Design documentation of the Ocerus project

Lukas Hermann, Ondrej Mocny, Tomas Svoboda, Michal Cevora

December 26, 2010

Contents

1 Introduction **2**
1.1 Purpose of this document 2
1.2 Project architecture 2

2 Core **5**
2.1 Purpose of the core 5
2.2 Application 6
2.3 Game 7
2.4 Loading screen 8
2.5 Configuration 9
2.6 Project 9
2.7 Glossary 9

3 Gfx system **11**
3.1 Purpose of the graphic system 11
3.2 Graphic viewport and render target 12
3.3 Renderer and scene manager 12
3.4 Application window 13
3.5 Mesh and texture 13
3.6 Glossary 13

Bibliography **15**

List of Figures **17**

List of Tables **18**

Chapter 1

Introduction

1.1 Purpose of this document

1.2 Project architecture

The project Ocerus is logically divided into several relatively independent systems which cooperate with each other. Every system maintains its part of the application such as graphics, resources, scripts etc. and provides it to other ones. In the picture 1.1 the relations among all systems are displayed with a brief description of what the systems provide to each other.

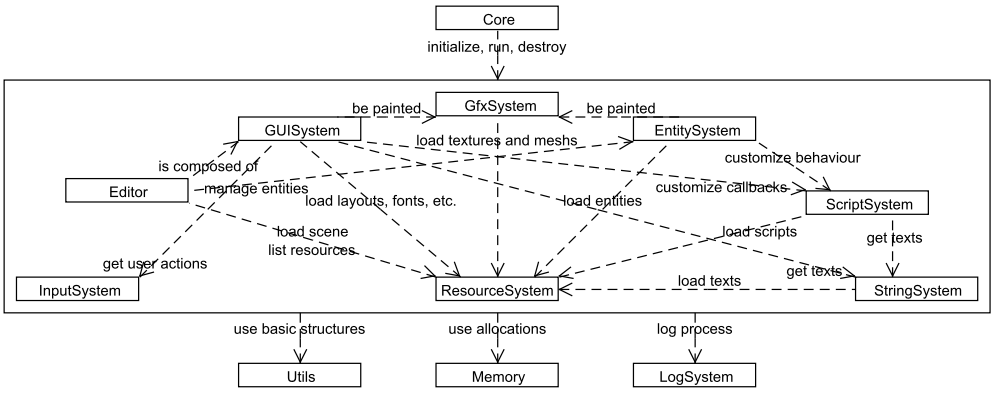

Figure 1.1: Dependencies among the systems

The project has not been created from scratch but it is based on several libraries to allow the developers to focus on important features for the end users and top-level design rather than low-level programming. All used

libraries support many platforms, have free licenses and have been heavily tested in a lot of other projects. All of them are used directly by one to three subsystems except the library for unit testing. The library dependences of each system are displayed in the picture 1.2.

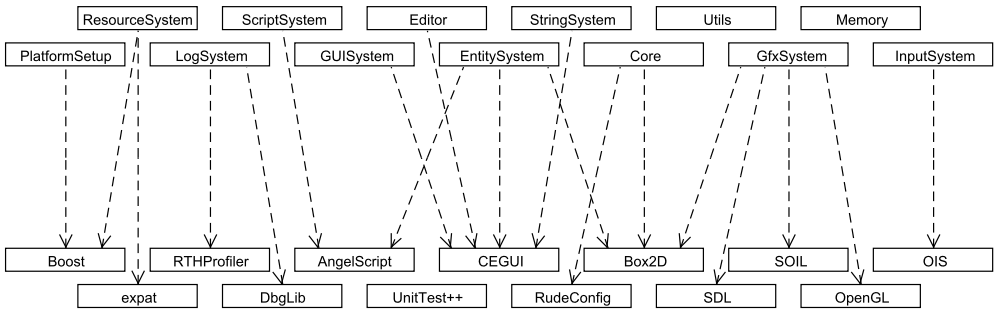

Figure 1.2: Library dependencies of the project systems

In this list a brief description of all used libraries is provided:

- AngelScript[1] – a script engine with an own language
- Boost[2] – a package of helper data structures and algorithms
- Box2D[3] – a library providing 2D real-time physics
- CEGUI[4] – a graphic user interface engine
- DbgLib[5] – tools for a real-time debugging and crash dumps
- Expat[6] – a XML parser
- OIS[7] – a library for managing events from input devices
- OpenGL[8] – an API for 2D and 3D graphics
- RTHProfiler[9] – an interactive real-time profiling of code
- RudeConfig[10] – a library for managing configure files
- SDL[11] – a tool for an easier graphic rendering
- SOIL[12] – a library for loading textures of various formats
- UnitTest++[13] – a framework for a unit testing

Except these libraries some small pieces of a third party code were used:

- Properties and RTTI[14] – a basic concept of entity properties and runtime type information
- Tree[15] – an STL-like container class for n-ary trees
- FreeList[14] – free lists / memory pooling implementation
- STL pool allocator[16] – pooled allocators for STL
- GLEW[17] – the OpenGL extension wrangler library
- OBJ loader[18] – the Wavefront OBJ file loader
- PlusCallback[19] – an easy use of function and method callbacks
- Script builder and script string[1] – an implementation of strings in the script engine and building more files to a script module

In the following chapters each of the project systems will be described from the design view. At the beginning of each chapter there are a UML class diagram and a section about a purpose of the described system and at the end of most chapters there is a small glossary of terms used in that chapter.

Chapter 2

Core

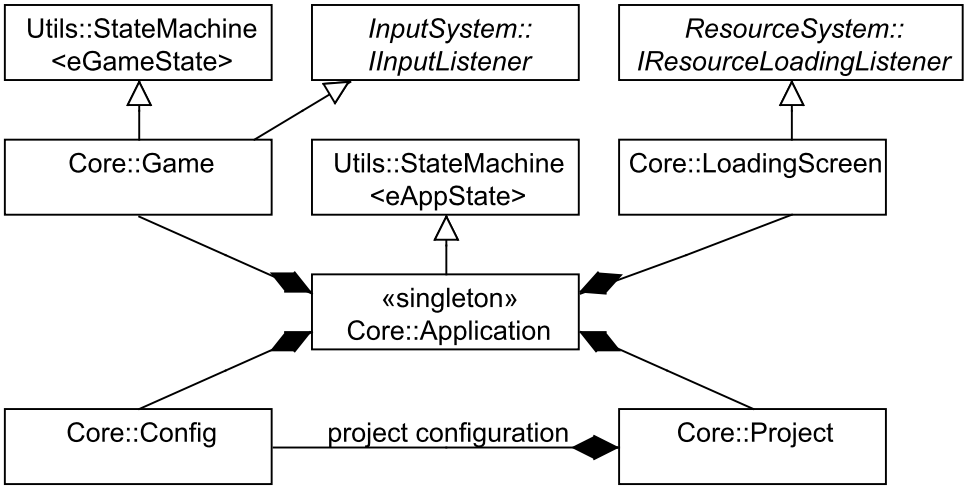

Figure 2.1: Class diagram of namespace Core

2.1 Purpose of the core

The Core namespace is the main part of the whole system. It contains its entry point and other classes closely related to the application itself. Its main task is to initialize and configure other engine systems, invoke their update and draw methods in the main loop and in the end correctly finalize them.

In the following sections the class representing the application as well as the classes corresponding to the application states (loading screen, game),

configuration and project management will be introduced. In the last section there is a small glossary of used terms.

2.2 Application

When the program starts it creates an instance of the class *Core::Application*, initializes it by calling its method *Init* and calls the *RunMainLoop* method which runs until the application is shutdown, then the instance is deleted and the program finishes (see figure 2.2).

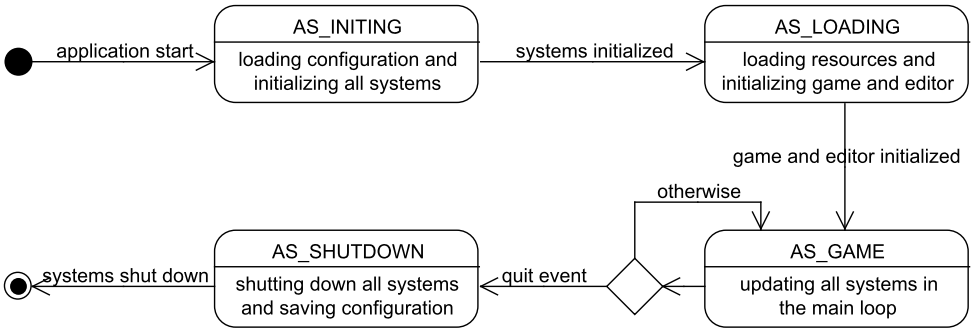

Figure 2.2: Possible states of the Application class

On the initialization of the application the configuration is read (see section 2.5) and all engine systems are created and initialized as well as the loading screen and game classes. The state of application is changed to *loading* and the main loop is running until the state is changed to *shutdown*. At the main loop window messages are processed, performance statistic are updated and other engine systems including the game class are loaded (in the *loading* state) or updated and drawn (in the *game* state).

In the application class there are also methods for getting an average and last FPS statistic, methods for showing and hiding a debug console as well as writing message to it or a method for executing an external file. There are also the variables indicating whether the current application instance includes the editor (in a game distribution the editor should be disabled) and whether the editor is currently turned on so the game is running only in a small window instead of a full screen mode. From this class it is possible to get the current project as well as deploy it to the specific platform and destination.

2.3 Game

The *Core::Game* class manages the most important stuff needed to run the game such as drawing a scene, updating physics and logic of entities, measuring time, handling a game action or resolving an user interaction. Of course it mostly delegates this work to other parts of the engine (see table 2.1).

Design entity	Relation to the Game class
Game is affected by	
Application	initializes, updates, destroys it
Editor	sets render target, can delegate input
InputMgr	can delegate input
ResourceMgr	loads the saved game
Game affects what	
GfxRenderer	invokes drawing entities
Physics	initializes, updates and destroys it, processes its events
EntityMgr	broadcasts update and draw messages to entities
ScriptMgr	gives the game time
GUIMgr	stores the root window for the game GUI

Table 2.1: Relations of the Game class

Before the game initialization at the method *Init* a valid render target (a camera and a viewport, see section 3.2) must be set by the method *SetRenderTarget* or the default one must be created by the method *CreateDefaultRenderTarget* to know where to draw the game content. This is done by the *Core::Project* class when a scene is being opened and it can delegate it to the editor if it is available. Then physics, time, an action etc. are initialized and in the *Update* method called in the main loop they are updated.

The drawing of a scene is invoked in the method *Draw*. The render target is cleared, all entities in the current scene are drawn by a renderer and the rendering is finalized.

There are several methods for handling a game action. The action can be paused, resumed and restarted to previously saved position. There is a global timer that measures game time (can be obtain by the method *GetTimeMillis*) when the game is running which is used by other systems such as the script system.

When the action is running physics and logic of entities are updated in the method *Update* which means the corresponding messages are broadcast to all entities before and after the update of the physical engine.

Since the class *Core::Game* registers the input listener to itself there are callbacks where it is possible to react to keyboard and mouse events such as

a key or mouse button press/release or a mouse move. The corresponding information such as a current mouse position is available through the callback parameters.

If it is necessary to store some extra information that is shared among the game scenes (i.e. total score) the dynamic properties of this class should be used. There are template methods for getting or setting any kind of value under its name as well as methods for deleting one or all properties and for loading and saving them from/to a file. The properties are now stored along with other game stuff.

2.4 Loading screen

The *Core::LoadingScreen* class loads resource groups into the memory and displays information about the loading progress. It is connected to the resource manager that calls its listener methods when a resource or a whole resource group is going to be loaded or has been already loaded so it can update progress information.

First it is necessary to create an instance of the *Core::LoadingScreen* class. The only method of this class that should be called explicitly is the *DoLoading* one. The first parameter represents the kind of data to be loaded. Basic resources containing necessary pictures for a loading screen must be loaded first, then general resources needed in most of the states of the application should be loaded. If the editor should be available its resources must be in the memory too. The last usage of this method is the loading of scenes where the second parameter (a name of a scene) must be filled.

The *DoLoading* method invokes the resource manager for loading corresponding resources and the manager calls callback methods informing about the state of loading. For each resource group the *ResourceGroupLoadStarted* method is called first with the group name and a count of resources in the group. Then for each resource in the group the *ResourceLoadStarted* method with a pointer to the resource class is called before the loading starts and the *ResourceLoadEnded* method is called after the loading ends. Finally when a whole resource group is loaded the *ResourceGroupLoadEnded* is called. Each of these methods calls the *Draw* method that shows the loading progress to the user.

In the present implementation the loading progress is shown as a ring divided to eight parts that one of them is drawn brighter than the others. Once a while the next part (in a clockwise order) is selected as a brighter one. Since this implementation shows only that something is loading but not the real progress it can be changed if it is necessary.

2.5 Configuration

The *Core::Config* class allows storing a configuration data needed by various parts of the program. It serves as a proxy class between the engine and the RudeConfig library[10]. Supported data types are strings, integers and booleans and they are indexed by text keys and they can be grouped to named sections.

This class is initialized by a name of the file where data are or will be stored. Although changes to a configuration are saved when the class is being destructed it is possible to force it and get the result of this action by the method *Save*.

There are several getter and setter methods for each data type that get or set data according to a key and a section name. A section parameter is optional, the section named `General` is used as a default. The getter methods have also a default value parameter that is returned when a specific key and section do not exist in a configuration file. It is possible to get all keys in a specific section to a vector with the method *GetSectionKeys* or remove one key (*RemoveKey*) or a whole section (*RemoveSection*).

2.6 Project

The *Core::Project* class manages the project and its scenes both in the editor and in the game. There are methods for creating and opening a project in a specific path as well as closing it and getting or setting project information (a name, a version, an author). Other methods of this class manage scenes of the project – creating, opening, saving, closing etc. Some methods like creating or saving scenes can be called only in the editor mode and are not accessible from scripts.

2.7 Glossary

This is a glossary of the most used terms in the previous sections:

Loading screen – a screen visible during a loading of the game indicating a loading progress

Main loop – a code where an input from user is handled, an application logic is updated and a scene is drawn in a cycle until an application shut down

FPS – a count of frames per second that are drawn indicates a performance of a game

Render target – a region in an application window where a game content is drawn to

Resource – any kind of data that an application needs for its running (i.e. pictures, scripts, texts etc.)

Configuration data – data that parameterizes the application running (i.e. a screen resolution, a game language etc.)

Project – represents one game created in the editor that can be run independently, it is divided to scenes

Scene – represents one part of the game that is loaded at once (i.e. a game level, a game menu etc.)

Chapter 3

Gfx system

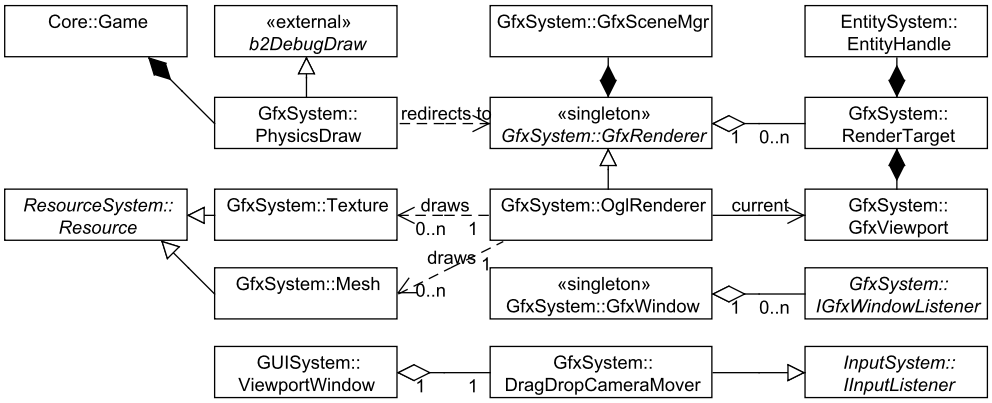

Figure 3.1: Class diagram of namespace GfxSystem

3.1 Purpose of the graphic system

The graphic system implements functionalities related to the rendering of game entities and the management of the application window. The design of this system is influenced by the requirement of platform independence. Note that the GUI system uses its own rendering system.

In the following sections the concept of viewports and render targets will be described as well as the process of rendering game entities, the way of creating the application window will be revealed and the management of meshes and textures will be introduced. In the last section there is a small glossary of used terms.

3.2 Graphic viewport and render target

The *GfxSystem::GfxViewport* class defines a place where all game entities will be rendered. It simply stores the information about a position and a size within the global window that can be obtained from some texture and also the data needed for drawing a grid which is useful in the edit mode. It has methods for getting and setting these properties as well as other ones for calculating its boundaries in the world or scene space.

For drawing the game entities it is also necessary to know from which position they are rendered so the *GfxSyste::RenderTarget* type is defined which is a pair of a viewport and an entity handle that must point to an entity with a camera component. This type is used by renderer classes described below where it is indexed by the *GfxSystem::RenderTargetID* type which is defined as an integer.

For easy moving and zooming a camera by a mouse in a render target the *GfxSystem::DragDropCameraMover* class was defined. In its constructor or later by its setters it is possible to adjust a zoom sensitivity and a maximal and minimal allowed zoom.

3.3 Renderer and scene manager

The *GfxSystem::GfxRenderer* is the main class that manages a rendering of entities to render targets. This is a platform independent abstract class handling a communication with other engine systems from which now derives only the *GfxSystem::OglRenderer* class implementing a low level rendering in the OpenGL library[8]. If it is necessary to implement a rendering for another library (i.e. DirectX) it should be done by deriving another class and implementing all abstract methods.

The abstract class has methods for managing its render targets, for drawing simple shapes as well as textures and meshes or for clearing the screen. The rendering must be started by the *GfxRenderer::BeginRendering* method, then the current render target must be set and cleared. After everything is drawn the *GfxRenderer::FinalizeRenderTarget* method must be called and then another render target is set or the whole rendering is finished by the *GfxRenderer::EndRendering* method.

An important attribute of the *GfxSystem::GfxRenderer* class is the pointer to the *GfxSystem::SceneMgr* class created on its initialization accessible by the *GfxRenderer::GetSceneManager* method. This is the class to which all drawable components (sprites, models) must be registered along with a *Transform* component of their entity by the *SceneMgr::AddDrawable* method

so then they are rendered by the *SceneMgr::DrawVisibleDrawables* method if they are visible.

To provide debug drawing of physics entities the *GfxSystem::PhysicsDraw* proxy class was defined and registered as an implementation of the *b2DebugDraw* class from the Box2D library. All methods are redirected to corresponding methods in the *GfxSystem::GfxRenderer* class.

3.4 Application window

The graphic system also manages creating and handling the application window which depends on the used operating system. This functionality is implemented by the *GfxSystem::GfxWindow* class with the usage of the SDL library. This class has methods for getting and setting a window position, size and title or a visibility of a mouse cursor, toggling a fullscreen mode and handling system window events. It is also possible to register a screen listener represented by a class implementing the *GfxSystem::IGfxWindowListener* interface. This class will be informed when the screen resolution is changed.

Note that the SDL library also provides features in low-level audio and input management but since audio is not yet implemented and input management is done by more specialized library, the only used SDL features used are window management and creating rendering context.

3.5 Mesh and texture

Meshes and textures are essential parts of the *Model* and *Sprite* components. They can be loaded via the *GfxSystem::Mesh* and *GfxSystem::Texture* classes that inherit from the *ResourceSystem::Resource* class (for more information see chapter about the resource system).

On loading of a texture resource the *GfxRenderer::LoadTexture* abstract method is called. For OpenGL implementation the SOIL library is used which is a tiny C library used for uploading textures into the OpenGL and which supports most of the common image formats.

For defining meshes the Wavefront OBJ file format [20] is used. Every texture used in the model definition is automatically loaded as a resource.

3.6 Glossary

This is a glossary of the most used terms in the previous sections:

Viewport – a region of the application window where entities are rendered to

Render target – a pair or a viewport and a camera

Sprite – a component for showing an entity as an image (even animated or transparent)

Model – a component for showing an entity as a 3D-model

Texture – a bitmap image applied to a surface of a graphic object

Mesh – a collection of vertices, edges and faces that defines the shape of a polyhedral object

Bibliography

[1] AngelScript – http://www.angelcode.com/angelscript

[2] Boost – http://www.boost.org

[3] Box2D – http://www.box2d.org

[4] CEGUI – http://www.cegui.org.uk

[5] DbgLib – http://dbg.sourceforge.net

[6] Expat – http://expat.sourceforge.net

[7] OIS – http://sourceforge.net/projects/wgois

[8] OpenGL – http://www.opengl.org

[9] Real-Time Hierarchical Profiling – Greg Hjelstrom, Byon Garrabrant: Game Programming Gems 3, Charles River Media, 2002, ISBN: 1584502339

[10] RudeConfig – http://rudeserver.com/config

[11] SDL – http://www.libsdl.org

[12] SOIL – http://www.lonesock.net/soil.html

[13] UnitTest++ – http://unittest-cpp.sourceforge.net

[14] Kim Pallister: Game Programming Gems 5, Charles River Media, 2005, ISBN: 1584503521

[15] Kasper Peeters, http://www.aei.mpg.de/ peekas/tree

[16] http://www.sjbrown.co.uk/2004/05/01/pooled-allocators-for-the-stl

[17] http://glew.sourceforge.net

[18] http://www.dhpoware.com

[19] http://codeplea.com/pluscallback

[20] Wavefront OBJ file structure – http://en.wikipedia.org/wiki/Obj

[21] CEGUI documentation – http://cegui.org.uk/api_reference/index.html

[22] AngelScript documentation – file /AngelScript/index.html

[23] ISO 639-1 – http://en.wikipedia.org/wiki/List_of_ISO_639-1_codes

[24] ISO 3166-1 alpha-2 – http://en.wikipedia.org/wiki/ISO_3166-1_alpha-2

List of Figures

1.1 Dependencies among the systems 2
1.2 Library dependencies of the project systems 3

2.1 Class diagram of namespace Core 5
2.2 Possible states of the Application class 6

3.1 Class diagram of namespace GfxSystem 11

List of Tables

2.1 Relations of the Game class 7