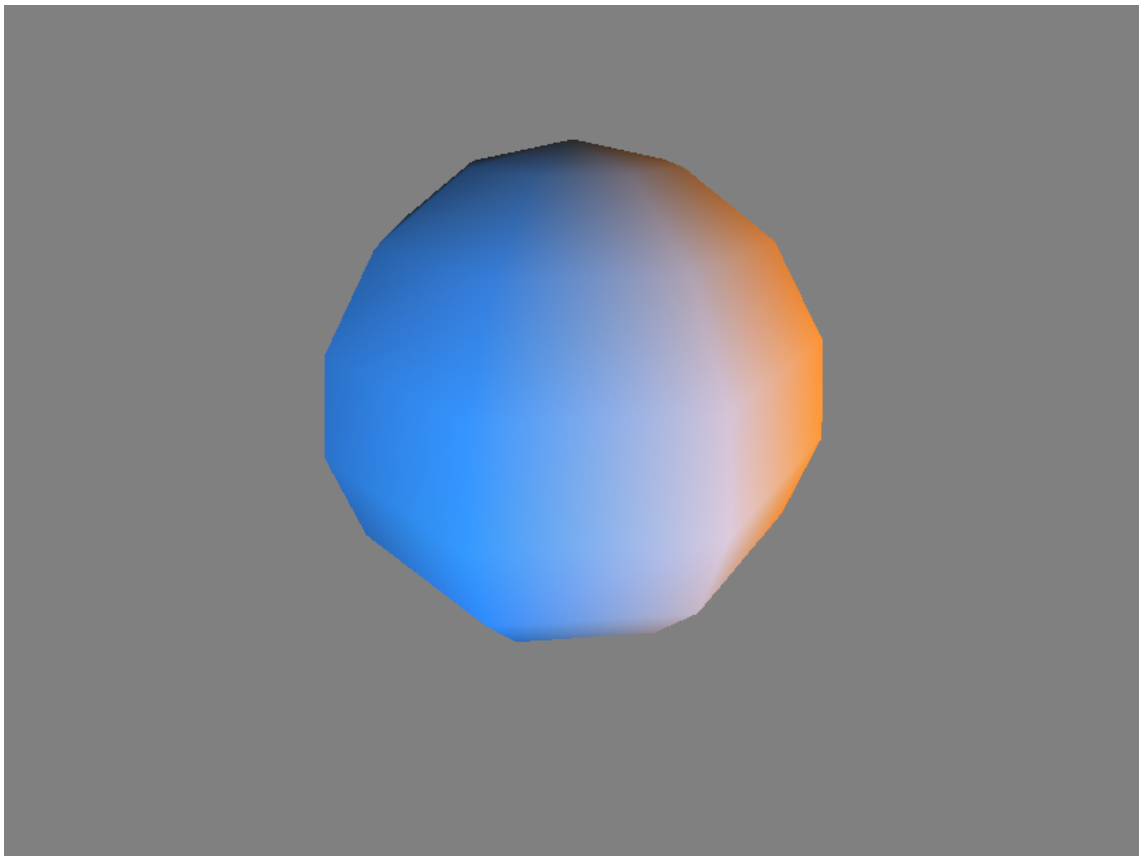

Figure 15: Default light settings

Note that the background color is setup on the graphic server object and not the light objects:

```
nebula:cmd( "/sys/servers/gfx", "setclearcolor", [ "0", "0", "0", "0" ] ).
```

2.6.2 Camera

Camera do not exist as such in the base Nebula framework. You are supposed to go at lower level and define window rendering parameters.

For the moment, we are working on, but still missing, a game framework that allows interaction with the objects on a higher level. It should not replace the access to the low-level functions, but it should ease the realisation of difficult scene in the game. In the meantime, low-level objects manipulation are available. For example, thanks to some trigonometric calculation you can define a camera movement. For the sake of simplicity, this example does only illustrate a camera movement on a plane. Upward or downward movement are not handled in this simple example.

A process is managing the camera status (position, heading, ...). Input events map to messages sent to this process. The process then takes care of readjusting the camera parameters. Here is the code that handle camera movements:

```
camera() ->
  Pid = spawn_link(?MODULE, camera_loop, [
   "/usr/camera",
   {50, 50, 150},
   0.5,
   0.0 ]),
  register(camera, Pid).
camera_loop( ObjectName, CurrentPosition, Speed, Heading ) ->
  Position = update_camera( ObjectName, CurrentPosition, Speed, Heading),
  receive
    {change_heading, Delta} ->
      NewHeading = Heading + Delta,
      camera_loop( ObjectName, Position, Speed, NewHeading )
  after 30 ->
    camera_loop( ObjectName, Position, Speed, Heading )
  end.
change_heading(Delta) ->
  camera ! {change_heading, Delta}.
```

The function to update the camera heading and position (according to its speed) does a bit of math:

```
%% Camera is not moving:
update_camera( ObjectName, Position, 0.0, Heading ) ->
  Position ;
update_camera( ObjectName, { Tx, Ty, Tz}, Speed, Heading ) ->
  % Do some maths first...
  Dist = 10,
  FixedHeading = (Heading+90) * math:pi() / 180,
  MX = math:cos( FixedHeading ),
  MZ = - math:sin( FixedHeading ),
  NewTx   = Tx + abs(Speed) * MX,
  NewTy   = Ty,
  NewTz   = Tz + abs(Speed) * MZ,
  %% The lookat rotation defines the camera orientation. The center
  %% of the rotation is defined by the translation.
  cmd( "/usr/lookat", "txyz", vector({NewTx, NewTy, NewTz}) ),  %% The rotat
 defined from the camera position
  cmd( "/usr/lookat", "rxyz", vector({0, Heading, 0}) ),  %% Camera heading
 a rotation on the Y-axis
  % Now update the nebula "states"
    mcmd( ObjectName,
        [
         { "txyz", vector({NewTx, NewTy, NewTz} ) }
        ]),
  % That's all
  { NewTx, NewTy, NewTz }.
```

2.6.3 Others effects

The Nebula engine supports many others effects that are not shown during this conference. For example:

- *Fog*
- *Particles effects*. They are used mainly used in pyrotechnic-like effects (explosion, smoke, disappearing trails in monowherl, ...)
- *Meshes interpolation*: Transformation from one object mesh to another.
- *Lens flares*: This is mainly a trick to emulate the behaviour reflection of lenses camera when pointing towards a light source.
- *3D spacial sounds*. It supports several interesting feature (i.e. doppler effect),
- *Skies*: We have worked on an example called `Monowherl`, which is an adaptation of the original monowheel Nebula example. This example include a sky environment. This is mainly a rendering trick as the scene and viewer are placed in between six textured planes.

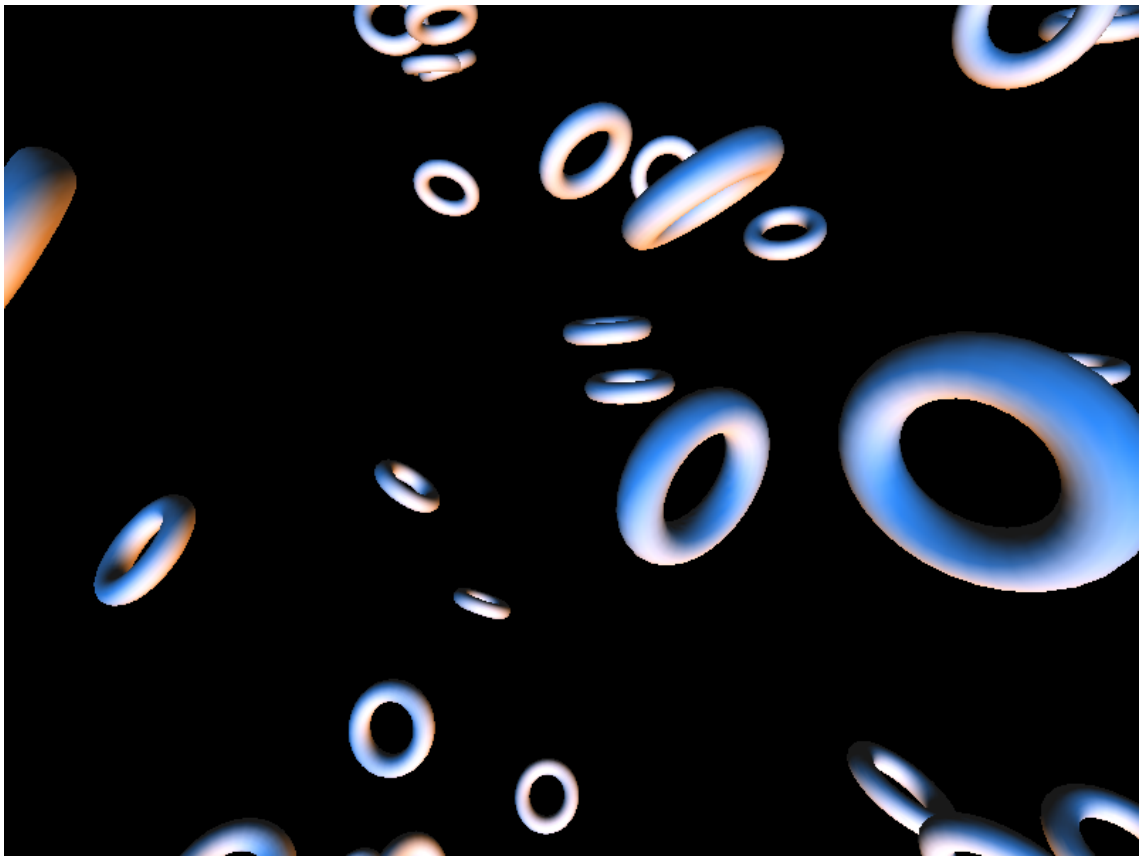

Figure 16: Camera movements: Lost in the Torus Space

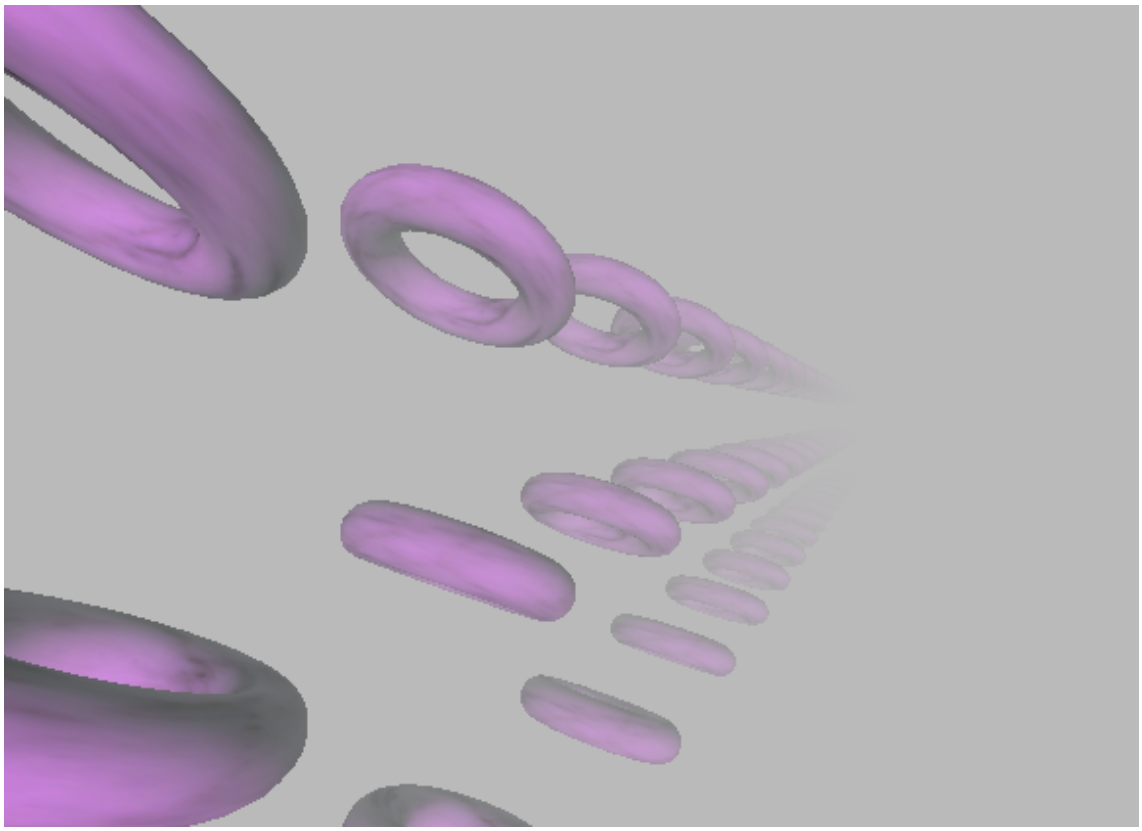

Figure 17: Fog in Nebula engine

- ...

3 Multiple Erlang processes

You can use multiple Erlang process to control the Nebula-Device engine. The `pmultitor` module show a case where every torus in the scene is controlled by a different process.

```
-module(pmultitor).
-export([start/0, start/1]).
-export([loop/1]).
-import( nebula, [new/2, cmd/3, mcmd/2] ).
start() ->
  start(10).
start( TorusNumber ) ->
```

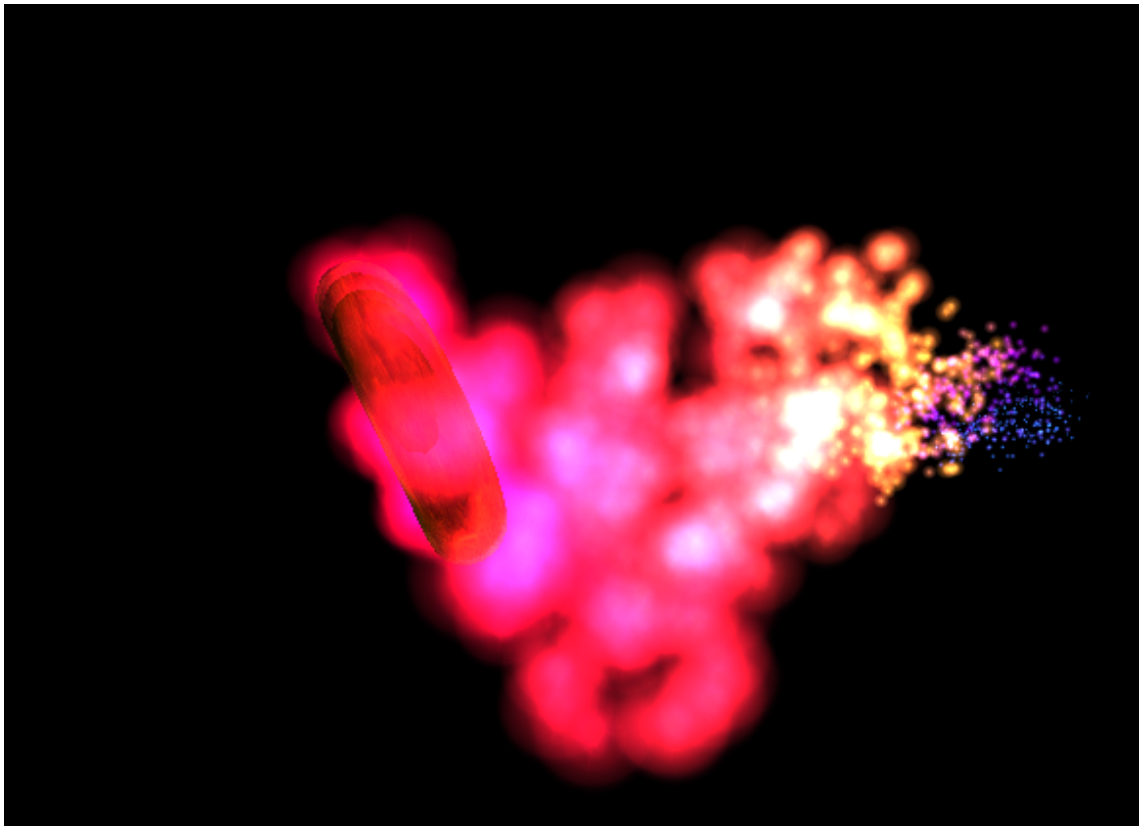

Figure 18: Particules

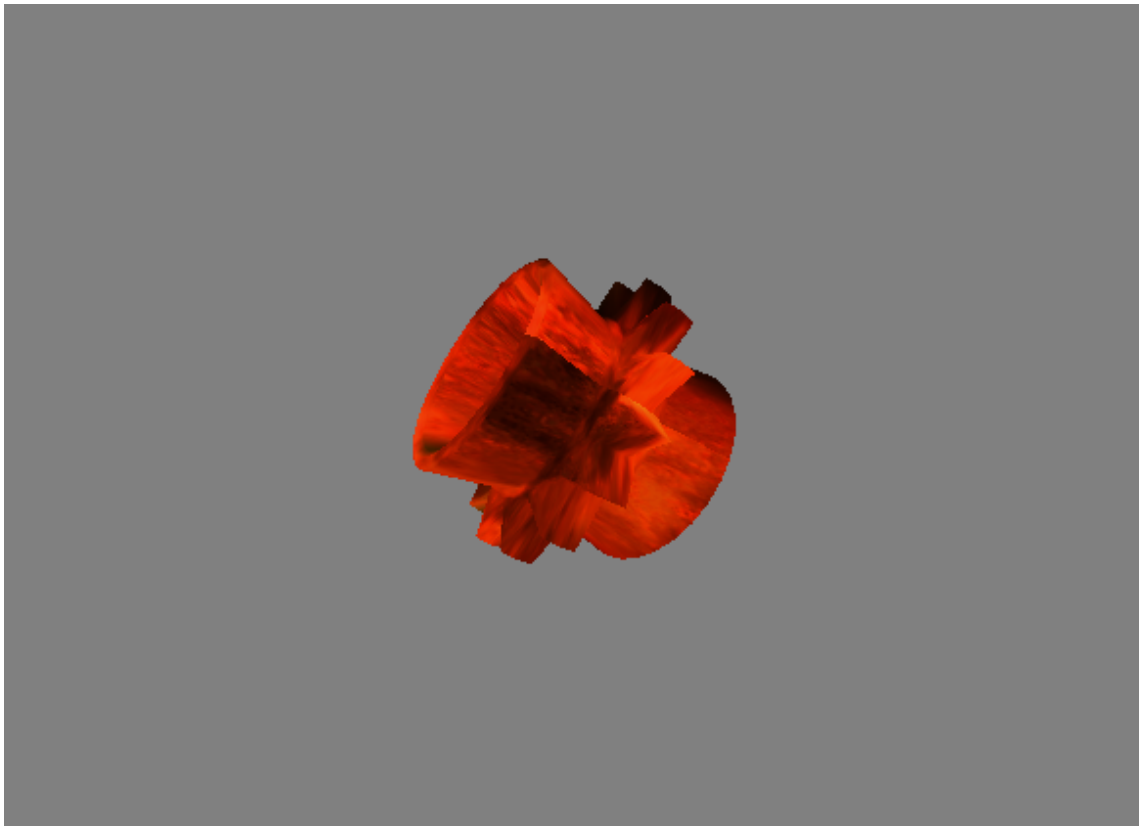

Figure 19: Mesh transformation by interpolation

```
  %% Setup a default "world"
  scene_setup(),
  Object = "/usr/scene/object",
  create_torus_group( Object, TorusNumber ),
  ndefault:eventloop().
%% Setup default environment
scene_setup() ->
  nebula:start(),
  ndefault:start(),
  cmd( "/usr/camera", "txyz", [ "50", "50", "150" ] ),
  cmd( "/usr/lookat", "txyz", [ "50", "50", "0" ] ),
  cmd( "/usr/lookat", "rxyz", [ "0", "0", "0" ] ),
  ok.
create_torus_group(BaseName) ->
  create_torus_group(BaseName, 50).
create_torus_group(_BaseName, 0) ->
  ok;
create_torus_group(BaseName, N) ->
  Number = lists:flatten(io_lib:format("~3.3.0w", [N])),
  Pos = random_pos(100),
  Orient = random_orientation(),
  create_torus(BaseName ++ Number, Pos, Orient),
  create_torus_group(BaseName, N - 1).
random_pos(Range) ->
  Tx = 1.0 * random:uniform(Range),
  Ty = 1.0 * random:uniform(Range),
  Tz = 1.0 * random:uniform(Range),
  { Tx, Ty, Tz }.
random_orientation() ->
  Rx = random:uniform(360),
  Ry = random:uniform(360),
  Rz = random:uniform(360),
  { Rx, Ry, Rz }.
create_torus(ObjectName) ->
  create_torus(ObjectName, {0.0, 0.0, 0.0}, {0.0, 0.0, 0.0}).
create_torus(ObjectName, TVector, RVector ) ->
  new("n3dnode", ObjectName),
  Mesh = ObjectName ++ "/mesh",
  new("nmeshnode", Mesh),
  cmd(Mesh, "setfilename", [ "torus.n3d" ]),
  update_position( ObjectName, TVector ),
  update_orientation( ObjectName, RVector ),
  ndefault:shader( ObjectName ),
  ripol_process( ObjectName, RVector ).
ripol_process( ObjectName, Orientation ) ->
  Params = [{objectname, ObjectName}, {rxyz, Orientation}],
```

```
  spawn(?MODULE, loop, [Params] ).
loop(State) ->
  Delay = 1000 div 50,
  timer:sleep(Delay),
  NewState = update(State),
  ?MODULE:loop(NewState).
update(State) ->
  {value, {objectname, ObjectName}} = lists:keysearch(objectname, 1, State),
  {value, {rxyz, Orientation}} = lists:keysearch(rxyz, 1, State),
  update_orientation(ObjectName, Orientation),
  %% For the moment, we don't change the state...
  State.
update_position( ObjectName, { Tx, Ty, Tz } ) ->
  cmd( ObjectName, "txyz", [ float_to_list( Tx ),
    float_to_list( Ty ),
    float_to_list( Tz ) ] ).
update_orientation( ObjectName, {Rx, Ry, Rz} ) ->
  Now = now_as_milliseconds() / 1000,
  NewRx = round(Rx + (360/10) * Now) rem 360,
  NewRy = round(Ry + (360/10) * Now) rem 360,
  NewRz = round(Rz + (360/10) * Now) rem 360,
  cmd( ObjectName, "rxyz", [ integer_to_list(NewRx),
    integer_to_list(NewRy),
    integer_to_list(NewRz)] ).
now_as_milliseconds() ->
  {MegaSecs, Seconds, MicroSecs} = erlang:now(),
    (MegaSecs*1.0e6 + Seconds + MicroSecs*1.0e-6) * 1000.
```

4 Monowherl example

`Monowherl` is a more complete example of what can be done in Erlang with the Nebula-Device engine.

5 Conclusion

5.1 Remaining tasks

If there are volunteers to help the project getting stronger, here are some crucial points on our todo list:

- Port the sound server to Linux, probably based on the OpenAL framework (`http://www.openal.org/`).

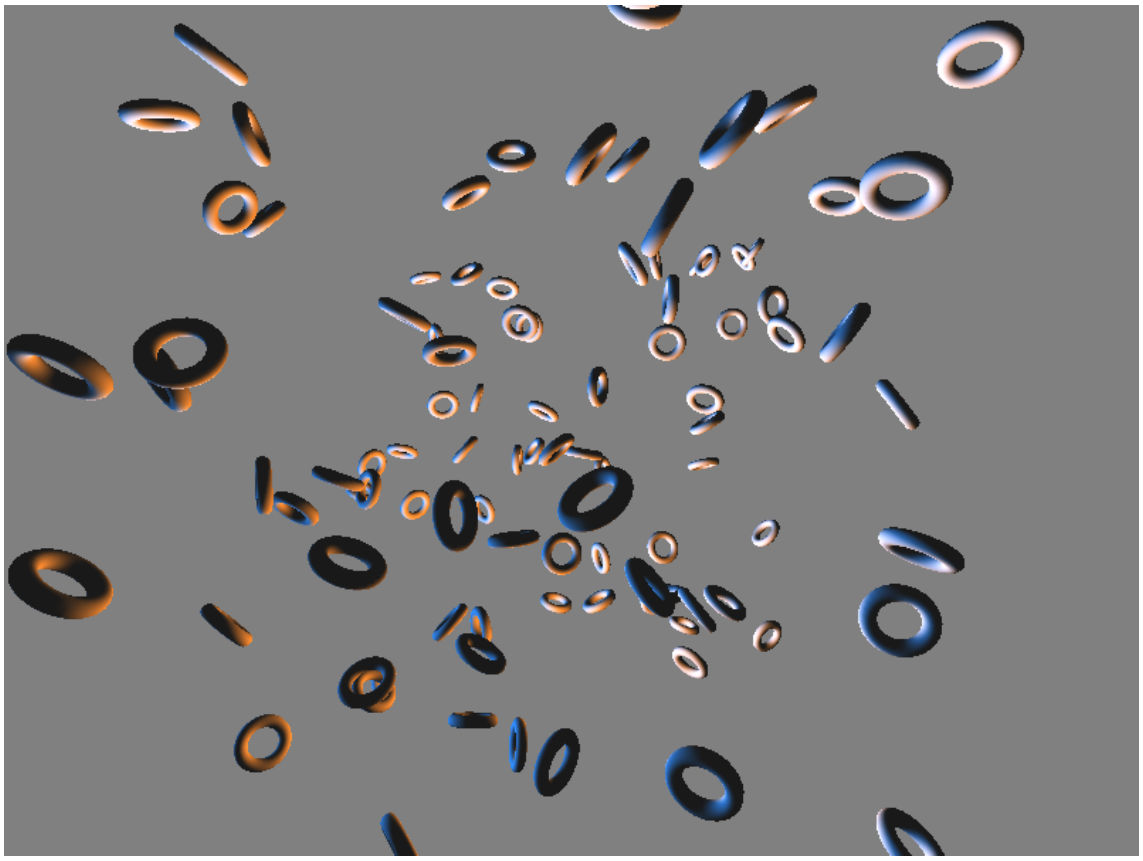

Figure 20: 100 Erlang processes controlling 100 torus objects in Nebula

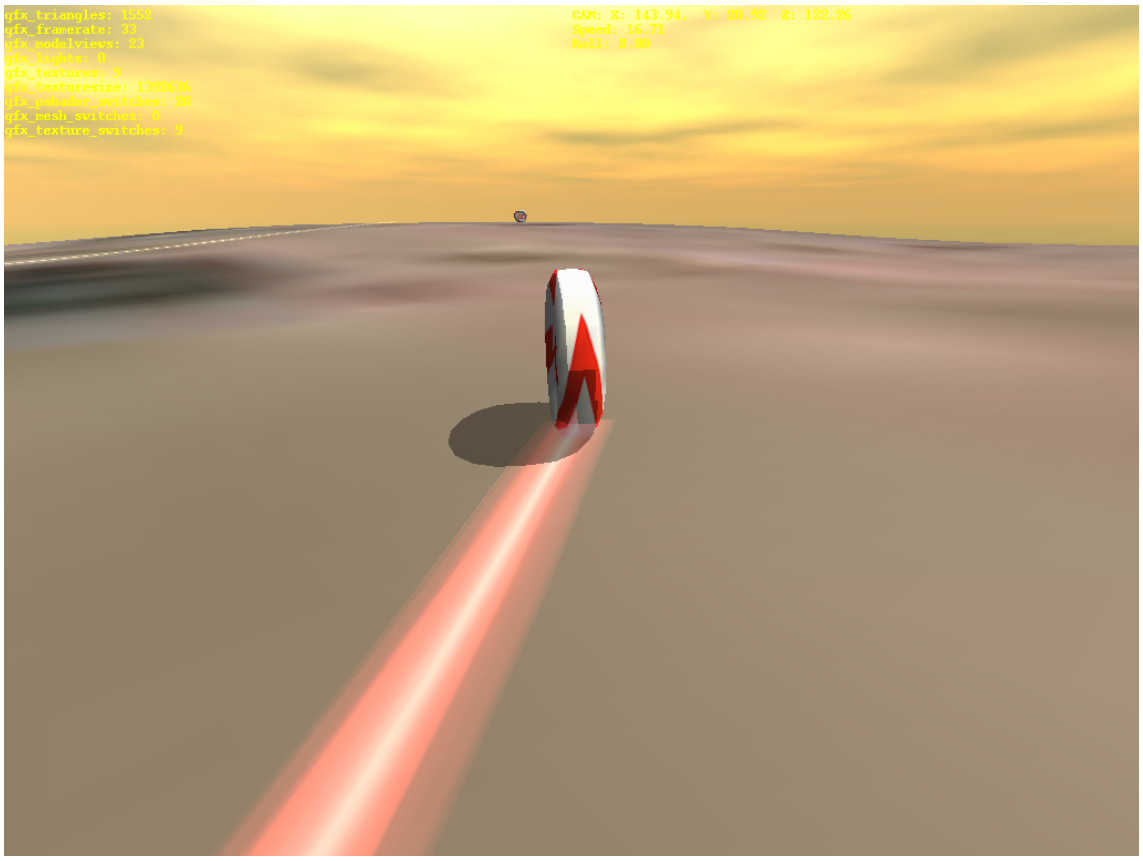

Figure 21: Monowheel in Erlang

- Clean up the binding code,
- Rewrite the Nebula Gaming framework,
- Writing the Wings3D shader exporter.

How long will it take before someone write a 3d Erlang process explorer with message passing representation ?

5.2 Why Erlang ?

Erlang does not seem a particularly skilled language for 3D development. However, today game development challenge is no more in 3D engine. Those tools are really mastered these days. The greatest challenges can be found in networking, fault-tolerance servers, hot-code upgrade to limit maintenance downtime. This is where the gaming industry is going, and we strongly believe that Erlang should plays a big role helping facing those challenges.

The Nebula Device code will be available after the conference from the Erlang-projects web site[7].

6 References

- Erlang Programmation, Mickaël Rémond, Editions Eyrolles, Collection Coming Next, 2003
- Nebula-Device: `http://www.nebula-device.org/`
- Wings3D: `http://www.wings3d.com/`
- Wings3D binary RPMs for Mandrake 9.2: `http://rpm.nyvalls.se/graphics9.2.html`

6.1 Erlang Projects Association

6.1.1 Goals

The Rei project is hosted by the Erlang Projects Association, whose mission is to provide more visibility to Erlang development that are made all around the world.

The association is promoting the programming language Erlang through several kinds of actions and goals:

[7] `http://www.erlang-projects.org/`

- Federate and mobilise the Erlang developers into a community through the Erlang-projects.org web site.
- Participate to existing project, help getting more action and more feedback or initiate development projects in Erlang.
- Offer an hosting infrastructure for Erlang based applications.
- Help companies using the Erlang language to promote their knowledge and achievements.
- Participate to computing related events to help spreading Erlang knowledge.
- Help companies wanting to evaluate Erlang as an implementation technology basis for their projects.
- Help Erlang developpers keep in touch and serve as a facilitor for informations exchange on projects and achievements with Erlang.

For more informations, please refer to `http://www.erlang-projects.org/`.

6.1.2 Infrastructure

The site is soon going to run under an Erlang infrastructure, based on a collaborative wiki approach. Please do not hesite to send feedback regarding your expectation towards the Erlang-projects.org web site.

List of Figures