3D SCENE EDITOR USING OGRE3D

REPORT

Author: Héctor Esbri Rodríguez-Xuárez
Director: Professor Dr. Borut Zalik
(FERI – Univerza v Mariboru)
(FIB – Universitat Politècnica de Barcelona)
June 2007
SUMMARY

The aim of the thesis is to development of an application which allows creating tridimensional virtual scenes.

To achieve this goal, it has been used the open sourced engine Ogre3D, which provides a useful transparency layer above the lowest level details of the rendering process.

By means of a highly user-friendly GUI (Graphical User Interface), it is possible to define all the parameters concerning the scene: sky, terrain and objects in the scene. All these parameters have a high degree of customization, making it possible to design a huge variety of very different scenarios. It also provides different approaches and solutions, so the user can choose the most suitable option for his scene.

The application is highly script-based, so any media addition (such as new materials, textures or meshes) to its filesystem is immediately incorporated to the application recourses. As the objective of the thesis was focused in the development of the application itself, just a small set of external media resources is provided, to offer a practical vision of the application usage.

The present document is intended to describe how all this features have been implemented in the application, and to present all the information used to do so. It should not be seen as a user's guide nor manual, although its reading should provide a full understanding of the application correct use.

It is structured in two main bodies. The first of them is dedicated to explain all the technical details related with the engine used by the editor. The second explains in detail the design of the application itself, and points out how does it match with Ogre’s logic.
CONTENTS

SUMMARY ................................................................................................................. 2

CONTENTS ................................................................................................................. 4

1. INTRODUCTION .................................................................................................. 6
   1.1 OBJECTIVES ................................................................................................. 6
   1.2 REACH ............................................................................................................ 6

2. OGRE .................................................................................................................... 7
   2.1 HIGH LEVEL OVERVIEW ............................................................................. 7
   2.2 THE ROOT OBJECT ...................................................................................... 11
   2.3 THE RENDER SYSTEM OBJECT .................................................................... 12
      2.3.1 FRAMELISTENERS .............................................................................. 13
   2.4 THE SCENE MANAGER OBJECT .................................................................. 16
      2.4.1 SCENENODE ....................................................................................... 17
   2.5 THE RENDER CYCLE .................................................................................... 18
   2.6 THE RESOURCE GROUP MANAGER OBJECT ............................................. 19
   2.7 ENTITIES ...................................................................................................... 20
   2.8 MATERIALS ................................................................................................. 22
   2.9 MATERIAL SCRIPTS ..................................................................................... 24
      1.9.1 LOADING SCRIPTS .............................................................................. 24
      1.9.2 FORMAT ............................................................................................... 25
   2.10 SKELETAL ANIMATION .............................................................................. 27
   2.11 ANIMATION STATE ..................................................................................... 28

3. EDITOR .................................................................................................................. 29
   3.1 OVERVIEW .................................................................................................... 29
   3.2 SCENENODES HIERARCHY ......................................................................... 30
   3.3 SKY ................................................................................................................ 31
      3.3.1 SKYBOX ................................................................................................. 32
      3.3.2 SKYDOME ............................................................................................. 33
      3.3.4 SKYPLANE ............................................................................................. 34
   3.4 TERRAIN ....................................................................................................... 36
      3.4.1 TERRAIN SCENE MANAGER .............................................................. 36
      3.4.1.1 CONFIGURATION PARAMETERS .................................................. 37
   3.5 MESHES (OBJECTS) ..................................................................................... 40
4. INTERFACE LAYER ................................................................. 41
   4.1 MAINAPP ................................................................. 41
       4.1.1 APPLICATION EXECUTION PROCESS ......................... 42

5. CONTROL LAYER ............................................................... 43
   5.1 MAINFRAMELISTENER ................................................... 43
       5.1.1 PREVIEW WINDOW, RENDER TO TEXTURE ................. 43
       5.1.2 PROCESS UNBUFFERED KEY INPUT ......................... 44
       5.1.3 FRAME ROUTINE ............................................... 45
       5.1.4 PROCESS MOUSE CLICK ...................................... 48
   5.2 MESHVIEWPORTLISTENER ............................................. 50

6. DOMAIN LAYER ............................................................... 51
   6.1 MESHOBJECT ............................................................ 51
       6.1.2 DYNAMIC MESHOBJECTS ...................................... 52
   6.2 SCENECONFIG ............................................................ 53

7. GRATITUDES ................................................................. 54

8. BIBLIOGRAPHY ............................................................... 56
1. INTRODUCTION

1.1 OBJECTIVES

The main objective of this thesis is to provide a extendible tool for generating 3D scenes. The application by itself could be used for designing and exploring those scenes, but it has been aimed to be easily extended to fulfill any other desired visualization usage. It could be done extending the editor itself or using the stored scene data generated with it to feed another visualization application.

The application logic is divided following the design principle of the three layers: Interface, Control and Domain. Any of them could be changed or improved requiring minimal derived changes in the other layers.

It is also intended to provide a suitable and user-friendly interface, as clear and clean as possible, so its design has had a considerable effort dedicated.

1.2 REACH

The editor allows the user to define an scene by means of a sky, a terrain and objects, both static and dynamic. There are other features, such as particle effects, waypoints or dynamic lights which have not been implemented, as they would fit in the scope of a much larger application. Anyway, all these features could be consistently included in the editor in an hypotetical extension without significant design changes.

The aim of the thesis is the development of the application itself, so no media data has been developed for the current version. A small set of external media resources is provided, to offer a practical vision of the application usage.
2. OGRE

2.1 High Level Overview

Ogre is an open-source graphics rendering engine (Object-oriented Graphics Rendering Engine) that is written and maintained by a small core team, and contributed to by its ever growing community.

Fig. 1.1 Ogre Class Diagram
The class diagram above shows the Root class of OGRE and all of the Manager classes that gives access to the different subsystems.

- The Root holds a reference to the current SceneManager and to an enumerator that allows other types of scene graphs to be loaded.

- The RenderSystem is an abstract class that separates the Root from the specific implementation of the RenderSystem, OpenGL or Direct3D.

- The ResourceManager is an abstract class that is subclassed by a number of managers that handles resources such as textures, materials and fonts.

All of the classes in the diagram except for RenderSystem, SceneManager and RenderWindow are using a singleton template to ensure that there is only one instance of each class.

Shown below is a diagram of some of the core objects of the Ogre engine. This is not all the classes by a long shot, just a few examples of the more significant ones to give an idea of how it slots together.
At the very top of the diagram is the Root object. This is the 'way in' to the OGRE system, and it's where the top-level objects of applications are created, like scene managers, rendering systems and render windows or loading plugins. Root is it for almost everything, although often it will just give another object which will actually do the detail work, since Root itself is more of an organizer and facilitator object.
The majority of the rest of OGRE's classes fall into one of 3 roles:

**Scene Management**

This is about the contents of a scene, how it's structured, how it's viewed from cameras, etc. Objects in this area are responsible for giving a natural declarative interface to the world it is being built.

**Resource Management**

All rendering needs resources, whether it's geometry, textures, and fonts, whatever. It's important to manage the loading, re-use and unloading of these things carefully, so that's what classes in this area do.

**Rendering**

Finally, there's getting the visuals on the screen - this is about the lower-level end of the rendering pipeline, the specific rendering system API objects like buffers, render states and the like and pushing it all down the pipeline. Classes in the Scene Management subsystem use this to get their higher-level scene information onto the screen.

OGRE is designed to be extended, and plugins are the usual way to go about it. Many of the classes in OGRE can be subclassed and extended, whether it's changing the scene organization through a custom SceneManager, adding a new render system implementation (e.g. Direct3D or OpenGL), or providing a way to load resources from another source (say from a web location or a database).
2.2 The Root Object

The 'Root' object is the entry point to the OGRE system. This object MUST be the first one to be created, and the last one to be destroyed.

The root object lets configure the system, for example through the showConfigDialog() method which is an extremely handy method which performs all render system options detection and shows a dialog for the user to customize resolution, color depth, full screen options etc. It also sets the options the user selects so that the system can be initialized directly afterwards.

The root object is also the method for obtaining pointers to other objects in the system, such as theSceneManager, RenderSystem and various other resource managers.

Finally, if OGRE is running in continuous rendering mode, it’s important to always refresh all the rendering targets as fast as possible. The root object has a method called startRendering, which when called will enter a continuous rendering loop which will only end when all rendering windows are closed, or any FrameListener objects indicate that they want to stop the cycle (see below for details of FrameListener objects).
2.3 The RenderSystem Object

The RenderSystem object is actually an abstract class which defines the interface to the underlying 3D API. It is responsible for sending rendering operations to the API and setting all the various rendering options. This class is abstract because all the implementation is rendering API specific - there are API-specific subclasses for each rendering API (e.g. D3DRenderSystem for Direct3D). After the system has been initialized through Root::initialise, the RenderSystem object for the selected rendering API is available via the Root::getRenderSystem() method.

However, a typical application should not normally need to manipulate the RenderSystem object directly - everything it is needed for rendering objects and customizing settings should be available on the SceneManager, Material and other scene-oriented classes. It's only if it is wanted to create multiple rendering windows (completely separate windows in this case, not multiple viewports like a split-screen effect which is done via the RenderWindow class) or access other advanced features that it is needed access to the RenderSystem object.
2.3.1 FrameListeners

In Ogre, it is possible to register a class to receive notification before and after a frame is rendered to the screen. This FrameListener interface defines two functions:

```cpp
bool frameStarted(const FrameEvent& evt)
bool frameEnded(const FrameEvent& evt)
```

Ogre's main loop (Root::startRendering) looks like this:

1. The Root object calls the frameStarted method on all registered FrameListeners.
2. The Root object renders one frame.
3. The Root object calls the frameEnded method on all registered FrameListeners.

This loops until any of the FrameListeners return false from frameStarted or frameEnded. The return values for these functions basically mean "keep rendering". If either of them return false, the program will exit. The FrameEvent object contains two variables, but only the timeSinceLastFrame is useful in a FrameListener. This variable keeps track of how long it's been since the frameStarted or frameEnded last fired.

Note that in the frameStarted method, FrameEvent::timeSinceLastFrame will contain how long it has been since the last frameStarted event was last fired (not the last time a frameEnded method was fired).
One important concept to realize about Ogre's FrameListeners is that the order in which they are called is entirely up to Ogre. It is not possible to determine which FrameListener is called first, second, third...and so on. If it is needed to ensure that FrameListeners are called in a certain order, then it should be registered only one FrameListener and have it call all of the objects in the proper order.

It might be also noticed that the main loop really only does three things, and since nothing happens in between the frameEnded and frameStarted methods being called, they can be used almost interchangeably.
2.3.1.1 Registering a FrameListener

Since the Root class is what renders frames, it also is in charge of keeping track of FrameListeners. The first thing it is needed to do is create an instance of the FrameListener and register it with the Root object.

```c++
// Create the FrameListener
mFrameListener = new FrameListener(mWindow,
mCamera, mSceneMgr);
mRoot->addFrameListener(mFrameListener);
```

The “addFrameListener” method adds a FrameListener, and the “removeFrameListener” method removes a FrameListener (that is, the FrameListener will no longer receive updates). Note that the add|removeFrameListener methods only take in a pointer to a FrameListener (that is, FrameListeners do not have names which can be used to remove them). This means that it will be needed to hold a pointer to each FrameListener it is created so that it can later be removed them.
2.4 The SceneManager Object

Apart from the Root object, this is probably the most critical part of the system from the application's point of view. Certainly it will be the object which is most used by the application. The SceneManager is in charge of the contents of the scene which are to be rendered by the engine. It is responsible for organizing the contents using whatever technique it deems best, for creating and managing all the SceneNodes (see SceneNode section) cameras, movable objects (see Entities section), lights and materials (surface properties of objects; see Materials section), and for managing the 'world geometry' which is the sprawling static geometry usually used to represent the immovable parts of a scene. There is no need for the application to keep lists of objects, the SceneManager keeps a named set of all of the scene objects for easy access.

The SceneManager also sends the scene to the RenderSystem object when it is time to render the scene. The SceneManager::_renderScene method is called automatically whenever a rendering target is asked to update.

So most of the interaction with the SceneManager takes place during scene setup. It can also modify the contents of the scene dynamically during the rendering cycle, using a FrameListener object.

Because different scene types require very different algorithmic approaches to deciding which objects get sent to the RenderSystem in order to attain good rendering performance, the SceneManager class is designed to be subclassed for different scene types. The default SceneManager object will render a scene, but it does little or no scene organization and it should not be expected the results to be high performance in the case of large scenes. The intention is that specializations will be created for each type of scene such that under the surface the subclass will optimize the scene organization for best performance given assumptions which can be made for that scene type.

In the case of this particular application, the SceneManager used is the Terrain Scene Manager (see Terrain Scene Manager section).
2.4.1 **SceneNode**

SceneNodes keep track of location and orientation for all of the objects attached to it. When an Entity is created, it is not ever rendered on the scene until it is attached to a SceneNode. Similarly, a SceneNode is not an object that is displayed on the screen. Only when a SceneNodes is created and attached an Entity (or other object) to it is something actually displayed on the screen.

SceneNodes can have any number of objects attached to them. SceneNodes may also be attached to other SceneNodes which allows creating entire hierarchies of nodes.

One major concept to note about SceneNodes is that a SceneNode's position is *always* relative to its parent SceneNode, and each SceneManager contains a root node to which all other SceneNodes are attached.
2.5 THE RENDER CYCLE

Fig. 1.3 The Render Cycle Sequence Diagram

All the elements of the rendering cycle have been defined now, so the entire process can be described in detail.

For each frame, the application asks the entrance point of Ogre, Root, to render a new frame. Root propagates the call to the RenderSystem object, which updates every possible renderTarget defined in the application.

Every renderTarget object updates the rendering viewport. This is done by means of the camera, which asks the SceneManager to take control of all the renderable objects declared, obtaining them from the render queues.

Finally, every renderable object applies the rendering operations declared, and is actually shown in the viewport.
2.6 The ResourceGroupManager Object

The ResourceGroupManager class is actually a 'hub' for loading of reusable resources like textures and meshes. It is the place that it can be defined groups for different resources, so they may be unloaded and reloaded when wanted. Servicing it are a number of ResourceManagers which manage the individual types of resource, like TextureManager or MeshManager. In this context, resources are sets of data which must be loaded from somewhere to provide OGRE with the data it needs.

ResourceManagers ensure that resources are only loaded once and shared throughout the OGRE engine. They also manage the memory requirements of the resources they look after. They can also search in a number of locations for the resources they need, including multiple search paths and compressed archives (ZIP files).

To tell the resource managers where to look for resources is done via Root::getSingleton().addResourceLocation, which actually passes the information on to ResourceGroupManager.

Because there is only ever 1 instance of each resource manager in the engine, if it is wanted to get a reference to a resource manager the following syntax must be used:

```
TextureManager::getSingleton().someMethod()
MeshManager::getSingleton().someMethod()
```
2.7 Entities

An entity is an instance of a movable object in the scene. The only assumption is that it does not necessarily have a fixed position in the world.

Entities are based on discrete meshes, i.e. collections of geometry which are self-contained and typically fairly small on a world scale, which are represented by the Mesh object. Multiple entities can be based on the same mesh, since often it is wanted to create multiple copies of the same type of object in a scene.

An entity can be created by calling the SceneManager::createEntity method, giving it a name and specifying the name of the mesh object which it will be based on. The SceneManager will ensure that the mesh is loaded by calling the MeshManager resource manager. Only one copy of the Mesh will be loaded.

Entities are not deemed to be a part of the scene until they are attached them to a SceneNode. By attaching entities to SceneNodes, complex hierarchical relationships can be created between the positions and orientations of entities. Modify the positions of the nodes indirectly affects the entity positions.

When a Mesh is loaded, it automatically comes with a number of materials defined. It is possible to have more than one material attached to a mesh - different parts of the mesh may use different materials. Any entity created from the mesh will automatically use the default materials.

All Mesh objects are actually composed of SubMesh objects, each of which represents a part of the mesh using one Material. If a Mesh uses only one Material, it will only have one SubMesh.

When an Entity is created based on this Mesh, it is composed of (possibly) multiple SubEntity objects, each matching 1 for 1 with the SubMesh
objects from the original Mesh. The SubEntity objects can be accessed using the Entity::getSubEntity method. With that reference to a SubEntity, the material it uses can be changed by calling its setMaterialName method. In this way an Entity can be deviated from the default materials and thus create an individual looking version of it.
2.8 MATERIALS

The Material object controls how objects in the scene are rendered. It specifies what basic surface properties objects have such as reflectance of colors, shininess etc, how many texture layers are present, what images are on them and how they are blended together, what special effects are applied such as environment mapping, what culling mode is used, how the textures are filtered etc.

Materials can either be set up programmatically, by calling SceneManager::createMaterial and tweaking the settings, or by specifying it in a 'script' which is loaded at runtime (section Material Scripts).

Basically everything about the appearance of an object apart from its shape is controlled by the Material class.

The SceneManager class manages the master list of materials available to the scene. The list can be added to by the application by calling SceneManager::createMaterial, or by loading a Mesh (which will in turn load material properties). Whenever materials are added to the SceneManager, they start off with a default set of properties; these are defined by OGRE as the following:

- ambient reflectance = ColourValue::White (full)
- diffuse reflectance = ColourValue::White (full)
- specular reflectance = ColourValue::Black (none)
- emissive = ColourValue::Black (none)
- shininess = 0 (not shiny)
- No texture layers (& hence no textures)
- SourceBlendFactor = SBF_ONE, DestBlendFactor = SBF_ZERO (opaque)
- Depth buffer checking on
- Depth buffer writing on
- Depth buffer comparison function = CMPF_LESS_EQUAL
- Culling mode = CULL_CLOCKWISE
- Ambient lighting in scene = ColourValue(0.5, 0.5, 0.5) (mid-grey)
- Dynamic lighting enabled
- Gourad shading mode
- Solid polygon mode
- Bilinear texture filtering

These settings can be altered by calling `SceneManager::getDefaultMaterialSettings()` and making the required changes to the Material which is returned. Entities automatically have Material's associated with them if they use a Mesh object, since the Mesh object typically sets up it's required materials on loading.
2.9 Material Scripts

Material scripts offer the ability to define complex materials in a script which can be reused easily. Whilst it can be set up all materials for a scene in code using the methods of the Material and TextureLayer classes, in practice it's a bit unwieldy. Instead, material definitions can be stored in text files which can then be loaded whenever required.

1.9.1 Loading Scripts

Material scripts are loaded when resource groups are initialized: OGRE looks in all resource locations associated with the group for files with the '.material' extension and parses them. The files can be parsed manually, using MaterialSerializer::parseScript.

It's important to realize that materials are not loaded completely by this parsing process: only the definition is loaded, no textures or other resources are loaded. This is because it is common to have a large library of materials, but only use a relatively small subset of them in any one scene. To load every material completely in every script would therefore cause unnecessary memory overhead. A 'deferred load' Material can be accessed in the normal way (MaterialManager::getSingleton().getByName()), but it must be called the 'load' method before trying to use it. Ogre does this when using the normal material assignment methods of entities, etc.

Another important factor is that material names must be unique throughout all scripts loaded by the system, since materials are always identified by name.
1.9.2 Format

Several materials may be defined in a single script. The script format is pseudo-C++, with sections delimited by curly braces ('{', '}'), and comments indicated by starting a line with '//'. The general format is shown in the example below:

```plaintext
// This is a comment
material walls/wall1
{
    // first, preferred technique
technique
    {
        // first pass
        pass
        {
            ambient 0.5 0.5 0.5
diffuse 1.0 1.0 1.0
        }
        // Texture unit 0
texture_unit
        {
            texture wibbly.jpg
            scroll_anime 0.1 0.0
            wave_xform scale sine 0.0 0.7 0.0 1.0
        }
        // Texture unit 1 (this is a multitexture pass)
texture_unit
        {
            texture wobbly.png
            rotate_anime 0.25
            colour_op add
        }
    }
}

// Second technique, can be used as a fallback or LOD level
technique
{
    // .. and so on
}
```
Every material in the script must be given a name, which is the line 'material <name>:' before the first opening '{'. This name must be globally unique. It can include path characters (as in the example) to logically divide up the application materials, and also to avoid duplicate names, but the engine does not treat the name as hierarchical, just as a string.

A material can copy from a previously defined material by using a colon : after the material name followed by the name of the reference material to copy. If the reference material can not be found then it is ignored.

A material can be made up of many techniques. A technique is one way of achieving different visual effects. More than one technique can be supplied in order to provide fallback approaches where a card does not have the ability to render the preferred technique, or to define lower level of detail versions of the material in order to conserve rendering power when objects are more distant.

Each technique can be made up of many passes, that is a complete render of the object can be performed multiple times with different settings in order to produce composite effects. Ogre may also split the passes defined into many passes at runtime, if a pass uses too many texture units for the card the application is currently running on. Each pass has a number of top-level attributes such as 'ambient' to set the amount & color of the ambient light reflected by the material. Some of these options do not apply if vertex programs are being used.

Within each pass, there can be zero or many texture units in use. These define the texture to be used, and optionally some blending operations (which use multitexturing) and texture effects.

Vertex and fragment programs can be referenced, too, in a pass with a given set of parameters. Programs themselves are declared in separate .program scripts.
2.10 Skeletal Animation

Skeletal animation is a process of animating a mesh by moving a set of hierarchical bones within the mesh, which in turn moves the vertices of the model according to the bone assignments stored in each vertex. An alternative term for this approach is 'skinning'. The usual way of creating these animations is with a modeling tool such as Softimage XSI, Milkshape 3D, Blender, 3D Studio or Maya among others. OGRE provides exporters to allow the user to get the data out of these modelers and into the engine.

There are many grades of skeletal animation. OGRE supports the following features:

- Each mesh can be linked to a single skeleton
- Unlimited bones per skeleton
- Hierarchical forward-kinematics on bones
- Multiple named animations per skeleton (e.g. 'Walk', 'Run', 'Jump', 'Shoot' etc)
- Unlimited key frames per animation
- Linear or spline-based interpolation between key frames
- A vertex can be assigned to multiple bones and assigned weightings for smoother skinning
- Multiple animations can be applied to a mesh at the same time, again with a blend weighting

Skeletons and the animations which go with them are held in .skeleton files, which are produced by the OGRE exporters. These files are loaded automatically when an entity is created based on a Mesh which is linked to the skeleton in question. The Animation State is used to set the use of animation on the entity in question (section Animation State). Skeletal animation can be performed in software, or implemented in shaders (hardware skinning). Clearly the latter is preferable, since it takes some of the work away from the CPU and gives it to the graphics card, and also means that the vertex data does not need to be re-uploaded every frame. This is especially important for large, detailed models.
2.11 Animation State

When an entity containing animation of any type is created, it is given an 'animation state' object per animation to allow the the animation state specification of that single entity.

A pointer to the AnimationState object can be retrieved by calling Entity::getAnimationState. Then, it is possible to call methods on this returned object to update the animation, probably in the frameStarted event. Each AnimationState needs to be enabled using the setEnabled method before the animation it refers to will take effect, and both the weight and the time position can be set (where appropriate) to affect the application of the animation using correlating methods. AnimationState also has a very simple method 'addTime' which allows altering the animation position incrementally, and it will automatically loop. addTime can take positive or negative values, so the animation can be easily reversed.
3. EDITOR

3.1 OVERVIEW

The Editor is designed following the design principle of the three layers: Interface, Control and Domain.

- Interface or Presentation. The presentation layer provides the application's user interface (UI), and handles the input of the user and the presentation of the generated output.

- Control or Business. The control layer implements the business functionality of the application. That is, the logic which rules the handling of all the data available in the domain layer.

- Domain or Data. The domain data holds all the persistent information about the entities which take place during the business processes of the application.

The Editor Interface Layer is implemented in the MainApp class. The Editor Control Layer is implemented in the MainFrameListener & MeshViewportListener classes. The Editor Domain Layer is implemented in the SceneConfig & MeshObject classes.
3.2 SceneNodes Hierarchy

![SceneNode Hierarchy Diagram]

Fig. 2.1 The Editor SceneNode Hierarchy

All the meshes which are displayed in the editor are attached to an entity, and belong to their own sceneNode hierarchy. These hierarchies are based upon the same scheme.

The world_node and the dialog_main_node are the only children of the Root sceneNode. All the other nodes, which will contain the attached entities, have world_node as their parent.

Every entity is attached to a mesh_node which is displaced to the center of the mesh respect its parent, called main_node. This main_node is attached to the world_node, and all the geometrical transformation are applied to it. This is done to achieve geometrical transformations based in a reference system centered in the geometrical center of the mesh, for those meshes which have a modeling center defined somewhere else.
3.3 **Sky**

The Editor allows providing the scene with three different types of sky. The material script to use for the Sky is parsed from the material scripts available in the `materials/` directory, and listed for the user's choice.

- **SkyBox**

- **SkyDome**

- **SkyPlane**
3.3.1 SkyBox

Fig. 2.2 SkyBox Properties Window

A set of 6 images that form a cube, which is rendered behind everything else, so the viewer is always "in" the cube.

Not all of the materials are supported by a SkyBox, as it needs 6 images to be rendered properly.

Another parameter allows setting the distance that the SkyBox is away from the Camera. The distance shouldn’t be set to be closer than the near clip distance on the Camera or the skybox will not be shown.

The last one sets whether or not the SkyBox is drawn before the rest of the scene or afterwards. It is not actually desirable to draw the SkyBox first, because the full of it is rendered. When it is drawn last, only the visible portions are drawn, which will provide a modest speed improvement. If the SkyBox is set to be too close, it could be cutting part of the scene geometry off.
3.3.2 SkyDome

Fig. 2.3 SkyDome Properties Window

A curved dome, which is then textured.

The first two parameters of the skyDome are the same as the skyBox. The third parameter is the curvature used for the SkyDome. The API reference suggests using values between 2 and 65; lower for better distance effect, but higher values for less distortion and a smoother effect.

The fourth parameter is the number of times the texture is tiled, which will be needed to tweak depending on the size of the texture.

The fifth and sixth parameters are distance and drawFirst, respectively, which have already been covered in the SkyBox section.
3.3.4 **SKYPLANE**

![SkyPlane Properties Window](image)

**Fig. 2.4 SkyPlane Properties Window**

SkyPlanes are very different from SkyBoxes and SkyDomes. Instead of a cube to render the sky texture on, a flat plane, which is then textured, is used to represent the sky.

The skyPlane does not have a distance parameter like SkyBox and SkyPlane. Instead that parameter is set in the d variable of the generating plane.

The fourth parameter is the size of the SkyPlane and the fifth parameter is how many times to tile the texture.
The primary problem with this technique is that if the user looks towards the horizon, he can see where the SkyPlane ends. This basic use of a SkyPlane is really only useful when the scene has high walls (or hills) all around the viewpoint. Using a SkyPlane in that situation would be considerably less graphics intensive than creating a full SkyBox/SkyDome.

Fortunately, that is not all what can be done with a SkyPlane. The sixth parameter to the skyPlane is the familiar "renderFirst" parameter which has already been covered in the SkyBox and SkyDome sections.

The seventh parameter allows specifying the curvature of the SkyPlane, so that a plane is used no longer, but a curved surface instead. It is needed now to set the number of x and y segments used to create the SkyPlane (initially the SkyPlane was one big square, but to have curvature it is necessary to have the plane made up of smaller squares). The eighth and ninth parameters to the function are the number of x and y segments, respectively
3.4 TERRAIN

The editor uses a specific scene manager to handle large outdoor terrains. This scene manager is called Terrain Scene Manager. Its principal feature is to provide the rendering of a terrain parsing a script.

3.4.1 TERRAIN SCENE MANAGER

The shape and coloration of the terrain in the scenes of the application scenes is computed from a **heightmap**, a **terrain texture** and a **detail texture**. The heightmap is a simple greyscale image in which each pixel represents a height value, where 0 is ground level and 255 represents the highest point in the terrain.

The terrain is textured with an image that is stretched over the terrain. These **terrain textures** are typically mottled brown, green, white or grey for earth, grass, snow or rock terrain. Since terrain textures are normally much smaller than the terrain being covered, terrain normally appears blurred when viewed close up. To help with this, a **detail texture** is used, which is blended with the terrain texture when viewed from close distances. The detail texture is not stretched as much as the terrain texture, removing the blurriness of terrains viewed close up.

Terrains are divided into a grid of **terrain tiles**. Terrain tiles may be displayed by the scene manager at a differing level of detail, depending on the topology of the tile and the distance from the viewer.

The entire terrain is projected within a world coordinate space which specifies the dimensions of the terrain in 3-space.
3.4.1.1 Configuration Parameters

The Terrain Scene Manager is configured via the `terrain.cfg` script.

**Basic Configuration Parameters**

The basic parameters from this script are:

- **WorldTexture**: Specifies the name of the terrain texture.
- **DetailTexture**: Specifies the name of the detail texture.
- **DetailTile**: This specifies the number of times that the detail texture will be repeated in each terrain tile. If this number is too low, the terrain may appear blurry at close distances. If this number is too high, the terrain may appear to have repeating patterns when viewed over a distance.
- **PageSource**: Specifies the source of the heightmap.
- **Heightmap.image**: The image file from which the heightmap is drawn. Thus must be a square where each dimension is of size $2^n+1$, for some integer $n$. The larger the size of the heightmap, the more detail the terrain will have, but the more resources the application will consume (including longer application start time.)

- **PageSize**: The terrain will be $PageSize \times PageSize$ vertices large. $PageSize$ must have the same value as the dimension of the heightmap image, and so must also have a value $2^n+1$ for some integer $n$.

- **TileSize**: Terrain tiles have the dimension $TileSize \times TileSize$ vertices. *This number must be smaller than PageSize.* $TileSize$ must have a value $2^n+1$ for some integer $n$. Making tile size too small impacts performance significantly while making tiles too large may result in unnecessarily high detail in some parts of the scene.

- **MaxPixelError**: This specifies the maximum error tolerated when determining which level of detail to use. Setting this value too high can result in seams in the terrain. Setting the value too low can impact performance.

- **PageWorldX, PageWorldZ**: This sets the extents of the terrain in world coordinates. The larger the terrain, the lower the detail will be in the terrain, as the number of vertices used for the terrain is based on the heightmap, not the world size. This can be used to scale the terrain to any size it is wanted.

- **MaxHeight**: The maximum height of the terrain in world coordinates. The 0..255 range from the heightmap is scaled to 0..MaxHeight in world coordinates.

- **MaxMipmapLevel**: Specifies the number of levels of detail that will be used in rendering the terrain. Terrain that is distant or is relatively flat can be rendered with less detail.
ADVANCED CONFIGURATION PARAMETERS

The following parameters control other aspects of the scene manager's operation.

- **VertexNormals**: This makes the TerrainSceneManager calculate and set vertex normals in the hardware buffer. If it is being used lighting or a GPU program that requires it, it should be turned on.

- **VertexColours**: This makes the TerrainSceneManager calculate and set the vertex colors in the hardware buffer.

- **UseTriStrips**: When set to yes, optimizes the order in which terrain triangles are sent to the GPU so that fewer vertices are sent.

- **VertexProgramMorph**: Use vertex program to morph LODs, if available.

- **LODMorphStart**: The proportional distance range at which the LOD morph starts to take effect. This is as a proportion of the distance between the current LODs effective range, and the effective range of the next lower LOD.
3.5 **Meshes (Objects)**

![Image of Mesh Preview Window](image)

**Fig. 2.4 Mesh Preview Window**

A Mesh object represents a discrete model, a set of geometry which is self-contained and is typically fairly small on a world scale. Mesh objects are assumed to represent movable objects and are not used for the sprawling level geometry typically used to create backgrounds.

Mesh objects are a type of resource, and are managed by the MeshManager resource manager. They are typically loaded from OGRE's custom object format, the '.mesh' format. Mesh files are typically created by exporting from a modeling.

Mesh objects are the basis for the individual movable objects in the world, which are called Entities.

Mesh objects can also be animated using skeletal animation.

The meshes and skeletons of the editor are loaded from the models/directory and listed for user's choice.
4. INTERFACE LAYER

4.1 MainApp

The MainApp class implements the Presentation Layer of the Editor. As the main tool, it has been used the CEGUI, a free library providing windowing and widgets for graphics APIs / engines where such functionality is not natively available, or severely lacking. The library is object orientated.

This class offers a transparency layer between the user’s input and the domain data handling. It also checks the data provided by the user to make sure it is consistent with the application logic.

For achieving this, the interface does not collect its own inputs, as they are injected under the editor control. As event notification is a vital aspect of GUI programming, the editor handles those using subscribers. In order to subscribe a window for an event, it is necessary to call the method 'Window::subscribeEvent', passing the function to be called when the specified event occurs, and the instance on which the method is called.
4.1.1 Application Execution Process

The MainApp first initializes the viewport of the editor, and the user’s camera, with the methods `createViewports(void)` and `createCamera(void)`.

Once the viewport is fixed and the camera set up, it is time to create the basic, empty scene which will be shown at the beginning of the execution. This includes:

- Initialize the gui layer, with the proper scheme and layout.
- Set the ambient light.
- Set the main light.
- Initialize the mouse cursor.
- Create the basic hierarchy of sceneNodes. Root, world_node and dialog_node.
- Setup the render to texture image for mesh preview window, including the preview window camera and viewport.
- Setup the widget event handlers.

MainApp is also in charge to create and register the mainFrameListener, and to select the Terrain Scene Manager as the application scene manager.
5. CONTROL LAYER

5.1 MAINFRAMELISTENER

The MainFrameListener has the control of every rendered frame of the application. Also, it is the one in charge to initialize the viewport and camera of the editor, as well as the OGRE scene, and the global lighting.

5.1.1 PREVIEW WINDOW. RENDER TO TEXTURE

The Mesh Dialog Window contains a preview window of the mesh that will be displayed with the next mouse click in the terrain.

The sub-scene of the mesh dialog window contains its own sceneNodes hierarchy, composed of the main_dialog_node and the mesh_dialog_node. The first is attached to the world_node and is the parent of the second. The center translation technique is applied to the mesh_dialog_node in order to achieve geometrical center based transformations.

The preview window is implemented using a RTT. With the Render To Texture technique the rendered pixels are blitted to a texture, instead of the screen. This texture is then attached to a StaticImage widget, which, after the proper setting up of the camera, shows the desired preview of the mesh.
5.1.2 **Process Unbuffered Key Input**

As the key input of the editor is unbuffered there is a `inputDevice` which captures the last input received between two consecutive frames, so the state of the keyboard can be requested using `mInputDevice->isKeyDown` method.

The available functional keys are the following ones:

- **A**: Move camera left
- **D**: Move camera right
- **UP**: Move camera forward by keypress.
- **DOWN**: Move camera backward by keypress.
- **W**: Move camera up
- **S**: Move camera down
- **RIGHT**: Yaw camera left
- **LEFT**: Yaw camera right
- **ESC**: Exit application
- **M**: Switch mouse mode
- **T**: Switch between TRILINEAR, ANISOTROPIC and BILINEAR filtering
- **SYSRQ**: Screenshot
- **R**: Switch between SOLID, WIREFRAME, POINTS polygon modes
5.1.3 Frame Routine

The core of all listener lays in the frame routine. In the case of theMainFrameListener, all the changes in the scene which have taken place between two frames are rendered in this step.

Among others, the main tasks the listener performs are the following ones:

5.1.3.1 Adjusting Dynamic Meshes Position, Ray-Tracing

For every frame and dynamic object currently shown in the scene, a ray is traced from the object center to the terrain, perpendicularly, to find the exact height point of intersection with the current terrain.

```cpp
actual_position = actual_entity.position;
RaySceneQuery* mRayQuery;
mRayQuery = createRayQuery(actual_position,
NEGATIVE_UNIT_Y);
intersection = mRayQuery->execute();
if (intersection )
{
    new_position = (actual_position.x,
        intersection.y +
    actual_entity.center.y,
        actual_position.z);
    actual_entity.position = new_position;
}
mSceneMgr->destroyQuery(mRayQuery);
```
5.1.3.2 Adjusting Animation State

For every frame and animated object currently shown in the scene, the animation state of its current animation is updated to match the elapsed time since the last frame rendered.

```c++
if (actual_entity->hasAnimation())
{
    mAnimationState = actual_entity.getAnimationState();
    mAnimationState->setEnabled(true);
    mAnimationState->addTime( 
        evt.timeSinceLastFrame );
}
```

5.1.3.3 Adjusting Preview Window Mesh Rotation

Every frame, if the mesh preview window is shown and has a selected mesh in it, it is updated with the new rotation, depending on the elapsed time since the last frame.

```c++
actualMeshRotangle = (ROTATION_SPEED * 
    timeSinceLastFrame));
if (actualMeshRotangle >= 360.0)
    actualMeshRotangle = actualMeshRotangle - 360;
dialog_main_node->yaw(actualMeshRotangle);
```
5.1.3.4 Adjust Camera Orientation and Position

Every frame, if the camera has to be moved by the application (to show a new placed object, or the terrain from the top, for example) its position and rotation is adjusted based on the elapsed time since last frame.

```cpp
if (moveCamera)
{
    actual_position = cameraNode.position;
    direction = camera_destination - actual_position;
    direction.normalise();
    displacement = direction * camMoveSpeed * timeSinceLastFrame;

    RotProgress += RotFactor * timeSinceLastFrame;
    if (RotProgress < 1.0)
    {
        cameraNode.position = actual_position + displacement;
        Quaternion delta = slerp(RotProgress, OrientSource, OrientDestination);
        cameraNodePitch->setOrientation(delta);
    }
    else
    {
        moveCamera = false;
    }
}
```
5.1.4 **Process Mouse Click**

After a user's mouse click, the MainFrameListener checks if it is in mesh placing state, and if so, calculates the terrain point clicked, and places a new object like the one shown in the preview window there.

After that, calculates the route for the camera to travel from its current point (probably, at the top of the terrain, looking to the ground for easier mesh placing).

```c++
if (clickableTerrain)
{
    // Setup the ray scene query
    mouseRay = getCameraToViewportRay( mouse.x, mouse.y );
    mRaySceneQuery->setRay( mouseRay );

    // Execute query
    intersection = mRaySceneQuery->execute();

    if (intersection )
    {
        // Get results, create a node/entity on
        // the position
        new_entity = mSceneManager->createEntity(entityName, meshName);

        box = dialog_entity->getBoundingBox();
        center = box.getCenter();

        main_node = world_node->createChildSceneNode();
        main_node->position = intersection + center.y;

        mesh_node = main_node->createChildSceneNode();
        mesh_node->position = -center;
        mesh_node->attachObject( new_entity );

        // Calculate destination point for camera
        // travelling

        Camera_destination = intersection -
        entity_radius.z);

        updateCEGUI();
    }
}
```
addAnimations();

// Calculate the speed of the camera
// movement, in order to get to
// destination in 3 seconds

camMoveSpeed = (destination -
camera.position).length()
    / 3;

// Obtain the source orientation and
// calculate the destination orientation

orientationSource = mCameraNodePitch.orientation * UNIT_Y;
orientationSource.normalise();

// We want dest orientation, not a
// relative rotation (quat)
Quaternion quat =
orientationSource.getRotationTo(UNIT_Y);
mOrientDest = quat * mOrientSrc;

mRotProgress = 0;
mRotFactor = (quat.yAxis().length()) / 3;

// Update flags
meshIndex++;
moveCamera = true;
clickableTerrain = false;
5.2 MeshViewportListener

The MeshViewportListener implements a rendering queue. When the preview window is showing a mesh, it is necessary to hide the rest of the scene, and when rendering the scene, it is not desirable to show the preview mesh in it.

To achieve this effect, the MeshViewportListener has the control of the preview window mesh rendering. Each time it is necessary to render it, the listener takes control and hides de world_node hierarchy, the sky, and the terrain.

As all the sceneNodes but the dialog_node hierarchy depend on world_node, the entire scene is hided whenever the world_node visibility is disabled.

At the end of the rendering pass of the preview mesh, the world_node hierarchy, the sky and the terrain visibility are enabled again, while the dialog_node hierarchy is shown.

```cpp
preRenderTargetUpdate()
{
    mSceneMgr->getTerrainRootNode()->setVisible(false);
    world_node->setVisible(false);
    scene->showSky(false);
    dialog_main_node->setVisible(true);
}

postRenderTargetUpdate()
{
    dialog_main_node->setVisible(false);
    mSceneMgr->getTerrainRootNode()->setVisible(true);
    world_node->setVisible(true);
    scene->showSky(true);
}
```
6. DOMAIN LAYER

6.1 MESHOBJECT

All the entities shown in the scene are embedded in a MeshObject. This meshObject contains both the node hierarchy and the mesh data of the entity.

The MeshObject also has the information about which animations has available the mesh of the entity, and allows to change between them dynamically.

Furthermore, the meshObjects can be static or dynamic. The static meshObjects remain in the same place they were placed by the user, although they can have an animation. The dynamic meshObjects move around the scene randomly.
6.1.2 Dynamic MeshObjects

The dynamic meshObjects move around the scene randomly. When an entity reaches its destination, a new one (between the terrain boundaries) is generated. The entity is then rotated to look ahead in the new direction, and, frame by frame, its position is updated based on the movement speed and the elapsed time between frames.

Vector3 pos = main_node->getPosition();

Vector2 v1(destination.x, destination.z);
Vector2 v2(pos.x, pos.z);
if ((v1 - v2).length() <= 0.0f)
{
    //If the object has arrived its destination,
    //we generate a new
    destination.x = (rand() % max_x) + min_x;
    destination.z = (rand() % max_z) + min_z;
    destination.y = pos.y;
    direction = destination - pos;
    direction.normalise();

    //we get the source orientation, to let the
    //entity face the //direction in which it is moving
    Vector3 src = mesh_node->getOrientation() * UNIT_X;
    Quaternion quat = src.getRotationTo(direc
    mesh_node->rotate(quat);
}

move = mWalkSpeed * time;
main_node->translate(direction * move);
6.2 SceneConfig

The SceneConfig class contains all the data related to the sky and terrain of the scene.

It keeps the information about which type of sky, if any, is being used: SkyBox, SkyDome or SkyPlane. For each of them, it keeps the current values of their parameters, as well as which material apply to them.

It also saves all the parameters of the current terrain (if any) like the heightmap, the textures or the size.

It has got also the methods to write and load scene configurations to a file, using the format .scene.
7. GRATITUDES

For all help provided, and making possible that I could develop this Master Thesis as an Erasmus student:

• Ms. Dr. Nuria Castell
  (ViceDean
   for International Affairs)

For all support, supervision, advices and general help provided

• Prof. Dr. Borut Zalik
  (Director of the Thesis)
• Elisa Grabner
  (Erasmus coordinator assistant at FERI – University of Maribor)
8. BIBLIOGRAPHY

http://www.ogre3d.org/docs/manual/

http://www.ogre3d.org/docs/api/html/


http://www.cegui.org.uk/api_reference/

Pro Ogre 3D Programming, Gregory Junker

3D Game Engine Programming, Stefan Zerbst with Oliver Düvel

Game Coding, Complete, Second Edition, Mike McShaffry