Integration of a 3D rendering engine with a physics simulator

Author: Alberto Montañola Lacort

Directors: Carlos Ansótegui Gil
Juan Manuel Gimeno Illa

June 2009
Index

1. Introduction.................................................................................................................. 9
   1.1. Project Description........................................................................................... 9
   1.2. Project Goals.................................................................................................... 10
   1.3. Document structure......................................................................................... 10
2. Definitions, concepts and technology................................................................. 13
   2.1. Definitions......................................................................................................... 13
       2.1.1. The scene.................................................................................................. 13
       2.1.2. Scene Objects....................................................................................... 14
   2.2. Technologies...................................................................................................... 18
       2.2.1. The rendering engine.............................................................................. 18
           2.2.1.1. The rendering process................................................................. 19
           2.2.1.2. Low level programing interfaces.............................................. 20
           2.2.1.3. High level programing interfaces......................................... 21
       2.2.2. The input system...................................................................................... 21
       2.2.3. The Graphical User Interface................................................................. 22
       2.2.4. The physics simulation subsystem......................................................... 23
       2.2.5. The Audio subsystem............................................................................. 23
       2.2.6. The Networking subsystem.................................................................... 23
       2.2.7. Application Logic................................................................................... 24
       2.2.8. Linking the services.............................................................................. 24
3. Requirements analysis............................................................................................ 29
   3.1. Basic idea of possible client applications..................................................... 29
       3.1.1. Educational.............................................................................................. 29
       3.1.2. A game.................................................................................................... 30
       3.1.3. Scientific applications........................................................................... 32
   3.2. Developing a Framework, the Hollywood principle.................................... 32
   3.3. The clients of our framework........................................................................ 33
   3.4. Use Cases - Putting it all together.................................................................. 34
### 3.4.1. Developer tasks
- **Build** .............................................................................. 35
- **Publish and distribute** ................................................... 36
- **Maintain code** .................................................................. 36

### 3.4.2. Content creator tasks .......................................................... 37
- **Material creation and edition** ........................................ 37
- **Testing** ............................................................................... 37
- **Material management** ...................................................... 37

### 3.4.3. Application ........................................................................... 38
- **Configure** ............................................................................ 38
- **Provide basic services** ...................................................... 38
- **Register module** .............................................................. 39

### 4. State of the art ........................................................................ 41

#### 4.1. Complete engines ................................................................. 42
- **Crystal Space** .................................................................... 42
- **Unreal Engine** ................................................................. 43
- **id Tech** ............................................................................. 44
- **Shark 3d** ........................................................................... 45

#### 4.2. Graphics ............................................................................. 46
- **Ogre3d** ............................................................................. 46
- **Irrlicht** .............................................................................. 47

#### 4.3. Physics ................................................................................ 49
- **ODE** .................................................................................. 49
- **PhysX** ............................................................................... 50
- **Bullet** ................................................................................. 51

#### 4.4. Sound ................................................................................ 52
- **OpenAL** ............................................................................. 52

#### 4.5. Summary of technologies .................................................... 53
- **Complete engines** .......................................................... 53
- **Graphics engines** ............................................................ 53
15. Appendix F - Example applications ........................................ 117
   15.1. Basic GUI Example ..................................................... 117
   15.2. Basic scene manipulation example ............................... 120
16. Bibliography and references ............................................. 123
17. License .............................................................................. 125
Figure Index

Figure 1: A scene, with some scene objects: A light, a camera and some boxes over a plane.................................................................13
Figure 2: Types of lights.................................................................................................................................15
Figure 3: Mesh with an skeleton.....................................................................................................................16
Figure 4: Textured gingerbread man.................................................................17
Figure 5: Mapping hardware events into logical actions.................................22
Figure 6: Example chat window.........................................................................................22
Figure 7: Block diagram of main modules...............................................................25
Figure 8: Ghemical, a Molecular Modeller.................................................................30
Figure 9: PlaneShift, a MMRPOG........................................................................31
Figure 10: Architecture.............................................................................................59
Figure 11: Python architectures.................................................................................61
Figure 12: SWIG wrapper..........................................................................................63
Figure 13: Py++ Framework.......................................................................................64
Figure 14: BootStrap diagram of classes.................................................................77
Figure 15: Engine Diagram of Classes.....................................................................78
Figure 16: World Manager involved classes..........................................................83
Figure 17: Hook Manager involved classes............................................................86
Figure 18: Example simulation..................................................................................89
Figure 19: Development environment......................................................................115
1. Introduction

1.1. Project Description

Every day real time 3d applications are being used in different ways at our society. These applications allow building up virtual 3d representations of any type of real object that can be seen, or concepts or ideas of objects that are not real or that are not easy to see.

Over the latest years, there has been a technology race that has created competing products innovating several aspects of the technologies used by these applications. On the one hand, in the hardware side there has been an interesting evolution of the GPU\(^1\), on the other hand, in the software side there has been a continuous improvement in the algorithms, compilers, languages, specifications, implementations,... One of the goals of these applications is to improve the graphic quality to generate photorealistic graphics in real time. Other goals include the improvement of the sound quality, the usage of a physics simulator to improve the overall realism, and the usage of innovative new ways to interact with the scene, etc...

Unfortunately not all these technologies are publicly available. Most of these technologies are part of restrictive applications whose usage is restricted to a set of companies or clients with the available funding to pay the license fees.

Over the latest years, several open source projects have pop up into the market, providing code and material under a FLOSS\(^2\) license. These projects are going to be studied and compared, on the next chapter, with some of the most representative privative solutions available. This study is going to be a fundamental part of the research project, in addition of how a Start-Up with low funding resources could use these technologies to create good quality products.

Starting with a set of requirements, some of these FLOSS projects are going to be integrated by assembling a framework with all required tools and functionali-

---

1 GPU: Graphics processing unit. Current end user PCs, in addition the the CPU have another processor for graphics computing.
2 FLOSS: Free Libre Open Source Software
ties. This will aid both the developer and the artist in the process of the development of any type of 3d real-time application.

### 1.2. Project Goals

The main goal is to develop a framework that integrates a render engine with a physics simulator. It must satisfy the next main goals:

- Licensed under a Free Libre Open Source license.
- Speed up the overall development process.
- Provide independent and different services (graphics, physics, etc...) in a modular way.
- Should be easy to change the inner subsystems (swap the render engine with another one, use another physics simulator, etc.)
- Realistic behavior of the virtual world by using a physics simulator.
- User interaction with a keyboard, mouse and other devices such a joystick or a gamepad.
- Be easy to create specific tools/plugins in order to export material from a modeler such as Blender into our engine.
- Should be easy to expand the functionalities of the framework.
- Documentation of the API.

### 1.3. Document structure

An introduction about the key concepts, that are going to be seen in this research project, are going to be seen in chapter two. This includes the definition of what is an scene and all associated elements, finally the different modules will be introduced.

Next, in chapter three the requirements are going to be analyzed from different points of view (different types of users).
Integration of a 3D rendering engine with a physics simulator

Besides, a brief market research about the available technologies will be seen in chapter four.

Furthermore, the different tools and applications needed by any FOSS project with a community behind are going to be seen in chapter five.

The justification of the usage of the different selected technologies will be seen in chapter six.

The design and implementation of the project will be seen in chapter seven.

Finally the conclusion of the project is presented in chapter eight, and appendixes A and B explain the necessary steps to download and build the sources and testing the framework. Appendix C explains how a Java library could be integrated into the engine, by showing the integration of the CoDiP2P platform. The document type definitions used for the configuration files of the engine are detailed in appendix D. Finally the development environment used is detailed in the appendix E and the final appendix F explains two example applications that can be run within the framework.
2. Definitions, concepts and technology

In this chapter, the basic concepts and related technologies needed to understand this research project will be introduced.

2.1. Definitions

2.1.1. The scene

A scene is a group of determined objects with an identity and some properties. These objects can be lights, any type of geometrical form or mesh with color or textures, cameras, particle systems, etc. For example a scene can be seen as a room full of objects of different types.

Figure 1 shows a very simple scene composed of different types of scene objects that will be seen in detail in the following sections.

Figure 1: A scene, with some scene objects: A light, a camera and some boxes over a plane
2.1.2. Scene Objects

A scene object is an abstract representation or idea of any type of object contained in a scene. This object has an unique identity “name” and it may have a set of basic properties such as:

- **Position in the world**: A \((x,y,z)\) vector containing the absolute position in function of the coordinate system being used (the origin).

- **Information about transformations applied**: For example, if the object it is scaled or rotated. This information is usually stored in a matrix known as the transformation matrix.

- **A set of physic properties**: Information about the mass, the density, the elasticity, etc... of the object. This information is used by the physics simulator.

- **Sound information**: Objects may emit sound, so one can associate sounds to specific events such as a collision between two objects.

- **A set of user defined logic properties**: Client code (user made scripts) may need to read/write specific properties, that may modify the behavior of the object in the scene. For example a door can have a logic property that stores the state of the door (closed/open).

- **A link to a set of logic code associated to this object**: Client code that will run when this object is instantiated in the scene. For example, an automatic door is associated to an specific logic code; It checks the presence of the movement of another object in front of the door and then changes the state of the door (closed/open).

The object may be:

- **A light**: All scenes must have at least one light. Figure 2 shows the three main types of lighting available:
Integration of a 3D rendering engine with a physics simulator

- **Point Light**\(^3\): It gives off light equally in all directions, so it requires only a vector with the position. Direction information is not required.

- **Directional Light**\(^4\): It simulates parallel light beams from a distant source, there isn’t any position associated to it. Only needs a vector with the direction in which it is pointing.

- **Spot Light**\(^5\): This is the typical cone of light, it has a source of origin and the direction where it casts its light.

![Diagram of different types of lights](https://via.placeholder.com/150)

> **Figure 2: Types of lights**

- **A camera**: The camera (or cameras) define the different viewports where the scene will be seen. Normally only one free camera will be needed. It will be associated to an input controller such as a mouse or a keyboard. On some other scenarios more than one fixed camera may be needed, and the simulation may output the angle captured by them.

---

3 **Point Light**: In Ogre3d, the library that is being used, this light is known as LT_POINT, in Blender it’s named Lamp.

4 **Directional Light**: In Ogre3d is known as LT_DIRECTIONAL, in Blender it’s named Area or Sun, the exporter code may set some specific attributes depending of the type.

5 **Point Light**: In Ogre3d is known as LT_POINT, in Blender it’s named Spot.
● **Any visual representation**: Anything that can be seen in the scene, so any object with a mesh⁶, for example it can be a simple cube, or a human figure.

● **Other types of objects, or composite objects**: Miscellaneous objects that may be part of the scene. Sets of objects can be grouped into a group, meta-objects without visual information such as reference points are also possible. For example the scene may have an startup spawn point where the initial free camera will be located at the beginning.

Depending on the type of the object, it will have a set of extra properties associated to it.

Objects with a visual representation, known as meshes, may contain information about:

● **A set of specific deformations** that can be applied to the Mesh. Such as scaling and rotation of the object, between others...

● **Skeleton**. In order to animate any object, a set of interconnected bones, like the ones of any vertebrate, are defined. Each bone has it’s own position, orientation and scale, and it’s associated with an specific area of the mesh of the object (a group of vertices). For example in figure 3 the mesh

![Figure 3: Mesh with an skeleton](image)

---

⁶ **Mesh**: A mesh is composed of a set of vertices, edges and faces.
Integration of a 3D rendering engine with a physics simulator

of a gingerbread man shows a very simple skeleton, each bone is linked to a part of the mesh. So by moving the bone of one of its arms all the associated vertices will move. We can do a simple animation were our small person salutes to the viewer.

- **Textures**: Scene objects may have associated color information, so they can be displayed in one solid color. When it is possible to have several objects in different colors, it is also interesting to go a bit further. The surface, or skin that conforms any mesh, may have associated a texture. A texture can be something as simple as an image, it can be a photograph or anything else such as the output of another program, a video feed, etc. Figure 4 shows a gingerbread man with a likewise bread texture generated in the modeler.

- **Animations**: Scene objects can be animated in several ways. Firstly, it is possible to modify the position of the object, causing a movement effect; Secondly, any modifications done to the transformation matrix will alter the rotation and dimensions. Finally, changes performed over the skeleton linked to it, produce in this way the deformation of the mesh. When it is possible to manipulate the skeleton of the object manually by client code, the recommended easy approach is to record animations in the modeler and play them on demand. For example an articulated doll may have associated animations for the walk cycle, say hello, sit down,
etc. So the client code will only need to know which animation needs to be played, there is no need to worry about the complexity of the skeleton.

- **Forward Kinematics**: Desired transformation information is individually applied into all the bones determining the final end-point of each one. For example the arm of an animated model can be individually animated in this way, at the beginning the arm will be in a initial state, then transformation information is applied to each individual bone to reach the desired final state. The end-point is always unknown but the transformations that need to be applied to reach it from the initial position are known.

- **Inverse kinematics**: The desired transformations needed to be applied to each bone in order to reach the desired end-point are calculated by the IK solver. Both the current source point and the end-point are known, but the required transformations in order to change between both states are not known. Animations are usually done in this way by the modeler, since this is a difficult mathematical problem all those transformations are previously stored and the simulator will just play them as a simple forward kinematics animation.

### 2.2. Technologies

#### 2.2.1. The rendering engine

The main responsibility and goal of any rendering engine is to generate a representation of a scene (the input data). The final output data is an image (also known as frame) of what we would see in that moment of the simulation (at the location where the camera is aiming).

The rendering engine is build on top of a low level API that provides an abstraction over the underlying hardware. It maintains all information about the graphical resources, and it is responsible to manage this information in the most efficient way. It usually exposes all this information as a scene graph for the client code. Decisions of which objects need to be rendered/processed, etc... are part of the responsibility of such engine.
2.2.1.1. The rendering process.

The basis of the rendering process consists on the generation of an image from a set of data (the model or scene). In order to achieve this different rendering algorithms are available, and research continues on this area.

The most commonly used render algorithms are: rasterization, ray tracing, ray casting and radiosity:

- **Rasterization**: The process consists on the transformation of geometrical information stored in a vectorial\(^7\) form to a raster\(^8\) image. This process is the most fastest and suitable for real-time rendering. The algorithm performs a transformation of the 3D data, and it projects it as 2D information into a surface. Then this information is sent to the display device (monitor). This is the method currently used by most engines, and it is the one currently implemented on most graphical hardware video cards.

- **Ray tracing**: The final image is generated by tracing rays from the virtual eye (camera) to the first object were the ray intersects. Then the algorithm calculates the pixel color in function of the object's material properties, the illumination, etc. When in nature (the real physics world) the human eye receives the lights rays, in ray tracing the same process is done in backwards to save a lot of useless computation for unneeded light paths. Ray tracing algorithms have an higher computational cost than rasterization ones, so them are currently not usable for real-time rendering. In the other hand, the results have a very high degree of photorealism, more than the realism achieved by rasterization. Currently available ray-tracing hardware is still experimental and under research, real-time rendering it is currently achieved by using parallel machines. See [brl08].

- **Ray casting**: Ray casting is based on the ray tracing process, but it is a faster version of it. On ray tracing when the ray intersects the surface it computes the new rays: reflection, refraction and shadow. Ray casting does not compute these new rays, and uses other methods to fake this in-

---

7 **Vectorial**: Graphical information is stored in a set of basic geometrical primitives (points, lines, curves, etc...)
8 **Raster**: Graphical information is stored as a matrix of pixels.
formation (like storing it on the textures). This increases a lot the speed of the rendering process allowing its use for real-time rendering. Currently there is not any hardware implementation of this technique.

- **Radiosity**: This technique attempts to simulate the way in which some illuminated surfaces could act as indirect light sources. The algorithm is based in the theory of thermal radiation. It generates several patches (pieces of the scene), and determines how the illumination information of these patches affects the other ones. It iteratively performs this action until the computation numerically converges or the image already looks well enough.

### 2.2.1.2. Low level programing interfaces

Nowadays the most currently render technique used in real-time rendering applications is obviously rasterization. There are currently two main interfaces over the currently available video hardware: OpenGL and Direct3D.

- **OpenGL**: The Open Graphics Library is an standard open specification. it is free, multi-platform and supports multiple programing languages. Although the API specifications are open, there are non-free (proprietary) implementations of the library. Graphics cards vendors provide its own implementation for a determined set of supported operating systems. For example Nvidia, Intel and Ati provide their specific library implementations, though Nvidia and Ati are relaying on a proprietary kernel driver when only the intel one is free. Some efforts are done by some individual groups in order to provide a full free implementation for Ati and Nvidia video cards. There is also a software renderer implementation (Mesa 3D).

- **Direct3D**: It’s part of the Microsoft’s DirectX API. It’s closed, restricted, proprietary and it does not follow any standard and only works on a very limited set of operating systems. The winehq project is working on a free implementation of the DirectX libraries in the effort to run windows applications that rely on them under Linux and other platforms.
Integration of a 3D rendering engine with a physics simulator

Unfortunately in the industry the most used interface is Direct3D, although several higher level interfaces offer an abstraction over both interfaces.

Microsoft is famous of doing anything to gain absolute control over the market, thus OpenGL is not very well supported over recent versions of Windows.

2.2.1.3. High level programing interfaces

The high level programing interfaces provide an abstraction over one or more low level interfaces. These engines usually organize all the graphical information in a graph, and provide several common resources commonly used by client code. Some of this interfaces may provide a resource provider, to manage the graphical resources that are going to be loaded into the scene.

Some of these interfaces are object oriented, and the objects of the scene can be managed by the programmer as instances of a class. In addition, there are implementations only focused in the rendering services, and more complex ones implementing a richer API with more services like physics, sound, networking, etc...

These engines will be shown in more detail in chapter 4.

2.2.2. The input system

The input subsystem is responsible for monitoring all events generated by all attached devices (keyboard, mouse, joystick), and fires up notifications to all subscribed client listeners.

It offers an abstraction layer in front of all the available hardware and maps whatever hardware is available to top level actions.

For example, an action named “Hello World” is defined and associated to an input event, such as clicking a mouse button. Then an action listener is defined and registered, so it can process this event. When the user clicks a mouse button the Input System will fire up the event that will be processed by the action listener executing any associated code. Figure 5 shows some possible mappings.
2.2.3. The Graphical User Interface

Most graphical engines do not provide any way to build up user interfaces, and most times the client code needs to build up widgets and controls. When it is possible to embed the render window into an application, there are no means for rendering the controls used by the window manager. This framework includes a built-in set of widgets (text control, buttons, dialogs, etc...) in a way that the client code only needs to instantiate them. Figure 6 shows an example window done with the CEGUI library that will be explained in chapter 6.4.

This subsystem must provide a rich API with everything that any user needs to build up a GUI, handle all events associated with the GUI, and fire up client hooks that are attached to specific controls.
Integration of a 3D rendering engine with a physics simulator

2.2.4. The physics simulation subsystem

The physics simulator stores the state and physical properties of all dynamic objects of the scene. The simulator is initialized with the initial state of the scene, or a previously stored state. Every time that the render engine is going to draw a frame, it needs to update the state of all entities by querying its state to the simulator.

For an application running at 70 frames per second, the simulator cannot consume more than 1/70 seconds in computing the next scene state. If that happens the frame rate can be reduced, but if the simulator needs always more 1/15 seconds, then the physics problem cannot be solved in real-time.

In general, in order to achieve a real-time simulation, the time required to simulate one second of the dynamics system must be less or equal than one second, being one second the maximum border line. For example, if a simulator needs one hour to compute a second of the simulation, then it is impossible to show the simulation in real-time.

2.2.5. The Audio subsystem

The main job performed by the audio subsystem is to reproduce all sounds emitted by the scene objects. The input data consists on the geometrical positions of the sound emitters used in the simulation, the intensity and other applied alterations to these sounds. It needs to know the layout of the speakers (stereo, quadruphonic, 5.1, 7.1, etc...) so it can correctly compute the resulting sound mix for each channel depending of the position of the virtual listener.

2.2.6. The Networking subsystem

The tasks performed by the networking system are mainly the synchronization of the state of a single shared physical world between a set of clients.

Real time protocols built over UDP are preferably used instead of those over TCP due to the additional overhead and latency issues.

The main functionalities that most network cores will offer is the possibility to
send reliable and non-reliable messages, and set a priority to these messages.

For example, if an object in the simulation sends a message with its position information, and the message is lost, then it is not correct to retransmit the message. The moving object may have a new position in the world, so the system needs to send in a reliable way the most recent information about what knows about a determined object.

### 2.2.7. Application Logic

The goals of the application logic are to provide means to easily manipulate the scene objects and to react when an event happens. For example do something when a collision between two objects is detected.

Every dynamic object may have an associated logic script that defines the behavior of this object since it is created until it is destroyed.

For example, a ball is defined without any type of associated logic. This ball will stay in repose until an external object collides or applies a force to it. Another ball is defined with an association to a logic script that compares the state of the ball every second. If it detects that the ball remains in a static state for some seconds, then it applies an impulse to the base of the ball. Then it will start to jump, until all the energy is consumed and stays again in repose until the logic script detects again the situation and applies another impulse to it.

### 2.2.8. Linking the services.

Finally, it is necessary to join together all the seen modules in some way, in order to assemble the final application.

Figure 7 shows a block diagram of the most commonly used modules linked together.

The input module will process events generated by the hardware and will be sent to the specific logic script.

The application logic module should have mechanisms to notify the physics simulator about changes performed on the state of the object, in addition, it
Integration of a 3D rendering engine with a physics simulator should be able to request to the sound system the possibility to play a sound, and perform other object manipulations directly over the renderer.

The physics simulator, contains an instance of all the physical world data, this data is updated by the logic system, and then it is synchronized with the renderer. A collision event may fire up a sound between other possible events.

Updates received by the render engine, both by the logic and the physics modules, will update the information about these object visualization properties.
The audio module maintains the audio view of the world, it consists of all the available sound emitters with information of the sound, volume and effects that need to be applied. This information will be updated by the physics simulator and the application logic scripts.

Every module needs to be updated at least one time every frame, so the framework needs to define a main maintenance loop. If the application needs to run at a framerate of x frames per second, then the whole maintenance cycle should never consume less than 1/x seconds. X should be the monitor refresh rate.

On a single-threaded environment the maintenance loop will look like this:

```python
while True:
    renderer.renderOneFrame()
    input.processInputEvents()
    logic.runLogicScriptsCycle()
    physics.simulateOneStep()
    sound.update()
    sleep()
```

1. The renderer will be updated with the latest information and a new frame will be rendered.

2. Input data will be processed, this will shot up the execution of any logic hooks attached to an specific event.

3. The application logic maintenance loop will run. It needs to run a cycle of life of every instantiated object in the scene.

4. The physics simulator will be updated with the latest updates by the application logic, it will run a new simulation step, and finally it may shot up the execution of any logic hooks attached to physics events like a collision.

5. The sound system will be updated with the latest information about the world, and the maintenance cycle will be run.

6. Finally the application may sleep the remaining time, if available. So it will sleep 1/x minus the time consumed by the maintenance cycle. If the time consumed by the maintenance cycle is higher than 1/x, then it is not pos-
Integration of a 3D rendering engine with a physics simulator

sible to run the simulation in real-time.

Communications were not included in this example, a communications module will also contain its maintenance cycle and will receive and send updates about the state of the world. It is an interesting challenge the synchronization of the physics simulator state of all nodes working on the same simulation.

Furthermore, this example shows a single threaded loop. Nowadays end user machines available in the market have processors capable of running multiple hardware threads, because them have more than one core in the same processor. On traditional single core processors only one thread of a multi-threaded application will be run by the processor at a time. On the other side these multi-core processors are capable of running in parallel 'n' threads, where 'n' is the number of available cores. So it should be possible to build up a multi-threaded loop, that will use all available hardware resources. Each module implements an isolated service that can be put on its own thread, the main challenge would be the synchronization of the unique and shared state of the world between all these processes, between others.
3. Requirements analysis

In the design process of any application the needs of the clients are studied. So on normal end-user applications the study is focused by the requirements of the people using it. This project consists on a framework, so one of the points to analyze are the requirements of the client applications. Another aspect to consider are the needs by the developers that are going to actively work with the framework. To find the main needs, the following questions need to be answered: What type of applications are going to be built?, How are going to be built up these applications?, For which public is intended the library?

3.1. Basic idea of possible client applications

Answering to the question: What type of applications are going to be built?

The most simple application will consist on displaying any arbitrary scene. Without the need of writing any type of code the client will only need to set the name of the file containing the description of the scene, and the framework will do the rest. Over this base, more functionalities will be added ensuring that the most common, and required functionalities are already available in the library.

In the following sections the requirements of different types of applications are going to be briefly analyzed.

3.1.1. Educational

In order to help in the learning process students can be presented with some models and ideas about the concepts studied in class. For example, in chemistry it is interesting to see a virtual representation of several molecules (as seen in Figure 8), in biology a representation of the cell and how it works, in physics is possible to simulate several practical problems, in history a simulation of the look of a city/environment can be shown in an specified epoch, etc...

These type of applications are usually simple, and the goals are to make the usage and implementation of these applications the most usable and easy for the professors and students.
3.1.2. A game

This framework is also intended for developing games that will display its graphics and contents in 3d. From the huge amount of games available in the market, the most known types of games are going to be analyzed from the most simple to the most complex games.

- **Typical puzzle games**: These games consist of just a simple puzzle with some interaction from the user. For example: The towers of Hanoi, a 3d Sudoku, a Rubik cube, and several others. These games will require only the graphics renderer to display the status of the puzzle, and a way to interact with the mouse/keyboard to manipulate the puzzle. Optionally a sound effect can be produced when for example a piece fits in one space, etc... In addition the physics engine may be necessary for some puzzles.

- **Adventure games**: They consist of large scenarios were the player will interact with the environment. Adventure games usually have huge interior and exterior environments with a lot of details. The player will move, interact, and alter the environment in several ways. The renderer should be capable of working with a huge amount of information. Moreover, it is also important the capability to play music and sound effects.

*Figure 8: Ghemical, a Molecular Modeller*
Integration of a 3D rendering engine with a physics simulator

- **Real time strategy games**: RTS games consist of a very large map, that may have a lot of detail, over these maps we will place a lot of mobile elements. On RTS games the player will need to build and move a large amount of units over the map, in order to expand its territory or/and attack other enemies. In addition to the renderer and the sound system, some helper aids may be required, like a path finding solver. Players may want to play against another humans, so it is required to allow easy exchange of game messages/states between them. Playing against the computer it is also an interesting problem.

- **Role playing games**: They are like adventure games, though are a bit more complex. They allow to customize the character, interact with other players, commerce with them, fight with players or monsters, etc... These games have an interesting complexity, by needing commerce, fighting, etc... services that may run in some times on a remote server. In the case of Massively Multiplayer Role Playing Online Games, instead of a single history-line, such as those one can find on a typical adventure game, on

![PlaneShift](image)

*Figure 9: PlaneShift, a MMRPOG*
these games there are missions were the player needs to achieve a set of goals in order to succeed. Figure 9 shows an example of a MMRPOG.

- **Action games**: These games are though with the “destruction” word in mind. The environment is intended to be destroyed by the player, while he/she fights with other players or monsters. On these games, players usually move and perform any type of actions very fast. Thus latency over the exchanged game messages negatively affects these type of games.

### 3.1.3. Scientific applications

There are several possible applications that would be interesting to be implemented using our library. Architects may want to display simulations of how an specific building may look. Also it should be possible to display more complex simulations like the results of any specific problem that can be the simulation of how a bridge can support an specific weight, etc...

These applications are very specific, so the application will be usually feed with the data generated by another program.

### 3.2. Developing a Framework, the Hollywood principle

**Answering to the question: How are going to be built up these applications?**

In order to build up one of these applications, due to the complexity of the library and the goals that should be achieved, the most suitable way is to build up a Framework.

The first goal is to reduce the amount of time required by the client code to configure and/or initialize all required components. In addition, one of the main goals is to make the most simple as possible the process of creation of a scene, or the manipulation of it. So with only few lines of client code it should be possible to create a basic primitive, or loading a complex scene.

The Hollywood principle is a software design pattern, also known as the template method. On this pattern the library will call the client code when it is necessary, in a similar way that actors are called by Hollywood: “don't call us, we'll call...
Integration of a 3D rendering engine with a physics simulator

The framework should work similar to an operating system. That means that the framework will offer a set of services that will be started when are necessary (services on demand), and the developers working with the framework may expand the functionalities of it by implementing other services. The most simple application with only few lines of code, or not code at all should be able to load up an scene.

In order to build our applications specific modules will be created in order to solve specific problems. Furthermore, any application can be divided into several parts, or individual problems. The global application will be completed by finishing all modules. This is also know as “divide and conquer”.

3.3. The clients of our framework

Answering to the question: For which public is intended the library?

An analysis of the different type of actors, or people that are going to be involved in a direct or indirect way with the framework, is going to be introduced. There are three main profiles: Developer, Content creator (artist) and End User.

- **Developer**: A developer is understood as all the people involved in the process of designing and writing new client code for the framework. They will usually start with an idea, and them will want to implement it without the hassle of dealing with low level graphics interfaces, etc...

- **Content creator (Artist)**: An artist, or the content creators, are all the people involved in creating all the media resources required to develop the project. Some applications will not require media at all, but on other it may become the most important part of it. For example: on games, scriptwriters will be actively involved in the decisions done by the developers. Content creators will deal with all the tools and processes required to create content for the framework and testing it.

- **End User**: The end user is the final recipient of the product created by the previously seen users. The end user will indirectly interact with the frame-
work. The end user will be always a potential future content creator or developer of possible future applications.

3.4. **Use Cases - Putting it all together**

The different types of tasks (use cases) and people (actors) involved in these applications are going to be introduced in this section.

The following table contains a list of the main tasks/goals:

<table>
<thead>
<tr>
<th>Profile</th>
<th>Tasks/Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developer</td>
<td>● Build&lt;br&gt; ○ Download code and dependencies&lt;br&gt; ○ Compile&lt;br&gt; ● Publish and Distribute&lt;br&gt; ● Maintain code</td>
</tr>
<tr>
<td>Content creator</td>
<td>● Create and edit material (graphics, sound, etc...)&lt;br&gt; ○ Export the material&lt;br&gt; ● Test&lt;br&gt; ○ Load the exported material into the engine&lt;br&gt; ○ Test it&lt;br&gt; ● Material management&lt;br&gt; ○ Publish&lt;br&gt; ○ Edit&lt;br&gt; ○ Remove</td>
</tr>
<tr>
<td>Application</td>
<td>● Configure&lt;br&gt; ● Provide basic services&lt;br&gt; ○ Scene management&lt;br&gt; ■ Load scene&lt;br&gt; ■ Manipulate scene&lt;br&gt; ○ Process input (keyboard, mouse, gamepad)&lt;br&gt; ○ Generate output (video, sound, etc..)&lt;br&gt; ○ Process physics, AI, etc..&lt;br&gt; ● Register plugin or module</td>
</tr>
<tr>
<td>User</td>
<td>● Download &amp; Install &amp; Run&lt;br&gt; ● Participate in the community&lt;br&gt; ○ Report bug&lt;br&gt; ○ contribute (code, material or ideas)</td>
</tr>
</tbody>
</table>

On the other hand, these tasks may be provided by:

- **Bootstrap application**: It is responsible of setting up all the requirements to build up the application. In addition, it is also responsible of the
Integration of a 3D rendering engine with a physics simulator

application distribution over Internet.

- **Application maintenance tools**: They are a set of tools responsible of the maintenance of the application code.

- **Content tools**: The content tools are the complete set of applications used to create assets for the final application. They are part of the content creation pipeline.

- **Framework**: It provides the necessary services to build up applications.

### 3.4.1. Developer tasks

#### 3.4.1.1. Build

- **Description**: Projects that rely on several dependencies will always need as a prerequisite the correct library compiled. Nowadays each library has a different method for obtaining the source, configuring, building and installing it into the system. For example any library may have several versions that may need to be downloaded from different possible locations: subversion, cvs, a web server...

- **Goals**:
  
  - **Download**: It should obtain the source code and all its dependencies. Thus the build system should contemplate all possible ways to download it. These will be: Code repositories (CVS, Subversion, mercurial, git...), hosted in a web or ftp server, obtained on a P2P network...

  - **Configure**: Each library has its own configuring and build mechanism, these may be automake, ant, scons... The system should support these mechanisms and pass to them the correct build options.

  - **Build**: The system should resolve all dependencies for each module, and build them in order.

  - **Install**: Finally it should be possible to install it into the operating system, or the user's home directory.
• **Responsibility provided by**: Bootstrap application.

### 3.4.1.2. Publish and distribute

- **Description**: Applications will require packaging and publishing mechanisms.

- **Goals**:
  - **Packaging**: Create platform specific install packages. For example, a debian based linux distribution will use DEB as the packaging format.
  - **Publish**: Provide mechanisms to upload the application into a public server for distribution.

- **Responsibility provided by**: Bootstrap application

### 3.4.1.3. Maintain code

- **Description**: The sources can easily be obsolete, the hardware, operating systems, libraries, will evolve and change. The source code needs to be updated for any possible changes in the dependency tree. In addition, new features and improvements can be introduced, by the other way, it should be also possible to fix possible errors or bugs that may arise.

- **Goals**:
  - **Code repository maintenance**: Maintain the evolution of the code, by keeping track of different possible branches and creating tags for individual milestones or releases.
  - **Improvements, features and bugs**: Keep track of new features, improvements, bugs...

- **Responsibility provided by**: Application maintenance tools
3.4.2. Content creator tasks

3.4.2.1. Material creation and edition

- **Description**: Different types of material can be created by several tools (Graphics, sounds, 3d models, scenery, etc...). These resources may be generated or edited in different ways in several formats. Thus, all of them should be converted into a format compatible with the application framework.

- **Goals**: Provide a development pipeline that will directly export into the framework different types of resources.

- **Responsibility provided by**: Content tools

3.4.2.2. Testing

- **Description**: Material created with the different content creation tools need to be visualized in the final application, so the artist can have an idea of how it will look.

- **Goals**: Provide a mechanism in the creation pipeline for directly loading and testing the material into the application.

- **Responsibility provided by**: Content tools

3.4.2.3. Material management

- **Description**: All generated material assets needs to be packaged, managed, edited and published. They require more disk space than the framework itself, and will consume more bandwidth in the distribution over Internet.

- **Goals**: Provide a publishing mechanism for content authors. This mechanism should allow them to easily add, modify and remove material from a “content server”.

- **Responsibility provided by**: Content tools
3.4.3. Application

3.4.3.1. Configure

- **Description:** There are several aspects of the framework that should be possible to configure, such as paths to resource files, graphics settings, and several other options.

- **Goals:** The initialization of the framework should be configurable, by either passing a configuration file or allowing runtime configuration by the client.

- **Responsibility provided by:** Framework

3.4.3.2. Provide basic services

- **Description:** Different functionalities need to be provided as services to the client code. It should be possible to have several different implementations of the same services.

- **Goals:**
  
  - To offer one or more implementations of the same service under the same fixed interface.
  
  - To provide an scene management service that will offer some basic actions:
    
    - **Load scene:** Load a set of objects into the renderer service.
    
    - **Modify scene:** Individual objects should be edited, by changing its position, animation, properties, textures, etc...
    
    - **Unload scene objects:** Deletion of the scene, or an specific object.

  - To provide an input service responsible of processing events generated by hardware devices such as a Keyboard, a Mouse, a Gamepad, a Joystick,....
Integration of a 3D rendering engine with a physics simulator

○ To provide a physics service over the loaded scene objects, that should allow the possibility to affect the behavior by changing its properties, applying forces...

○ To provide a sound service responsible of registering sound callbacks over events that can occur between objects, or in the scene in general.

○ Output generation: Finally all manipulations performed over the scene, or world, should be presented graphically and related sounds should be reproduced.

● **Responsibility provided by:** Framework

3.4.3.3. *Register module*

● **Description:** The framework should be open to extension and implementation of new functionalities. Adding new modules should be possible without the need to re-implement, or change existing code.

● **Goals:** To provide a mechanism to register external modules into the framework.

● **Responsibility provided by:** Framework
4. State of the art

Researching the market allows us to have a clear view of the solutions and base technologies currently available.

This study will focus on five important criteria: the license, the maturity and stability, the community, the programming language and paradigm used, and the overall features.

- **License**: All available licenses can be classified in two main groups: free and non-free. Using always the meaning of “freedom” we consider a license to be free if the user has:
  - The freedom to run the program, for any purpose
  - The freedom to study how the program works, and adapt it to your needs
  - The freedom to redistribute copies so you can help your neighbor
  - The freedom to improve the program, and release your improvements to the public, so that whole community benefits.

Free licenses, like the GPL or the LGPL, are the ones that will be used in this project.

- **Maturity and stability**: Projects that have already been around for some years, and have been used in several solutions will have an important maturity value against younger projects. Furthermore this affects the overall stability of the project and the risk of being abandoned or not maintained anymore. Mature projects are generally more stable and maintained that any young or new project.

- **The community**: The size of the community behind any project reveals the impact and importance of it. The quantity and quality of the documentation is related to the size of the community, so projects with an average community will have a lot of available documentation, plugins, examples, etc. If the community size increases, the amount of chances that someone
will document, use and contribute to the project will also increase.

- **Programing language and paradigm**: Each solution will provide interfaces to a different set of languages and will use a different set of programming paradigms.

- **Overall features**: The most important overall features of each engine in order to compare which the other ones.

On the first hand, a set of the most known engines (with all modules) will be studied. In addition, individual graphics engines, physics libraries, sound systems, networking, and other ones like path finding, animation... are going to be briefly described.

### 4.1. Complete engines

#### 4.1.1. Crystal Space

Crystal Space [cry08] and [col09] is an open source cross-platform software development kit for real-time 3D graphics, with particular focus on games.

- **License**: LGPL

- **Maturity and stability**:
  
  - 630,446 lines of code.
  
  - 34 developers and 146 contributors.
  
  - Project registered at sourceforge on December of 1999.
  
  - On February 2003 Crystal Space was featured on sourceforge as the project of the month.

- **Community**: Although the forum and the mailing lists have very little participation, the developers hang on very frequently on the project's IRC channel.

- **Programing language and paradigm**: C++, other languages like
Python can also be used to write programs.

- **Overall features:**
  - It supports Windows, Linux, MacOS/X and more.
  - Based on a plugin system, so it can be extended easily.
  - 3D rendering is done with OpenGL. Features of the latest as well as older hardware are supported seamlessly.
  - State of the art features supported.
  - It's not only a graphics library, it also has modules for character animation, sound and networking between others.

- **Clients:** In comparison with other engines, only a few projects are using CS. Between these clients resides the Planeshift project. Planeshift is an open source MMRPOG.

### 4.1.2. Unreal Engine

Unreal Engine [epi08] is a non-free proprietary development framework. It provides a vast array of core technologies, content creation tools, and supports infrastructures required by top game studios, advanced visualization and simulation developers and creators of 3D animated content for linear productions.

- **License:** Proprietary. Unreal 2 engine license fees start on a base of 350000$, adding 50000$ for each additional platform. In addition there is a royalty fee of 3% of the revenue. Unreal 3 engine license fees are not published.

- **Maturity and stability:** There are various versions of the Unreal Engine available. Each of these versions have some titles developed by Epic Games and other companies. Unreal Engine may have some stability issues on some Linux distributions and different versions of other supported operating systems, mainly due to the closed source nature of the engine, and the lack of the possibility to build up optimized binaries for an specific architecture.
● **Community**: The developer community is restricted to the licensees of the framework, so it's closed to any type of external contribution or review. On the other hand, there is an established user base community of enthusiasts that are actively modifying or expanding the assets and contents of existing titles published by the licensees.

● **Programming language and paradigm**: C++, it also provides a custom scripting language known as UnrealScript.

● **Overall features**:
  - It only officially supports Windows and most current video game consoles. Although there are plans to port to Linux the latest version, at current time Linux is only supported by old versions.
  - Up to the state of the art features supported.
  - Designed with the ease of content creation and programming in mind, with the goal of putting as much power as possible in the hands of artists and designers to develop assets in a visual environment with minimal programmer assistance.
  - Highly modular, scalable and extensible framework for building, testing, and shipping games in a wide range of genres.
  - It supports animation, physics, audio, cinematics, particle effects, visual scripting, between others.

● **Clients**: Epic Games, the owner of the engine, has several titles published, by the other way, other companies like Microsoft Games Studio, or 2k Games are also users. In addition, it's also being used by television and cinema.

4.1.3. **id Tech**

Id Tech [ids08] is a 3d computer game engine developed by id Software.

● **License**: Proprietary for current technology (id Tech 4), old versions are all available under the GPL license.
Integration of a 3D rendering engine with a physics simulator

- **Maturity and stability**: There are 4 major versions of the engine, and version 5 is now under active development. It has published several titles by id Software and other companies. There are also open source projects using the GPL versions of this engine.

- **Community**: On the one hand, for the current technology the developer community is restricted to the licensees. On the other hand, there is a community of people around the open source version of the engine. In addition there is an established user base community of enthusiasts that are actively modifying or expanding the assets and contents of existing titles.

- **Programming language and paradigm**: C on old versions, C++ was introduced in id Tech 4.

- **Overall features**:
  - It supports Windows, Linux, Mac and most current video game consoles.
  - State of the art features supported.

- **Clients**: The owner of the engine, id Software, has Doom, Quake, and other titles published. In addition there are other titles developed by other companies, and some open source projects have done some games.

4.1.4. **Shark 3d**

Spinor is the owner and maintainer of Shark 3d [spi08], a realtime 3d graphics engine.

- **License**: Proprietary.

- **Maturity and stability**: Although shark 3d is young, there is at least one game published. On the other hand it is being used in television and industrial applications.

- **Community**: There isn't any public community. Licenses purchase the services needed.
● **Programing language and paradigm**: Object Oriented. C++, Java, an custom scripting language.

● **Overall features**:
  
  ○ It supports Windows, Linux, Mac and most current video game consoles. By the contrary licenses are only supporting Windows and some video game consoles.
  
  ○ Run-time live editing. The possibility to edit the scene when it's running.
  
  ○ Component system design. Architecture by individual components or modules responsible of encapsulating specific services.
  
  ○ It supports animation, physics, audio, networking, scripting, between others.
  
  ○ Up to the state of the art features supported.

● **Clients**: Funcom has published a game named “Dreamfall”, Siemens is using it on industrial applications, and the ARD/ZDF (German televisions) are using it to produce television broadcast material.

4.2. **Graphics**

4.2.1. **Ogre3d**

The Object-Oriented Graphics Rendering Engine (OGRE) [ogr08] is a scene-oriented, flexible 3D engine written in C++ designed to make it easier and more intuitive for developers to produce applications. It provides an interface based on world objects.

● **License**: LGPL

● **Maturity and stability**:
  
  ○ 1,015,010 lines of code.
Integration of a 3D rendering engine with a physics simulator

- 18 developers and 33 contributors.
- Project registered at sourceforge on February of 2000.
- On March 2005 Ogre 3d was featured on sourceforge as the project of the month.

**Community**: High participation in the forum.

**Programming language and paradigm**: C++, there are bindings available to Java, C# and Python.

**Overall features**:

- It supports Windows, Linux, MacOS/X and more.
- Based on a plugin system, so it can be extended easily.
- 3D rendering is either done with OpenGL or Direct3D depending on the platform.
- State of the art features supported.
- Highly customizable scene management done using an hierarchical scene graph.
- Supports character animation.
- API well documented.

**Clients**: A list of client projects is available at Ogre3d's website. Deck 13 is a German video game company that has already published several titles. In addition it is commonly used by most FOSS projects.

### 4.2.2. Irrlicht

Irrlicht [irr08] is a cross-platform high performance realtime 3D engine written in C++. It is a powerful high level API for creating complete 3D and 2D applications like games or scientific visualizations.
● **License**: zlib/libpng

● **Maturity and stability**:  
  o 532,685 lines of code.  
  o 10 developers and 9 contributors.  
  o Project registered at sourceforge on February of 2003.

● **Community**: The Irrlicht project community is smaller than previously seen projects. The main communication channel used are the project forums.

● **Programing language and paradigm**: C++, it also supports .NET bindings (C# and Visual Basic)

● **Overall features**:  
  o Multi-platform: Windows, Linux, MacOS/X and more.  
  o 3D rendering can be done with OpenGL or Direct3D. In addition a software renderer is also available.  
  o State of the art features supported.  
  o Scene Management done using an hierarchical scene graph.  
  o Supports character animation.

● **Clients**: Used in research by the Biologically Inspired Robotics Group of the École Polythenique Fédérale de Lausanne. There are several freeware and/or FOSS games available, between other commercial products. A full list of clients is available on Irrlicht home site.

---

9 **zlib/libpng license**: The zlib/libpng license is an OSI (Open Source Initiative) and FSF (Free Software Foundation) approved license. It is a permissive FOSS license compatible with the GNU GPL. The license is used by the zlib/libpng compression libraries, but it has been adopted by other FOSS projects.
4.3. **Physics**

4.3.1. **ODE**

The Open Dynamics Engine (ODE) [ode09], is an open source high performance library for simulating rigid body dynamics.

- **License**: BSD

- **Maturity and stability**:
  - 88,089 lines of code.
  - 25 developers and 11 contributors.
  - Project registered at sourceforge on the April of 2001.

- **Community**: The ODE community (users and developers) primary communication channel are the project mailing lists.

- **Programming language and paradigm**: C, C++.

- **Overall features**:
  - Multiplatform library (Windows, Linux, MacOS...).
  - Rigid bodies with arbitrary mass distribution.
  - Joint types: ball-and-socket, hinge, slider (prismatic), hinge-2, fixed, angular motor, universal.
  - Collision primitives: sphere, box, capped cylinder, plane, ray, and triangular mesh.
  - Collision spaces: Quad tree, hash space, and simple.
  - Contact and friction model.
  - Different time stepping methods available.

- **Clients**: About more than 100 applications use ODE. Game companies
use it on their game engines, such as “Call of Juarez” by “Techland”, “TitanQuest” by “Iron Lore Entertainment”, “World of Goo” by “2D Boy”, between others. In addition it's also used on FOSS projects like “OpenSimulator”...

4.3.2. PhysX

PhysX [nvi09] is a proprietary realtime physics engine currently owned by Nvidia. It is not only a software engine, this SDK supports a PPU (Physics Processor Unit). When the PPU is present on the system, the simulation runs on this dedicated processor rather than using the main system CPU.

- **License**: Proprietary

- **Maturity and stability**: It is a young technology, but it has already several titles published. It is one of the first implementations to use a PPU.

- **Community**: Information, documentation and other resources are restricted to the licensees.

- **Programming language and paradigm**: C++.

- **Overall features**:
  
  - It supports Windows, MaxOS/X and gaming platforms. Linux is also supported but with a lot of drawbacks (32 bits, PPU not supported...)
  
  - Complex rigid body object physics system
  
  - Advanced character control
  
  - Ray-cast and articulated vehicle dynamics
  
  - Multi-threaded/Multi-platform/PPU Enabled
  
  - Volumetric fluid creation and simulation
  
  - Cloth and clothing authoring and playback
  
  - Soft Bodies
Integration of a 3D rendering engine with a physics simulator

- Volumetric Force Field Simulation

- **Clients:** There are several game titles published using this technology. For example it is being used by the latest version of the Unreal Engine.

### 4.3.3. Bullet

Bullet [bul09] is a 3D Collision Detection and Rigid Body Dynamics Library for games and animation.

- **License:** zlib/libpng

- **Maturity and stability:**
  - 628,175 lines of code.
  - 5 developers and 11 contributors.
  - Project registered at sourceforge on September of 2005.

- **Community:** The main communication channel used by this project is the project forums.

- **Programming language and paradigm:** C

- **Overall features:**
  - Multiplatform library.
  - Provides state of the art collision detection, soft body and rigid body dynamics.
  - Discrete and continuous collision detection (CCD)
  - Swept collision queries
  - Rigid body dynamics including constraint solvers, generic constraints, ragdolls, hinge, ball-socket
  - Support for constraint limits and motors
Soft body support including cloth, rope and deformable

- **Clients**: Used by many game companies in AAA titles on Playstation 3, XBox 360, Nintendo Wii, PC and iPhone.

### 4.4. Sound

#### 4.4.1. OpenAL

OpenAL [ope09] is a cross-platform 3D audio API appropriate for use with gaming applications and many other types of audio applications. It is currently hosted and developed by Creative Technology, with support from Apple and FOSS enthusiasts.

- **License**: LGPL

- **Maturity and stability**:
  - 67,884 lines of code.
  - It has been around for several years now, and it has been used by several applications.

- **Community**: The project mailing lists are the main communication channel used for development and discussions about the future of the library.

- **Programming language and paradigm**: C.

- **Overall features**:
  - Multiplatform library.
  - The API is very similar like the OpenGL.
  - Offers a multi-platform abstraction over all the available hardware in the market.
  - Supported by one of the main sound card vendors.

- **Clients**: Extensively used by several applications and games published on
Integration of a 3D rendering engine with a physics simulator

different type of platforms including gaming consoles.

## 4.5. Summary of technologies

### 4.5.1. Complete engines

<table>
<thead>
<tr>
<th>Engine</th>
<th>License</th>
<th>Maturity &amp; Stability</th>
<th>Community</th>
<th>Programming language</th>
<th>Supported platforms</th>
<th>Features</th>
<th>Clients</th>
</tr>
</thead>
<tbody>
<tr>
<td>CrystalSpace</td>
<td>LGPL</td>
<td>9</td>
<td>7</td>
<td>C++</td>
<td>All</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>Unreal Engine</td>
<td>Proprietary</td>
<td>8</td>
<td>0</td>
<td>C++, Custom</td>
<td>Mostly All</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Id Tech</td>
<td>Proprietary/GPL</td>
<td>9</td>
<td>5</td>
<td>C, C++</td>
<td>Mostly All</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>Shark3d</td>
<td>Proprietary</td>
<td>8</td>
<td>0</td>
<td>C++, Java, Custom</td>
<td>All</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

### 4.5.2. Graphics engines

<table>
<thead>
<tr>
<th>Engine</th>
<th>License</th>
<th>Maturity &amp; Stability</th>
<th>Community</th>
<th>Programming language</th>
<th>Supported platforms</th>
<th>Features</th>
<th>Clients</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ogre3d</td>
<td>LGPL</td>
<td>9</td>
<td>9</td>
<td>C++, C#, Python</td>
<td>All</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Irrlicht</td>
<td>zlib/libpng</td>
<td>8</td>
<td>7</td>
<td>C++, .NET</td>
<td>All</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

### 4.5.3. Physics engines

<table>
<thead>
<tr>
<th>Engine</th>
<th>License</th>
<th>Maturity &amp; Stability</th>
<th>Community</th>
<th>Programming language</th>
<th>Supported platforms</th>
<th>Features</th>
<th>Clients</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODE</td>
<td>BSD</td>
<td>9</td>
<td>7</td>
<td>C, C++</td>
<td>All</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>PhysX</td>
<td>Proprietary</td>
<td>8</td>
<td>0</td>
<td>C++</td>
<td>Windows</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>Bullet</td>
<td>zlib/libpng</td>
<td>9</td>
<td>8</td>
<td>C</td>
<td>All</td>
<td>9</td>
<td>8</td>
</tr>
</tbody>
</table>
5. Project infrastructures and community

Any successful open source project will always have a community behind it. By the way, this chapter describes the key tools and applications required to maintain any open source project. The main goals are to make the project attractive to the general public and easy to be maintained by the developers.

A community of an open source project is understood as all the people that participates in the project in some way (e.g. contributions, testing, resources, ...)

5.1. Source code repository

The source code needs to be published, so people can download and test it, and send back contributions. In open source projects there is more than one developer involved in the project. Due to this, it is necessary to control possible conflicts between different versions maintained by different people, between other things.

There are currently several tools, each one has its advantages and drawbacks. Subversion\(^{10}\) was set up to maintain the framework code. Although subversion is not a distributed repository like git\(^{11}\) or mercurial\(^{12}\), it is currently the most extended and used tool in most projects, in addition it is the most supported by mosts IDEs.

In order to serve the code, subversion was configured in an apache web server with the mod_svn module.

5.2. Bug reporting and patches

It is important and useful to give to the community a way to report bugs, submit patches and contribute code. There are several applications that may aid in this functionality. Some examples of these applications are: Mantis\(^ {13}\), Bugzilla\(^ {14}\)

---

10 Subversion: Centralized version system. See http://subversion.tigris.org/
11 Git: Distributed version system. See http://git-scm.com/
12 Mercurial: Distributed version system. See http://www.selenic.com/mercurial/
13 Mantis: PHP based Bugtracker. See http://www.mantisbt.org/
and Trac\textsuperscript{15}.

Trac is a very simple forge software that can be very well integrated with sub-
version, it has a minimal wiki to maintain documentation, it allows to do some
project planning and browse the code. Obviously it has a bug reporting tool. Man-
tis and Bugzilla are very specialized in the bug-reporting process, but they are
too complex for the size and type of project and trac already meets the require-
ments.

In order to allow adding permissions to manage/use different features of trac,
the “authzgroups\textsuperscript{16}” plugin should to be installed.

\section*{5.3. Documentation}

Although trac already provides a wiki tool that can be used to maintain docu-
mentation, MediaWiki\textsuperscript{17} was installed as the portal for the project and entry point
to different documents and information about the project. One of the reasons of
using MediaWiki over the trac built-in wiki is that the first one has more features
and more available functionalities. Another point in favor is that people is more
familiarized with the syntax used by MediaWiki.

It is also interesting to install a syntax highlighting plugin, a custom made
plugin was installed. It uses GeSHi\textsuperscript{18} to generate the corresponding XHTML code.
However, there are now several implementations available in the MediaWiki’s
plugin repository.

In addition the documentation of the project generated by Epydoc it is also
published.

\textsuperscript{15} \textbf{Trac}: A small forge (bugtracker, wiki, repository, ...) written in python. See http://trac.edgewall.org/
\textsuperscript{16} \textbf{AuthzGroupsPlugin}: See https://trac-hacks.org/wiki/AuthzGroupsPlugin
\textsuperscript{17} \textbf{MediaWiki}: See http://www.mediawiki.org/
\textsuperscript{18} \textbf{GeSHi}: PHP Library that formats the syntax of different languages in XHTML. See http://qbnz.com/high-lighter/
5.4. **Message boards**

Message boards, distribution lists or IRC, are examples of means of communication for the community. Between the amount of available solutions, phpBB is one of the most extended, with the most available features.

A phpBB forum was installed and configured to host possible discussions, and ideas related with the project.

5.5. **Linking authentication and users with all applications**

Each installed tool requires an authentication backed in order to determine which user does each contribution: posting to the discussion board, reporting a bug, or uploading code to the repository.

There are several ways to store information about users and groups, permissions and authentication credentials. Using an LDAP directory is currently one of the most used mechanisms, but also one of the most complex ones. The most easy mechanism is to use the forum application as the backend to store all this information.

In order to be able to use the forum as the authentication backend, it necessary to install the following plugins:

- MediaWiki: Auth_PHPBB from [http://uber.leetphp.com](http://uber.leetphp.com), the original code was modified to implement the new password hashing used by the lastest phpbb releases.

- Subversion and Trac: Apache is configured to do Auth Basic authentication against a small python module that fetches the correct credentials from the phpbb database.

All the installed applications, and project information is publicity available at: [http://7d7.almlys.org](http://7d7.almlys.org), where you can find the documentation, build instructions and the source code.
6. Selected technologies

Different technologies, libraries and implementations were selected to assemble the framework. Figure 10 shows a summary of the main used libraries, each one of these different technologies are going to be justified in this section.

![Figure 10: Architecture](image)

6.1. Python as the programming language

Python [pyt09] is an high-level programming language created by Guido van Rossum in 1991. It is an interpreted and dynamic language like Perl, PHP, TCL, Ruby...

Its key characteristics are:
Multi-paradigm language supporting object oriented, imperative and functional paradigms.

Strong dynamic duck typing discipline.

Open source project with a community based development model managed by the Python Software Foundation.

Ideal as an scripting language.

Syntax and semantics are minimalistic.

Large and complete standard library with unit testing batteries included.

Cross-platform.

Easy to inter-operate with C extensions.

In general development in Python is faster than in other languages. The first reason is that python as a programming language is more simple and lacks of the language boilerplate required by other languages. Another advantage that saves development time is the compilation and linking time required by some languages on complex systems. For example, the time of building and running any sample C++ application of the Ogre3D package is higher than the corresponding application in Python from the python-ogre distribution. In addition the creation of C/C++ extensions for Python is a simple task, this allows access from python to any available C/C++ library. Thus the productivity increases due to all the development time saved.

Nowadays python and other interpreted languages such as Lua or PHP are being used in the development of complex applications because of the gains in the development time. For example, in the development process of a solution, sometimes the design path taken is not the correct one, and doing the corresponding changes, due to a redesign, will take more time in a system fully written in C/C++ than in a language such as Python.

On the other hand, python has the drawback of being a slow language in some cases. There is an important penalization in the general efficiency of the application. In order to avoid this penalization is important to check the possible
Integration of a 3D rendering engine with a physics simulator

application bottlenecks. A low level implementation in C of some modules can help in reducing these bottlenecks.

6.1.1. Python programing architectures

The usage of python as seen on [ril04], can be classified in different architectures depending on the usage of the language in the application. This varies from an application purely developed in python to an application purely written only on another language such as C++ (see figure 11).

---

**Figure 11: Python architectures**

- **No Python usage**: Python is not used or involved in any way.

- **Python as data model**: In the first level, python is used only for data modeling and representation. There are several situations were applications need complex ways to store and represent complex data structures. So instead of implementing or using complex data parsers, these applications use python to load, and save the application data.

- **Python as a scripting language**: One of the main key characteristics of python, is that it can be embedded into any application. When the main code of the application is written in languages such as C/C++ or Java, python is used in the dynamical parts of the application. For example, in game development when the main engine application is written in C++, all the game logic code is written in an scripting language such as Python. Therefore this allows a fast development of the game logic, without the penalization of having the entire engine written in python.

- **Python as a programing language**: In this architecture, python is used as the main programing language of the application, and it uses the extending capabilities of python by using modules written on other languages such as C/C++. Consequently, all the critical code that needs to remain as efficient as possible, remains written in a low level language and is used by python as an extension. For example, working with 3d graphics
require the usage of search algorithms, complex mathematical computations, and other algorithms that directly implemented in python would be affected by a considerable penalization, so this algorithms are implemented in C/C++.

- **Pure usage of python**: In this level, the entire application is written in python avoiding the usage of python extensions. It’s true that for some tasks it will be always necessary the usage of an extension. For example, if we want to write an application that used 3d graphics, the first level will be the OpenGL bindings of python. The difference between working directly with OpenGL and a graphics engine like Ogre or Crystal Space corresponds with the level of python architecture used.

In this project python is used as an integration tool, so python is used as a programming language. A set of technologies are integrated and made them available to the client applications as services for the framework. The most computing intensive code will reside in the implementation of the C/C++ low-level libraries. Some modules written in python can always be re-implemented in C, providing a faster version.

Although python provides a very complete cross-platform library, it is necessary to support the portability of any library written in C/C++.

### 6.1.2. Python specialized compiler – Psyco

**Psyco**\(^\text{19}\) is like a JIT (Just-In-Time) compiler that emits code on the fly. When psyco is enabled in a python application there is a considerable gain on the speed.

The approach used by psyco, to speed up the execution, is: from a block of code it generates multiple blocks of code specialized on some kinds of variables. The acceleration rate depends a lot on the type of code, being two and four times faster. The only drawback where there are not performance gains is in the algorithmic code, thus algorithmic code is recommended to be written in C and used as a python extension.

\(^{19}\) **Psyco**: See [http://psyco.sourceforge.net/](http://psyco.sourceforge.net/)
6.2. Building tools

In order to build all required dependencies, the first step is to build up each independent library, then the required python bindings are built up.

Building each independent library is a simple task that requires the GNU compilers GCC and G++, and each independent building tools required by the specific library. So tools like make, autoconf, automake, scons, cmake... are required. Finally to access the API of the library the different classes and methods are exposed or wrapped into the python world.

This process can be accomplished by using two main tools, swig and py++. Although there are other code generators available Pyste and SIP currently the most used ones are Py++ and SWIG.

6.2.1. SWIG

The Simplified Wrapper and Interface Generator [swi09], is a development tool aimed to connect C/C++ with high-level languages like python, php, java...

SWIG is capable of generating wrapper code for eighteen different languages. It fully supports all of ANSI C language, and most of the ANSI C++. Nested classes are not supported and advanced template techniques require the manual instantiation of all template classes.

In order to work with SWIG (see figure 12), it is necessary to write specific in-

![SWIG wrapper diagram](image-url)
Selected technologies

Alberto Montañola Lacort

Interfaces defining the functions to wrap. SWIG processes the input interface files (*.i) and generates the corresponding code for the desired language.

SWIG is distributed under a BSD like license.

6.2.2. Py++

Py++ [yak09] is an object-oriented framework for creating a code generator for Boost.Python library and ctypes package. The key features are:

- All Boost.Python\textsuperscript{20} features supported.
- Documentation is extracted from the source code and written as Python documentation strings.
- License is written at top of all generated files.
- Ensures the completeness of the bindings. Exposed declarations are checked for references to unexposed ones.

\textbf{Figure 13: Py++ Framework}

Py++ (see figure 13) uses the GCC compiler for parsing the source code files. It depends on GCC-XML that is a tool based on the GCC parser that generates XML files with the source code description. Then the pygccxml package is used

---

\textsuperscript{20} \textbf{Boost.Python}: Will be introduced later in this chapter.
Integration of a 3D rendering engine with a physics simulator to process the generated XML files, and with the corresponding user build instructions, Py++ generates the corresponding Boost.Python sources files.

Py++ is licensed under the Boost license\textsuperscript{21}.

6.2.2.1. Boost

Boost\textsuperscript{22} provides a set of free peer-reviewed portable C++ libraries. It offers implementation for complex data structures (arrays, bitmaps, circular buffer), design patterns (fly weight), input/output, mathematical functions, signals, strings, threads and python.

From the boost distribution, only the boost python library is necessary by the Py++ code generator. In addition if the renderer is compiled with threading support it also needs the boost thread library.

- **Boost Python**: The Boost Python library is a framework for interfacing Python and C++. It allows to expose C++ classes and functions to python and vice-versa.

- **Boost Thread**: It is a portable C++ multi-threading library, by providing an abstraction above the specific operating system threading libraries.

6.2.3. Conclusions

The two main code generators frameworks for exposing C/C++ libraries as python extensions have been introduced. Both of them have it's advantages and disadvantages:

- SWIG is more simple than Py++

- SWIG supports eighteen languages, when Py++ is only focused on python.

- SWIG does not require Boost python when Py++ uses it for the features

---

\textsuperscript{21} **Boost license**: The Boost license is an OSI (Open Source Initiative) and FSF (Free Software Foundation) approved license. It is a permissive FOSS license, in the style of the BSD and MIT licenses, compatible with the GNU GPL. The license is used by the Boost C++ libraries and other projects have adopted it.

\textsuperscript{22} **Boost**: http://www.boost.org/
Selected technologies

Alberto Montañola Lacort

provided. Thus SWIG is more simpler because it does not depend on a complex library like Boost.

- **Py++** relies on GCC-XML, that is a source code parser based on the source code of the GCC compiler, when SWIG implements its own parser.

- When SWIG consists on a stable release distribution, Py++ depends on other tools like GCC-XML, and at current time there is no stable version of GCC-XML.

- GCC-XML is often out of date with the latest version of the GNU C compiler.

- The Py++ build process requires more system resources than SWIG. During the development of this research project it was necessary to increase the size of the system swap because the gcc process took over all the memory on a system with two gigabytes of RAM. In addition building a complex library such as the one used for the graphics rendering can easily take about four or more hours of time.

- SWIG requires a lot of human work and maintenance, when Py++ is more automated. For an average project in order to generate the required code, if SWIG is used, then it is necessary to manually create all the interface files, and those need to be maintained in synchronization with the upstream project sources. On the contrary, Py++ requires less human work and the process is more simple.

In general Py++ is more complex than SWIG but requires less human time in the process of generating the desired source code. In this research project both tools are used in some way, because it depends on the tool-set used by the library authors. For example, for the rendering engine the python bindings are generated with Py++, historically SWIG was used, but the authors dropped it because it required too much time to maintain in comparison with Py++.

### 6.3. The graphics rendering library - Ogre 3d

From all the available rendering libraries, ogre 3d was selected because:
Integration of a 3D rendering engine with a physics simulator

- It is the most feature complete engine.
- It is highly maintained and keep up to date with the “state of the art” in a context of continuous technological changes.
- The usage of the design patterns, it is an excellent way to learn by example the usage of the most common ones.
- It is only a rendering engine.
- Python bindings are available and keep up to date with the latest version of the library.
- It is well documented and has a rich set of examples and tests with its source.
- The amount of client projects using it.

In order to build ogre, there are some third party libraries required:

- **Nvidia CG Toolkit**: The Nvidia CG toolkit\(^2^3\) provides a compiler for the Cg language. CG is an high-level programing language for writing shaders. A shader is a set of instructions for the GPU rendering pipeline that allows to perform transformations over the geometry. Different types of shaders are written an loaded by the rendering engine to perform different types of special effects. The library and the compiler are distributed by nvidia under a restrictive license, and there isn't yet any FLOSS alternative. It's possible to build Ogre without this library, consequently all the nifty special effects features that can be done with shaders will be disabled.

- **OpenEXR\(^2^4\)**: The ILM (Industrial Light & Magic) developed an special HDR (High dynamic-range) image file format. The library is distributed under the BSD license, and compiling ogre with it enables the support for this type of images.

- **Freeimage\(^2^5\)**: It is a common graphics library supporting a vast set of im-

---


age file formats. It is licensed under a dual license, the GNU GPL and the FreImage Public License (another FOSS license). The library provides the renderer with all the support required for processing different types of image formats. Ogre can also be compiled with another graphics library such as the DevIL\(^\text{26}\) library instead of FreImage.

### 6.4. The Graphical User Interface – CEGUI

In addition of the graphics render engine, there is also the need to display into the render window typical GUI controls, such as buttons, text controls,... When working inside the 3d window there is not access to the guest window manager widgets. There are some frameworks that implement fully customizable graphical user interfaces that can be easily integrated with the rendering engine.

The Crazy Eddie's GUI System (CEGUI\(^\text{27}\)) is a free library (MIT license) providing the wanted functionalities. It natively supports Ogre3D, Irrlicht, and directly over OpenGL or Direct3D. When it's used with Ogre3D the call that needs to be performed to render the GUI is automatically integrated into the renderer main loop. On the other implementations you need to manually call the “renderGUI” function after rendering each frame, so the GUI is currently displayed.

CEGUI is only a graphical user interface, this means that it does not listen for keyboard or mouse events, it is completely deaf. In order to update the position of the mouse, or to send keyboard events, it is necessary to use the “inject” functions to send information about the status of the keyboard or any other input controllers. This information is collected by the input system, that will be the responsible to fire up the corresponding input events.

### 6.5. The Input System – OIS

An input library is required to capture all events or movement generated by the input hardware (mouse, keyboard, joysticks, gamepads,...). The Object Oriented Input System (OIS\(^\text{28}\)) provides this functionality by offering a cross-platform

---

\(^{26}\) [DevIL](http://openil.sourceforge.net/)

\(^{27}\) [CEGUI](http://www.cegui.org.uk)

\(^{28}\) [OIS](http://sourceforge.net/projects/wgois)
abstraction over the different operating system specific input libraries.

It is possible to work with OIS by polling the input controllers, or by registering a callback function that will be called when an event occurs.

OIS supports capturing information from the system configured mouse and keyboard, and any number of available game controller devices. Each device is seen as an object, and you can register a function for each device.

The library is distributed under the zlib/libpng license.

### 6.6. The Physics Simulator – ODE

The physics simulator is responsible of simulating the behavior of all the objects described. The simulator is just a system that expects an input situation and computes the changes that will happen on this situation in function of the time.

The Open Dynamics Engine is a simple physics library with python bindings. The usage of the library is very simple, you need to create an “ode world”, an attach to it all the physical objects with the corresponding properties (mass, velocity, shape...). You can manipulate the physical world by adding, removing or altering objects to it. Then, on each frame you need to step the physics simulator so it computes the next state of the simulation.

In order to step the physics simulator, you need to compute the $dt$ time in function of the framerate of the renderer. So if we are running at 60 frames per second, then $dt=1/60$ seconds. The challenge, is to dynamically adjust the frame rate and the $dt$ when the system were the simulation is running does not have enough cpu time to compute everything in 1/60 seconds.

### 6.7. The Sound Library - OpenAL

Finally in order to reproduce music or sound effects, it is necessary to interact with the system soundcard. OpenAL is an abstraction library built on top of the different operating systems specific sound libraries (DirectSound on Windows, ALSA on Linux...).
Working with OpenAL is similar to how we have worked with the physics simulator. At the beginning it is necessary to configure and initialize the sound system. Then, in a similar way it is necessary to load the sounds and tell to the OpenAL sound world the characteristics of the sounds (position, volume, velocity...). This information needs to be extracted from the physics simulator. Finally, one of the most important steps is configuring the listener. The listener is the position in the sound world where the virtual “ear” is situated. This listener can be moved around the world.

6.8. **Legal compatibility of library licenses**

The license of the final product is affected by the licenses that govern the whole set of technologies and libraries used. These libraries are available under one or more licenses.

<table>
<thead>
<tr>
<th>Library</th>
<th>License</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG Toolkit</td>
<td>Propietary</td>
</tr>
<tr>
<td>OpenEXR + IlmBase</td>
<td>BSD</td>
</tr>
<tr>
<td>Freeimage</td>
<td>GPLv2 or later / Freimage Public License</td>
</tr>
<tr>
<td>pcre</td>
<td>BSD</td>
</tr>
<tr>
<td>zziplib</td>
<td>LGPL/MPL</td>
</tr>
<tr>
<td>OIS</td>
<td>zlib/libpng</td>
</tr>
<tr>
<td>CEGUI</td>
<td>MIT</td>
</tr>
<tr>
<td>Ogre3d</td>
<td>LGPLv2.1 or later / unrestricted (commercial)</td>
</tr>
<tr>
<td>gccxml</td>
<td>GPLv2 or later</td>
</tr>
<tr>
<td>pyggxml</td>
<td>Boost</td>
</tr>
<tr>
<td>Boost</td>
<td>Boost</td>
</tr>
<tr>
<td>scons</td>
<td>MIT</td>
</tr>
<tr>
<td>Python-ogre</td>
<td>LGPLv2 or later</td>
</tr>
</tbody>
</table>

The Nvidia CG Toolkit is distributed with a proprietary license, mainly because the toolkit is restricted software. Point two of the license follows:

2. **GRANT OF LICENSE**
2.1 Rights and Limitations of Grant. NVIDIA hereby grants Customer the following non-exclusive, worldwide, royalty-free, non-transferable right to use the SOFTWARE, with the following limitations:

2.1.1 Rights. Customer may use, reproduce, distribute, publicly display and publicly perform the SOFTWARE.

2.1.2 Limitations.

No Reverse Engineering. Customer may not reverse engineer, decompile, or disassemble the SOFTWARE, nor attempt in any other manner to obtain the source code.

No Modification. The SOFTWARE may be redistributed providing that distributed Cg compiler and runtime binaries are unmodified, except for decompression and compression.

No Rental. Customer may not rent or lease the SOFTWARE to someone else.

No Support. NVIDIA will not be responsible for providing maintenance and support to Customer or any other end users for the Software distributed by Customer or others.

The license grants rights to use and distribute the library, but is incompatible with the GPL terms. Copyright holders of GPL applications that need to use this library may grant an exception allowing its usage.

The Berkeley Software Distribution (BSD) License is a permissive FLOSS license compatible with the GPL.

Copyright (c) <year>, <copyright holder>
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

* Neither the name of the <organization> nor the names of its contributors may be used to endorse or promote products de-
The MIT license is FLOSS permissive license compatible with the GPL.

Copyright (c) <year> <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

The software is provided "as is", without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

The zlib/libpng is another permissive FLOSS license, it is also compatible with the GPL.

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:
Integration of a 3D rendering engine with a physics simulator

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

The Boost license is also another permissive FLOSS license like the MIT and BSD licenses. It is also compatible with the GPL.

Boost Software License – Version 1.0 – August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization obtaining a copy of the software and accompanying documentation covered by this license (the "Software") to use, reproduce, display, distribute, execute, and transmit the Software, and to prepare derivative works of the Software, and to permit third-parties to whom the Software is furnished to do so, all subject to the following:

The copyright notices in the Software and this entire statement, including the above license grant, this restriction and the following disclaimer, must be included in all copies of the Software, in whole or in part, and all derivative works of the Software, unless such copies or derivative works are solely in the form of machine-executable object code generated by a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

The resulting assembled project source code is currently licensed under the GPLv3, although the idea is to release stable and mature versions under the LGPL. The licenses from the technologies used are enough permissive to license the project under a compatible custom or any other FLOSS license.
7. Engine Design - Assembling everything

7.1. Bootstrap

7.1.1. Introduction

One of the main issues when working with systems that rely on a variety set of libraries and tools is the process of resolving all the dependencies required for the function of the system. This process consists on:

1. Knowing the required libraries, tools...
2. Downloading the correct version of each tool/library.
3. Configuring and building the tool/library.

Bootstrap is a small python application responsible of doing all the initial steps required for building the engine.

The idea behind Bootstrap is to have a small initial multi-platform tool that can be easily downloaded from Internet. Then by running the application it will automatically download and build all required dependencies in order to run the engine.

7.1.2. Usage

Bootstrap requires at least two configuration files:

- The first configuration file “config.xml” (see section 13.1) is an XML file containing a set of properties to configure the destination directories used by Bootstrap to store and install the downloaded dependencies.
- The second configuration file “bootstrap.xml” (see section 13.2.) is another XML file that contains information of how to download and build all dependencies.

When Bootstrap runs it reads the corresponding configuration files and correctly sets up the environment. The normal user will not have rights to install
anything in the system, so BootStrap uses a fake root destination directory. The library, include and python paths are correctly set up with the corresponding compiler flags.

BootStrap expects an action to perform and a list of modules. For BootStrap a module is a library or a tool that needs to be downloaded, configured and installed. The actions supported are:

- **get**: Downloads the module from Internet. It can be downloaded from a web or ftp server, or it's source can be obtained from a code repository such as subversion. At current time cvs and subversion are the only ones supported, but it's easy to add new methods.

- **update**: If the module has been downloaded from a code repository it checks for updates and downloads a new version from the repository. It’s equivalent to doing “svn update” or “cvs update” over the downloaded code.

- **remove**: The module is deleted.

- **patch**: Some modules need to be patched, mainly due to some bugs not fixed by the module author.

- **build**: Configures and builds the module. It first runs the correct “configure” script with the correct configuration switches and then it builds up the module using the corresponding build system (make, cmake, scons...)

- **install**: Installs the binary module into the fake root directory.

- **auto**: It is a shortcut for doing the get, patch, build and install actions at the same time.

BootStrap can be fully configured by changing the following configuration keys:

- **BootStrap.downloadPath**: Specifies the destination path for storing the downloaded modules.

- **BootStrap.outputPath**: Specifies the destination directory for unpack-
Integration of a 3D rendering engine with a physics simulator

- **BootStrap.prefix**: Sets up the fake root directory where the binary run times will be installed.

- **BootStrap.patches**: Specifies the directory where bootstrap finds the different patches to apply.

- **BootStrap.manifest**: Specifies the name of the BootStrap XML manifest file.

### 7.1.3. Involved classes

Figure 14 shows all involved classes in the BootStrap application:

![BootStrap diagram of classes](image)

- **BootStrap**: Is the main application class responsible of loading up the configuration files, setting up the environment, creating the Tool Factories
and performing the desired operation.

- **ToolFactory** and **DownloadFactory**: These factories are responsible for the creation of specific tools. The download factory is only specialized in tools responsible in the process of obtaining the sources.
- **Tool**: Any command or operation that may be requested by the build instructions of a module.

### 7.2. Engine facade

The main python package that encapsulates all the framework functionalities is the engine package. Engine provides a facade to the client code that will use the framework.

![Engine Diagram of Classes](Figure 15: Engine Diagram of Classes)

It is responsible of loading the configuration, importing, initializing and keeping a reference of all the required subsystems and running the main system.
Integration of a 3D rendering engine with a physics simulator maintenance loop. Then it should pass the reference of each individual subsystem to the client code that needs it. In addition it also initializes the Python Psyco accelerator if it is found installed in the system.

The engine will maintain a set of subsystems, these subsystems are derived from the class “SubSystem”, see figure 15. Any subsystem will have:

- **An specific configuration**: The subsystem constructor expects a dictionary with a set of configuration settings.
- **A log file**: Each subsystem stores the debugging information in it's own exclusive log file.
- **An event processor**: Subsystems will receive events from other subsystems, but the subsystem must be previously registered.
- **An event generator**: Subsystems, may sent events to other subsystems.

There can be only one instance of the engine facade (singleton pattern). It is started by requesting this unique instance, and consequently by calling the method run.

```python
e = Engine(options)
e.run()
```

Engine expects a python dictionary containing a set of dictionaries with specific options. This configuration options are then passed to the corresponding low-level implementations.

Some of the main configuration settings expected in the “global” dictionary are:

- **Engine.OgreRenderer.PluginsPath**: Specific configuration setting for the Rendering library, it sets the path of the library plugins.
- **graphics.fullscreen**: The render window will take over the full screen.
- **graphics.width** and **graphics.height**: Sets the window size
- **system.logdir**: Sets the directory were the log files will reside.
● **python.psyco**: Enables the Python specialized compiler.

● **input.actionmap** and **system.hooks**: Sets the path to the ActionMapper and HookManager definition files.

For automatically setting up the correct parameters and launching the engine, there is another class called sd7Engine. It is the main application, it reads the configuration file and mixes the options with the command line ones. Then it passes these options to the engine by requesting it’s unique instance, and finally it blocks in the run method of the engine.

Client code should not access directly to the low level code, or specific implementations of the renderer, the gui, audio system... it will only interact with the Engine facade and some of the provided subsystems.

For example, a client application that needs access to the GUI will request it to the Engine facade:

```python
 gui = Engine().getGUI()
gui.loadView("console.layout")
gui.subscribeEvent("Console/ConsoleWindow/ConsOkBtn",
                      "PushButton/EventClicked", self.onInput)
```

This sample code requests the GUI subsystem, then it loads the view stored in the “console.layout” file, and finally it subscribes a method called “onInput” to the event generated by a button of the view.

The subsystems that the Engine will load by default on startup and will provide a specific facade to the client code are:

● **renderer**: The graphics renderer.

● **gui**: The graphical user interface.

● **input**: The input subsystem.

● **hookManager**: All the system logic.

● **physics**: The physical simulator.

● **sound**: The sound system.
Integration of a 3D rendering engine with a physics simulator

- **net**: Communications system.
- **world**: The world manager.

Client applications will usually use only the gui and world facades because the world facade is automatically responsible of interacting with the other subsystems.

The maintenance main loop should periodically update all the subsystems. The algorithm currently implemented is:

```python
def run(self):
    frame_freq = 1/70.
    while 1:
        t0 = time.time()

        self._physics.step(frame_freq)
        self._worldmgr.update()
        if not self._renderer.renderOneFrame():
            break

        loop_time = time.time()-t0
        sleep_time = frame_freq - loop_time

        if sleeptime > 0:
            time.sleep(sleeptime)
        else:
            print "DROPPING FRAMES"
```

The goal of the maintenance loop is to run the simulation at a fixed frame rate, in this case 70 frames per second. In order to force an specific frame rate, it is necessary to perform a sleep operation. This sleep will never have the same fixed time and needs to be computed on each frame.

At the beginning a measure of the time is keep in the t0 variable. Then in order to achieve a real-time simulation, the physics simulator is stepped $1 / frame_rate$. An higher value, for example $2 * (1 / frame_rate)$ would cause the effect of a simulation two times more faster than in the real world.

After the simulator has been updated then the world manager is responsible
of keeping in synchronization the data from the simulator with the other subsys-
tems. Thus the world manager will get the position, orientation, velocity data
from the physics world and set it on the other worlds maintained by each subsys-
tem.

The next step is requesting to the renderer the rendering of another image, it
will automatically fire up a frame event that will be received by the input and gui
subsystems so them can run it's update cycle.

Finally the time is measured again and the $t0$ measure is subtracted in order
to obtain the time consumed in the update algorithm, this time is know as the
loop time. The difference between this measure and the frame frequency is the
idle time that the system will have to do other tasks.

$$idle\_time = \frac{1}{frame\_rate} - loop\_time$$

In order to have a constant frame rate, the idle time must be always higher
than zero. In consequence a negative loop time will reduce the amount of frames
displayed per second, and the physics simulation will be negatively affected. This
maintenance algorithm does nothing to avoid it. Because the physics time step
needs to be adjusted for the new frame rate.

In conclusion, if the frame rate drops to 20 or a smaller value, then the simu-
lation can not be computed in real time in the machine where it is running.

### 7.3. World Manager

The world manager is responsible of loading, modifying and altering objects
into the simulation. It is the top level facade that client code should use to per-
form actions over the simulation.

It maintains a list of all objects in a world, these objects have graphical, physi-
cal, logic and sound information. The world manager is responsible of synchroniz-
ing the state of the objects on each independent subsystem.

On each simulation step, the world manager will perform the next actions:

1. Query the simulation for the stat of all objects.
2. Update the renderer with the new information.

3. Update the sound system with the new information.

In addition, it should provide to the client code access to the objects. Client code should be able to search or manipulate objects. Figure 16 shows the main classes involved with the World Manager.

The Physics simulator, the rendering engine, and the audio system are responsible for maintaining specialized information about the different objects of the scene. The world manager needs a way to synchronize all the information together, and should offer a common abstraction of the functionalities to the client code.

The involved subsystems provide the required mechanisms to create, query, manipulate, and destroy objects of their world. The world manager uses this interfaces through a SceneObject that contains references to all involved objects.

The client code only needs to interact with the World and SceneObject classes.
The following example code shows how to keep in synchronization the graphical representation of the object with the physical status of it:

```python
def update(self):
    for obj in self._dynaObjects.values():
        obj.node.position = obj.physical.position
        obj.node.orientation = obj.physical.orientation
```

*DynaObjects* is a dictionary containing a list of all objects that are not static and have a physical object associated to it. Some objects may have a physical object associated to it and can be static, like the ground of the simulation, and other objects may not have a physical object associated at all.

For each object, the position and orientation information is requested from the physics simulator object “*physical*” and set to the rendering object “*node*”.

### 7.4. Renderer

The renderer is responsible of maintaining all the information about the graphical objects in the simulation. It provides a facade to be used mainly by the world manager. Some of the basic tasks are: Loading, altering and removing any object from a scene. Other more advanced tasks are: Playing animations, modifying textures, creating a mesh in a manual way...

For the renderer, the current implementation used is the Ogre 3d engine. There are two ways to design the main simulation loop.

A) **startRendering**: The renderer will block into it's own main loop, and will render as many frames per second as possible.

B) **renderOneFrame**: The renderer will issue the rendering of only one frame.

If the main loop resides in the renderer, then it is necessary to register an event callback that updates all the subsystems that require it. By the other way if the non-blocking approach is done it will be necessary to build up a custom main loop as seen on chapter 7.2.

The main client class that will work with the rendering engine is the world manager. The rendering engine offers functions to create, modify and destroy ob-
Integration of a 3D rendering engine with a physics simulator

jents such as meshes, lamps, cameras.

7.5. **Input**

The input subsystem has an update method that tells OIS to query information about the status of all devices and fires up any events.

The ActionMapper loads up the definition map of all possible events, see chapter 13.3. This map contains an association of the event with the corresponding controller combination. The ActionMapper contains two maps of associations: the first map contains associations with a higher priority, these input events will never reach the gui event manager and will be always fired up. The second one is for the rest of the events, those events will be fired up only if the gui does not process them. For example, if the keys \( w, a, s, d \) are associated to movement events, it is not wanted to fire up those events when the user is writing a message in the gui.

When an input event occurs the following logic path is taken:

```python
if ActionMapper.PriorityEvents.has(inputEvent):
    if ActionMapper.PriorityEvents(inputEvent).fire():
        return
if gui.processEvent(inputEvent):
    return
if ActionMapper.Events.has(inputEvent):
    if ActionMapper.Events(inputEvent).fire():
        return
if HookManager.processEvent(inputEvent):
    return
print "Unprocessed event: %s" (inputEvent,)
```

An example of event with priority is the “exit” event associated with the key “ESC” or the “CTRL-C” key combination. The ActionMapper never sends events to the gui, it will always send them to the HookManager.

Client code should use the ActionMapper to define and process events, instead of processing raw input events, although the later ones are always sent if them are not mapped.
The GUI will only process events if it has the focus, when the gui has the focus almost all events die on the GUI, although them do not reach the HookManager.

When an event reaches the HookManager it is propagated to all attached logic scripts and will stop on the script that processes it.

### 7.6. GUI

The GUI facade provides access to most common GUI elements. Its main functionalities are the possibility to load up windows layouts from disk, and attaching specific client code to events that may be fired by the layout. It also needs to provide a way to access the gui controls, so them can be altered or queried for information.

Client code should also be able to disable or not the gui focus, and hide or display the gui or individual gui windows.

The GUI will only fire up events to the client code that requested it. For example a client application may request to be notified when an specific button of the interface is clicked.

### 7.7. Hook Manager

![Diagram of Hook Manager classes]

Figure 17: Hook Manager involved classes.

The Hook Manager subsystem (see figure 17), is responsible creating, updat-
Integration of a 3D rendering engine with a physics simulator

... binding and destroying one or more Controllers. This subsystem encapsulates all the logic of the framework, it also propagates the generated events to all the registered controllers.

The Hook Manager provides the following methods:

- **getController**: Returns or creates an instance of the selected controller.
- **destroyController**: Destroys the selected controller.
- **findController**: Searches for the existence of an instance of a controller.
- **registerHook**: Registers a controller with the associated name. It's possible to register the same class under different names.
- **listControllers**: Returns a list of all running, stopped and registered controllers.
- **startControllers**: Automatically starts up the controllers configured to start at engine initialization.
- **update**: Performs the update loop.

A Controller is a class that will run a specific application. Client code will be derived from the Controller class, this class like the subsystem class has its own log file and event processor. It has the following methods that may be overridden by client code:

- **initialize**: Implements the initialization code.
- **terminate**: Implements the required code to shutdown the application.
- **log**: Writes a message to the associated log file
- **processEvent**: May be derived by the client to process specific events.
- **onAction**: May be derived to process Action related events sent by the ActionMapper.
- **onAxis**: May be derived to process events generated by devices with axis, like the mouse or a joystick.
● **onFrame**: received by the renderer before a frame is rendered.

All logic hooks are defined in a configuration file, they are like applications that will run inside the framework.

The hookmanager has the following responsibilities:

- Reads the definition file with all the subprograms.
- Starts and stops a subprogram when requested.
- Sends events to all the subprograms.

The following code shows a very simple subprogram that will terminate the engine when the “exit” action is received.

First it is necessary to register the corresponding hook in the *hooks.xml* configuration file:

```xml
<controller name='basicInput' bound='sd7.system.basicInput.MyBasicInputHandler' autostart='true' target='main' />
```

Then it is necessary to add the corresponding key mapping in the *input.xml* configuration file:

```xml
<input device='keyboard' >
   <action name='exit' key='ESC' />
</input>
```

Finally the code of the application will look like:

```python
from sd7.engine.controller import Controller
from sd7.engine.Events import EventType

class MyBasicInputHandler(Controller):
    def __init__(self, params):
        Controller.__init__(self, params)

    def processEvents(self, evt):
        if evt.getType() == EventType.ACTION:
            if evt.getObject() == "exit":
                # Terminate the engine
```

```
An Event is a class that contains the type of event, and a associated object. Depending of the event the contained object may be a string or an object. For example, ACTION events will be associated to a string with the name of the action. A MOUSE_MOVE event contains an object with the mouse state.

7.8. Physics

The responsibilities of the physical simulator are:

- Creation of the physical world.
- Manipulate, add or remove objects into the world.
- Fire up collision events between objects.

The physics simulator needs to be updated on each frame with the desired
time to simulate (normally $1 / \text{frame rate}$) so the simulation can run in real-time.

The world manager queries up the status of the world, updates the other subsystems and performs changes over the physics world.

Figure 18 shows a screen shot of an example physics simulation. On this simulation about 200 balls are created in the sky, and them just fall and collide together.

### 7.9. Sound

The sound system is responsible of setting up the correct sound environment and playing the corresponding sounds when it corresponds. The world manager is the responsible of submitting correct information about the position of the sounds in the world and the location of the sound files in the disk.

### 7.10. Network Manager

The Network Manager is responsible of sending and receiving messages between other machines sharing the same simulation. It is necessary to synchronize the status of the simulation over all the peers participating on it.


8. Conclusions

During the last decade, the development and usage of real time applications has left an important mark in the computing engineering industry.

One of the main reasons that explain the success of these applications is the gaming sector, where gaming companies are expending a lot of resources to generate entertainment products with graphics and effects near to the photorealism. Graphics engines have been around for more than ten years, the usage of physics simulators were introduced about four years ago having an important impact.

The basic concepts, ideas and architectures behind these technologies have been studied in detail in this research project. In addition a deep requirements analysis have been done to know the specific requisites from different situations and points of view. Furthermore, the most representative technologies currently available in the market have been presented and studied in detail. The presentation of the state of the art has been presented with this study, showing that the most feature complete engines are commercially distributed under restrictive licenses that are only available to a few select number of lucky companies. One of the furthest goals of this research project is to achieve someday a FLOSS engine capable of competing with the top ones in a similar way of how Linux has won a privileged position in the operating system market.

Setting up a source code repository has also been one of the tasks done during the development of this project, in addition of the installation of a forge with a wiki to store documentation, information and other related resources. Besides, a discussion forum was also installed.

Also a complete study of Python and the involved process required to generate the required Python extensions from the C/C++ libraries was done to understand and work with them. In addition the advantages of using Python as the programming language were discussed and seen in practice. The usage of each specific technology was also justified, with the study of compatibility of its associated license.
During the development of this project, one of the main troubles or drawbacks is the time that the average developer needs to invest in the development tasks. On the one hand, in the traditional way it was necessary to download each library, configure it, build it and install it into the system or a runtime directory. Then it was necessary to set up the correct environment variables. Although the task can be fully automated in part with a simple shell script, there are still some drawbacks. For example the python-ogre distribution already comes with a script that automates a bit the task, but in order to update only one library it was necessary to modify at least five different files. On this project an automated build system was provided to fully automate the task.

At the beginning of the project it was necessary at least one week worth of time to build all dependencies, during those initial days the BootStrap application was written to simplify the process, as an example it was possible to update all the dependencies to the latest version as for date of June 2009 by doing a few modifications to the BootStrap manifest file. Another example is the constant recommendation by the maintainers of the python ogre bindings, to use precompiled binaries due to how difficult is the build process for the average user. On this project the user only needs to type one command, take a long coffee while it’s compiling and then it can start working with the framework.

Finally the design and architecture used for assembling the framework was presented with the most important subsystems. Them have been integrated into the engine providing a simple facade to client applications. This facade is a key part of the design of the framework, because it keeps encapsulated and hidden all the low level and complicated things from the client code. The framework provides rendering services, an input system supporting different types of devices, a graphical user interface and a slight not complete integration of the physics simulator. The integration of the audio and communications subsystems are still not finished but them are considered inside the global design view.

The framework allows the possibility to run applications that will create GUI windows, objects in the scene, or doing other simple tasks. It is a first step for a project that will take more years in order to reach a satisfactory feature complete set version. These applications can be started, stopped and reloaded with new modifications without the need of stopping the running engine. This in fact saves
Integration of a 3D rendering engine with a physics simulator

a lot of development time.
9. Future work and open research lines

During this project a first prototype of the framework was done, this prototype sets an important feature set base that can be extended and customized in several ways. In fact there is a considerable amount of features and possibilities to extend the framework.

- **Redesign and refactoring**: One of the key benefits of python, is that the refactoring process is very fast. It is possible to completely change a complex system in a small amount of time.

- **C++ is faster**: Python serves very well in the design and model phase of any system, but it still has the drawback of being slow. Once a python application has been finished and has a perfect architecture design, then it can serve as a base for a C++ version of the project. The port can be done in a module basis, without the need of rewriting the whole application again.

- **Physics simulator**: The current physics simulator integration is not yet fully finished.

- **Audio subsystem**: It would be nice to have an Audio subsystem to be able to play sound on collision events, or other things.

- **Communications subsystem**: In order to develop and work with simulations that will involve the participation of more machines, there is the need of implementing a communication layer for sending updates about the status of the local world of each machine. Writing a client/server version would be an easy an simply approach, mainly because the server will maintain and define the simulation rules. By the other way a peer to peer implementation would be a complete challenge, being at current time unknown territory. One of the problems on a distributed environment is searching the correct synchronization algorithms and methods. Another challenge are the security related issues.

- **Artificial intelligence routines**: Provide services to solve some common problems like the path finding problem, feeding animated characters some
behavior information, between other things.

- **Data persistence**: Implement a way to store and read back information about the simulated scene.
10. Appendix A – Download and build

10.1. Minimal Recommended Hardware requirements

This is an approximation of what may be needed to work with sd7, in order to build or run it.

10.1.1. Building Requirements

In order to build up the framework it is needed at least:

- Operating System: Any that has the required compiler and tools and is capable to build everything (the application is being developed under Linux)

- Processor: At least an AMD or Intel processor of 2 GHz or more (Dual core processors better)

- RAM: At least 1 Gigabyte (2 recommended), also if you don't have enough available memory, then you may need at least 2 gigabytes of swap or the compilation process may fail.

- Video: A video card is not necessary to build, but it is required to run the application.

- Disk Space: You need at least 3 Gibabytes of free space.

10.1.2. Running Requirements

These requirements are an approximate idea of the type of system required to run the engine, but the real requirements depend on the type of application that is going to be built, and the amount of data that is going to be processed.

- Recommended OS: Any recent flavor of GNU/Linux. (Other Operating systems may work, or not)

- Processor: At least a 1.5 Ghz processor.

- Memory: At least 512 MBytes for low demanding applications. Depending
of the usage of the textures and the amount and complexity of the world to be loaded, more memory may be required or the framerate may drop to 0 if your system starts to trash. (More memory may be needed depending on the resources consumed by the desktop environment and other running applications).

- Video: A compatible video card with 3d graphics acceleration. Recommended suggestions:
  - Nvidia Geforce 4 or higher.
  - Ati Radeon 9600 or higher.
  - Intel and other integrated cards may also work

### 10.2. Downloading and Building 7d7

#### 10.2.1. Linux and compatibles Instructions

#### 10.2.1.1. Prerequisites

It is required a recent version of python (2.5). Other tools may also be required.

#### 10.2.1.2. Deb based systems (Debian/Ubuntu)

These instructions at current date are known to work on Debian Lenny.

As root, or appending "sudo" these commands should be run to install all dependencies.

```
$ apt-get install python python-dev python-ctypes 
libexpat1 libexpat1-dev libfreeimage3 libfreeimage-dev 
subversion cvs gcc g++ autoconf automake libtool patch 
libxaw7 libxaw7-dev libpcre3 libpcre3-dev libglut3-dev 
cmake libzzip-dev libxxf86vm-dev libfreetype6-dev 
automake libtool zlibg-dev pkg-config libxrandr-dev
```
10.2.1.3. Getting the Bootstrap Script

In order to get the bootstrap script, it is necessary to point any subversion or compatible client to the following address.

```
$ svn co https://7d7.almlys.org/svn/7d7/trunk sd7
```

10.2.1.4. Running The BootStrap Script

The bootstrap script should be run with the next packages. (If the OS has already installed one of the libraries/packages, them may be removed from the command line.)

```
$ cd sd7
$ python bootstrap/BootStrap.py auto cg ilmbase openexr \ freeimage gccxml pygccxml boost ois cegui ogre scons \ $ python-ogre
```

The script will automatically download, build and install all required libraries and tools in order to build all dependencies used by sd7.

If everything worked fine, then it should be possible to launch sd7 without any problems.

10.2.1.5. Testing that everything is working

In order to check that everything is working, the next commands need to be run:

```
$ scripts/shell.sh
$ python sd7/sd7Engine.py
```

Then a black window should pop up and stays, this means that the engine is successfully running, in the other hand an error or exception will appear or the window will vanish or don't open.

These black window is the only thing that will be shown, it is only the engine it is missing graphics an all the other stuff.
11. Appendix B – Download and install a pre-compiled version

The running requirements are the same as seen on chapter 10.1.2.

In order to download and install the pre-compiled version, it is necessary to have python and subversion installed.

The following commands will install everything on the system.

```
$ svn co https://7d7.almlys.org/svn/7d7/trunk sd7
$ cd sd7
$ python bootstrap/BootStrap.py auto binary_depends
```

Finally to test the application, you need to run:

```
$ scripts/shell.sh
$ python sd7/sd7Engine.py
```
12. Appendix C - Integrating CoDiP2P as a communications module using Jpype

CoDiP2P is a distributed peer to peer computing system being developed at the Grup de Computació Distribuïda (group of distributed computing) of the University of Lleida.

This platform is written in Java, and offers a communications interface aimed for distributed computing problems. It currently uses Jxta\(^{29}\) as the low level peer to peer communications library. Jxta is an abstraction platform built over TCP/IP that provides all the services and required functionalities to built up peer to peer applications.

The goal is to be able to integrate into the project the CoDiP2P platform as a communications module. The problem is finding if it is possible to call Java classes from Python code.

Python can be embedded in C/C++ applications, it can also be embedded in a native or not native way in Java applications. Python can also be extended with C/C++ extensions. The situation is an application written in python were the control path resides on python code. There is a possible way to embed the Java virtual machine into Python applications.

Jpype\(^{30}\) is a python module that builds a “pipe” between the Java and Python worlds. Jpype does this bridge by embedding the Java virtual machine.

The main reasons to embed Java in this way and not the other way:

- The desire to maintain the control of the application in Python.
- Jython does not provide access to C/C++ python modules.

So when the Jpype module is installed an available on the system, it’s possible to interact with the CoDiP2P facade. For example:

---

\(^{29}\) Jxta: See [https://jxta.dev.java.net/](https://jxta.dev.java.net/)

import CoDiP2P
import time

class MyApp(object):
    def __init__(self):
        # Get and configure the CoDiP2P Platform
        self.cp2p = CoDiP2P.CoDiP2P.getInstance()
        self.cfg = self.cp2p.getConfigurer()
        self.cfg.setString("port","7000")

    def run(self):
        # Initialize the CoDiP2P platform
        self.cp2p.initialize()

        while(True):
            # Get My Jobs
            print "My Jobs"
            jobsmgr = self.cp2p.getJobsMGR()

            for j in jobsmgr.getMyJobs().toArray():
                print j.getName() + " " + j.getOwner().getName()

            print "Local run jobs"
            for j in jobsmgr.getLocalJobs().toArray():
                print j.getName() + " " + j.getOwner().getName()

            time.sleep(10)

            self.cp2p.terminate()

if __name__ == "__main__":
    app = MyApp()
    app.run()

The example imports the CoDiP2P module and interacts with the facade class CoDiP2P. All operations over the platform are done over this facade, there is absolutely no need to import any other class, or to know the way the platform has been implemented.
The following code configures and starts the Java virtual machine, and hides from the client code the creation of the JVM and the mechanisms required to import the desired CoDiP2P classes.

```python
import sys
import os

__all__ = ["CoDiP2P"]

def _config():
    JPYPE_PATH = "/home/alberto/doc/dev/jpype/runtime/lib/python2.5/site-packages"
    JAVA_HOME_PATH = "/usr/lib/jvm/java-6-sun-1.6.0.13"
    sys.path.append(JPYPE_PATH)
    os.environ["JAVA_HOME"] = JAVA_HOME_PATH

    return _config()

def _getCoDiP2P():
    CLASS_PATH = "/home/alberto/NetBeansProjects/CoDiP2PngFacana/dist/CoDiP2PngFacana.jar"
    import jpype
    jpype.startJVM(jpype.getDefaultJVMPath(),
                   "," , "-Djava.class.path=%s" % CLASS_PATH)

    jpype.java.lang.System.out.println("Importing CoDiP2P")
    return jpype.JPackage('CoDiP2P')

class _CoDiP2P(object):
    def __init__(self):
        self.CoDiP2P = _getCoDiP2P()
    def getInstance(self):
        return self.CoDiP2P.CoDiP2P.getInstance()

CoDiP2P = _CoDiP2P()
```

The example code sets up all the correct environment variables: JAVA_HOME, CLASS_PATH. So the correct JVM can be found in the system, and the CoDIP2P platform is also available. Then it searches for the correct Java package and returns the desired facade class to the client code.
13. Appendix D - Document Type Definitions

13.1. Project Configuration

The engine configuration files consists on a set of different named sections with a set of key paired options. This configuration is mixed with the command line passed options. The command line options will always have more priority.

Example:

```xml
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE sd7config SYSTEM "http://7d7.almlys.org/spec/draft/sd7Config.dtd">
<sd7config>
  <section name='global'>
    <option name='graphics.fullscreen' value='false' />
    <option name='graphics.width' value='800' />
    <option name='graphics.height' value='600' />
  </section>
  <section name='engine.OgreRenderer.plugins'>
    <option name='RenderSystem_Direct3D9' value='disabled' />
    <option name='RenderSystem_GL' value='enabled' />
    <option name='Plugin_ParticleFX' value='enabled' />
  </section>
</sd7config>
```

The example shows an excerpt of the main configuration file. Options will reside always in the global section. Other sections are used for more specific configuration customizations, for example the OgreRenderer may load a specific list of plugins.
13.2. Project BootStrap Manifest

<!ELEMENT bootstrap (module)+>
<!ATTLIST bootstrap
default_target CDATA #REQUIRED>

<!ELEMENT module (comment?, license?, source*, patch?, build?, clean?, install?)>
<!ATTLIST module
name ID #REQUIRED
ignore CDATA #IMPLIED
depends CDATA #IMPLIED>

<!ELEMENT comment (#PCDATA)>
<!ELEMENT license (#PCDATA)>

<!ELEMENT source EMPTY>
<!ATTLIST source
addr CDATA #REQUIRED
method CDATA #REQUIRED
renamefrom CDATA #IMPLIED
md5 CDATA #IMPLIED
module CDATA #IMPLIED
revision CDATA #IMPLIED
platform CDATA #IMPLIED
arch CDATA #IMPLIED
branch CDATA #IMPLIED
python CDATA #IMPLIED>

<!ELEMENT patch EMPTY>
<!ATTLIST patch
addr CDATA #REQUIRED>

<!ELEMENT build (cmd)+>
<!ELEMENT clean (cmd)+>
<!ELEMENT install (cmd)+>

<!ELEMENT cmd (#PCDATA)>
<!ATTLIST cmd
cmd CDATA #IMPLIED
path CDATA #IMPLIED>
Integration of a 3D rendering engine with a physics simulator

The Boostrap manifest configuration file consists on a list of modules containing instructions of how to download and build each independent module. Each independent module has a set of dependencies. Each module will have:

- **Comment**: A small explanation about the module.
- **License**: The type of license of the module.
- **Source**: One or more paths and methods to download the module source or binary code.
- **Build**: The instructions required to automatically configure and build the module.
- **Clean**: The instructions required to delete the binary module.
- **Install**: The instructions required to install the module.

**Example:**

```xml
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE bootstrap SYSTEM "http://7d7.almlys.org/spec/draft/BootStrap.dtd">
<bootstrap default_target='sd7'>

  <module name='ois' ignore='no' depends='freeimage' >
    <comment>
      Object Oriented Input System
    </comment>
    <license>zlib/libpng</license>
    <source addr='http://7d7.almlys.org/downloads/depends/ois/ois_1.2.0.tar.gz'
      method='wget'
      md5='6a8cedad04f095127ca1455162fec955'/>
    <source addr='http://downloads.sourceforge.net/wgois/ois_1.2.0.tar.gz'
      method='wget'
      md5='6a8cedad04f095127ca1455162fec955'/>
    <build>
      <cmd>./bootstrap</cmd>
      <cmd>./configure --prefix=$PREFIX</cmd>
      <cmd>make -j 2</cmd>
    </build>

</module>
```

Page 109
This example shows the definition of a module named OIS. There are two locations where the module source code can be downloaded, and the corresponding configuration, build and install instructions are provided in the build and install sections.

13.3. **ActionMapper map file**

```xml
<!ELEMENT sd7input (input)+>

<!-- target: Name of a destination instance that will receive the event -->
<!-- priority: Input priority
  prior: 0 – Goes to the PriorityEvents list of the Action-Mapper
  if it will be not processed by the GUI if the HookManager processes it.
  prior: 1 – Goes to the normal Events list of the Action-Mapper -->

<!ELEMENT input (action?,axis?)>*
<!ATTLIST input
device CDATA #REQUIRED
target CDATA #IMPLIED
priority CDATA #IMPLIED>

<!ELEMENT action EMPTY>
<!ATTLIST action
name CDATA #REQUIRED
key CDATA #IMPLIED
mask CDATA #IMPLIED
target CDATA #IMPLIED
button CDATA #IMPLIED
priority CDATA #IMPLIED>

<!ELEMENT axis EMPTY>
<!ATTLIST axis
Integration of a 3D rendering engine with a physics simulator

The ActionMapper XML configuration file contains an association list of events generated by hardware devices with its corresponding actions. For example specific keys of the keyboard may be mapped to actions.

Each hardware device is represented by the *input* element. Each *input* element will have the following attributes:

- **device**: The name of the device to map. (keyboard, mouse, joy1, joy2...)
- **target**: The destination logic hook (python script) that will handle the input event. It's optional, if it's not provided the event will be sent to all.
- **priority**: The event priority. Events with priority <=0 are not send to the GUI.

Finally an *input* element will have one or more *action* and *axis* elements.

Any *action* has the following attributes:

- **name**: The name of the action that will be sent in the generated event.
- **key**: On keyboard devices the name of the keystroke. For example “a”, “ESC”, “UP”... Name definitions can be found in the sd7/engine/input/OISKeys.py file.
- **mask**: On keyboard devices a coma separated list, containing the desired status of the *ALT*, *CTRL* and *SHIFT* keys. The list may contain only one or more of the following names (“alt”, “ctrl”, “shift”) separated by coma.
- **target**: Overrides the target destination hook defined by the device that will process this event.
- **button**: On devices like the mouse, the joystick, gamepad... it defines the number of the button associated to the action. It will be always an integer value higher than zero.
● **priority**: Overrides the event priority defined by the device.

Any *axis* has the following attributes:

● **name**: The name of the action that will be sent in the generated event.

● **id**: On Mouse devices it will be either x, y, or z, on joysticks it will be an integer value higher than zero. This associates an specific axis to an event.

● **Target**: Overrides the target destination hook defined by the device that will process this event.

● **Priority**: Overrides the event priority defined by the device.

**Example**:

```xml
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE sd7input SYSTEM "http://7d7.almlys.org/spec/draft/sd7Input.dtd">
<sd7input>

  <input device='keyboard'>
    <!-- Exit action -->
    <action name='exit' key='ESC' priority='0' target='basicInput' />
    <action name='exit' key='c' mask="ctrl" priority='0' target='basicInput' />

    <action name='console' key='F11' priority='0' target="chatapp" />
    <action name='gui' key='tab' priority='0' target='basicInput' />

   <!-- Camera 1 -->
    <action name='up' key='w' target='Cam' />
    <action name='down' key='s' target='Cam' />
    <action name='left' key='a' target='Cam' />
    <action name='right' key='d' target='Cam' />
    <action name='up' key='up' target='Cam' />
    <action name='down' key='down' target='Cam' />
    <action name='rotate_left' key='left' target='Cam' />
    <action name='rotate_right' key='right' target='Cam' />

</input>
</sd7input>
```
Integration of a 3D rendering engine with a physics simulator

```xml
<action name='rotate_up' key='PgUp' target='Cam' />
<action name='rotate_down' key='PgDown' target='Cam' />
</input>

<input device='mouse' target='Cam' >
  <action name='up' button='1' />
  <action name='down' button='2' />
  <axis name='x' id='x' invert='false' />
  <axis name='y' id='y' invert='true' />
</input>

<input device='joy1' target='Cam' >
  <action name='up' button='1' />
  <action name='down' button='2' />
  <axis name='x' id='1' />
  <axis name='y' id='2' invert='true' />
  <axis name='z' id='3' />
  <axis name='speed' id='4' />
</input> </sd7input>

The example sets up the following associations:

- **ESC** and **CTRL+C** are associated to the “exit” action, that will be received by the `basicInput` logic hook.

- **F11** is associated to the “console” action, that will be received by the `chatapp` logic hook.

- **TAB** is associated to the “gui” action, that will be received by the `basicInput` logic hook.

- Then it associates the “w,a,s,d” and the cursor keys for the movement actions. These events will be processed by the `Cam` logic hook.

- Finally it also associates a Mouse device an a Joystick device to the movement actions. These actions will be also processed by the `Cam` logic hook.
13.4. **HookManager Configuration file**

```xml
<!ELEMENT sd7hooks (controller)+>
<!ELEMENT controller (param)*>
<!ATTLIST controller
    name CDATA #REQUIRED
    bind CDATA #REQUIRED
    autostart CDATA #IMPLIED>
<!ELEMENT param (#PCDATA)>
<!ATTLIST param
    name CDATA #REQUIRED>
```

The HookManager configuration file defines a set of logic controllers or sub-programs, that may be initialized.

Each controller will have the following attributes:

- **name**: An unique name for the controller.
- **bind**: The python class associated.
- **autostart**: Optional, it sets whenever the controller needs to be created on application startup.

In addition a list of parameters may be passed. Parameters are formed by key-value pairs.
14. Appendix E - Development environment

The following appendix contains a reference of applications and tools involved in the development process.

14.1. Source implementation

From the beginning the project source code has been implemented on normal code editors. Also specific python editors like Idle\(^{31}\) and SPE\(^ {32}\) has been used on the implementation of almost all code.

At the end NetBeans\(^ {33}\) with the Python plugin has been adopted for the development process.

In order to work with any of these editors and run the framework from the IDE, it is necessary to previously set the corresponding environment variables so Python can find the binary dependencies. The environment can be set by running

\(^{31}\) Idle: [http://docs.python.org/library/idle.html](http://docs.python.org/library/idle.html)

\(^{32}\) SPE: [http://pythonide.blogspot.com/](http://pythonide.blogspot.com/)

the “scripts/shell.sh” script.

In addition NetBeans Python path needs to be configured in the IDE so it can find the modules in the project path.

14.2. **Assets creation**

For the edition an creation of assets for the engine the following tools are being used:

- **Blender**: Blender is a FLOSS 3d graphics editor. 3D Graphical assets are done with it and exported into the ogre mesh format.

- **CELayout Editor** and **CEImageSet Editor**: These tools are part of the CEGUI library used, and aids in the creation of graphical users interfaces in a visual way. The output files, are layout xml files that can be loaded by the GUI.
15. Appendix F - Example applications

15.1. Basic GUI Example

```python
from sd7.engine.controller import Controller
from sd7.engine import Engine

# All application must derive from Controller
class ChatApp(Controller):

    _history = []
    _maxlines = 20
    _time = 0

    def __init__(self, params):
        Controller.__init__(self, params, "console")
        self._visible = True

    def initialize(self):
        Controller.initialize(self)

        # get the GUI and load the chat View
        self._gui = Engine().getGUI()
        self._gui.loadView("chat.layout")
        # Watch for input events
        self._gui.subscribeEvent("Chat/ChatWindow/ChatOkBtn",
                                 
                                 "PushButton/EventClicked", self.onInput)
        self._gui.subscribeEvent("Chat/ChatWindow/ChatInput",
                                 "Editbox/EventTextAccepted", self.onInput)
        self._gui.subscribeEvent("Chat/ChatWindow/ChatInput",
                                 "Editbox/EventActivated", self.onFocus)

        self._textBox = self._gui.getObject("Chat/ChatWindow/ChatInput")
        self._chatBox = self._gui.getObject("Chat/ChatWindow/ChatBox")

    def terminate(self):
        self._gui.destroyObject("Chat/ChatWindow")
        Controller.terminate(self)

    def setVisible(self, vis = None):
```

Page 117
if vis is not None:
    self._visible = vis
else:
    self._visible = not self._visible
self._gui.getObject("Chat/ChatWindow").setVisible(self._visible)
    
    if self._visible:
        bi = Engine().getHookMGR().findController("basicInput")
        if bi:
            bi.toggleGUIFocus(self._visible)
        self._textBox.activate()

def onFocus(self,e):
    self._textBox.setText("")

def processCommand(self, cmd):
    try:
        args = cmd.split()
        cmd = args.pop(0).lower()
        if cmd.lower() in ("quit", "exit", "logout", "killme"):  
            Engine().terminate()
        elif cmd == "hsize" and len(args)>0:
            self._maxlines = int(args[0])
        elif cmd == "help":  
            self.addMsg("Available commands are:")
            self.addMsg("quit, hsize")
            self.addMsg("Type help command for more information")
        elif cmd == "clear":  
            self.clear()
        elif cmd == "list":  
            ll = Engine().getHookMGR().list()
            map = { False: { True: "Running", False: "Stopped"},
                    True: { True: "Running*", False: "Stopped*"}}
            for l in ll.keys():
                a, b = ll[l]
                self.addMsg("%s - %s %s%ls%smap[a][b]

elif cmd == "start" and len(args)>0:
    Engine().getHookMGR().start(args[0])
    self.addMsg("%s started..." % (args[0]))
elif cmd == "stop" and len(args)>0:
    try:
        Engine().getHookMGR().stop(args[0])
        self.addMsg("%s stopped..." %args[0])
    except:
        pass
elif cmd == "restart" and len(args)>0:
    try:
        Engine().getHookMGR().restart(args[0])
        self.addMsg("%s restarted..." %args[0])
    except:
        pass
elif cmd == "bind" and len(args)>1:
    try:
        Engine().getHookMGR().bind(args[0],args[1])
    except:
        pass
else:
    self.addMsg("I'm sorry, I'm afraid I can't do that")

except Exception,e:
    self.addMsg("Exception: %s" %e,)

def clear(self):
    self._history = []
    self._chatBox.setText("")

def onInput(self,e):
    msg = self._textBox.getText()
    self._textBox.setText("")
    msg = str(msg).strip()
    if len(msg) == 0:
        return
    elif msg.startswith("/"):
        self.processCommand(msg[1:])
    else:
        self.addMsg("You: %s" %msg)

def addMsg(self, msg):
    self._history.append(msg)
    if len(self._history) > self._maxlines:
        self._history.pop(0)
    txt = "\n".join(self._history)
    self._chatBox.setText(txt)
    print "%s" %msg,
    self._chatBox.setCaratIndex(len(txt))
The example application, is just a simple text window that outputs written text to it and is capable of running some commands.

All framework applications must be derived form the Controller class, and may override if necessary the initialization, init and the event related methods.

On the `initialize` method, it load the “chat.layout” view, and requests to the GUI to receive event messages for the selected controls.

The `onInput` method will be called by the GUI when the user writes a message in the input text box and does click on the send button or the key enter was pressed. Then the message is parsed, if it starts with “/” then it is a command as is processed, on the contrary the message is just written in the bigger text control.

The example code, shows also how to request to the HookManager the creation and destruction of other applications.

### 15.2. Basic scene manipulation example

```python
from sd7.engine.controller import Controller
from sd7.engine import Engine

import random

class World(Controller):
    def __init__(self, params):
        Controller.__init__(self, params, "World")
        self._worldMGR = Engine().getWorldMGR()

    def initialize(self):
        Controller.initialize(self)
```
Integration of a 3D rendering engine with a physics simulator

```python
self.loadScene()
print "World was initialized"

def loadScene(self):
    self._worldMGR.setAmbientLight((0.25, 0.25, 0.25))
    #self._worldMGR.loadScene("test.xml")

    #World ground
    self._worldMGR.createPlane("ground", (0, 1, 0), -10)
    self._worldMGR.createBall("ball01", (10, 100, 0), 20, 10)
    self._worldMGR.createBall("ball02", (40, 120, 0), 10, 100)
    self._worldMGR.createBall("ball03", (80, 220, 10), 25, 150)
    self._worldMGR.createBall("ball04", (120, 400, -20), 5, 12)

    for i in range(200):
        self._worldMGR.createBall("random%i" % (i),
                                (-20 * random.uniform(0, 0.1), i*42+150, 20 * random.uniform(0, 0.1)),
                                15, 50)
        self._worldMGR.createBall("randoma%i" % (i),
                                (-180 * random.uniform(0, 0.1), i*40*random.uniform(1, 5)+150, 120 * random.uniform(0, 0.1)),
                                5, 50)
```

This simple example shows how to interact with the World Manager. It creates a simple plane base, and then it creates a set of balls of different sizes. The object creation methods always expect the name of the object and its properties.

For example, the first ball created is named “ball01”, it's positioned at the (10, 100, 0) coordinate, it defines a radius of 20 meters and sets 10 Kg of mass.

Then 400 more balls are created in the sky at different positions, it mainly creates two columns of balls of two sizes: 5 and 15 meters of radius.

When the application is run, it will display a lot of balls falling from the sky and colliding between them and with the ground.
16. Bibliography and references


[ope09] OpenAL. *OpenAL*. Available at: [http://openal.org](http://openal.org) (Accessed:
01/06/2009)


17. License

This work is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/3.0/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Note: The source code is licensed under the GNU GPL.