Ocerus Design Documentation

http://ocerus.sourceforge.net

February 21, 2011

Contents

Chapter 1

Introduction

This is the design documentation of the Ocerus project. It is written for developers who want to understand how the engine work or need to change some parts of the engine. It may be useful for everyone who is curious how the application is designed or how libraries listed in the table 1.1 can be used. However, it is not intended for developers who want to extend Ocerus. For this purpose there is the Extending Ocerus guide.

1.1 Project architecture

The Ocerus project is 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 and interface to the other ones. In the figure 1.1 the relations among all systems are displayed with a brief description of what the systems provide to each other.

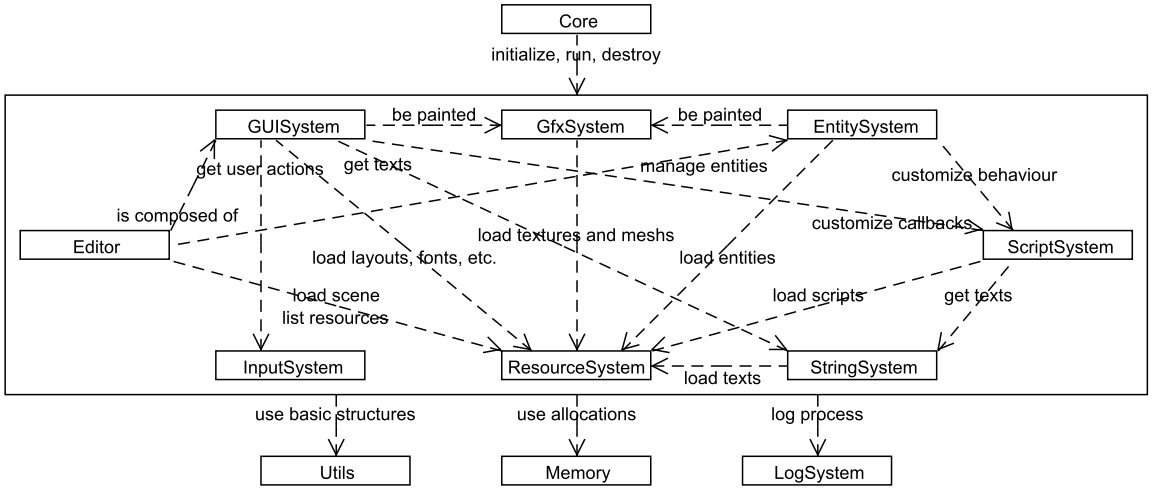

Figure 1.1: Dependencies among the systems

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 the low-level programming. All libraries used 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 four subsystems except the library for unit testing. The library dependencies of each system are displayed in the figure 1.2.

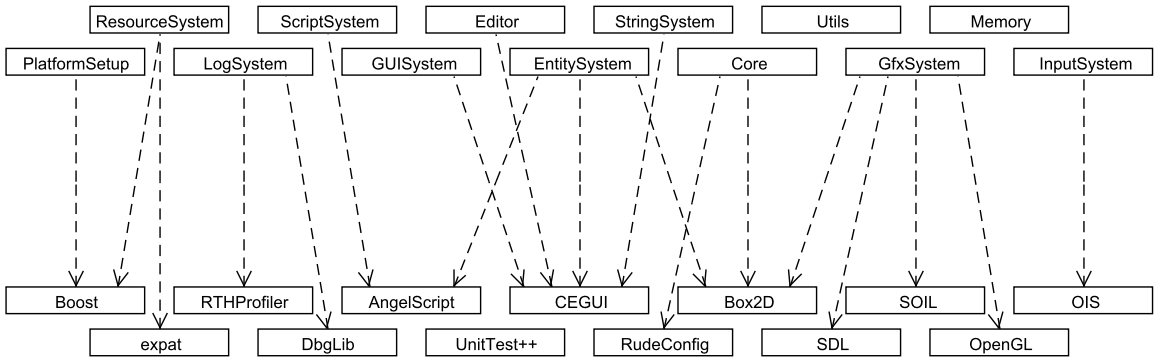

Figure 1.2: Library dependencies of the project systems

In the table 1.1 thre's a brief description of all libraries used.

Library	Description
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

Table 1.1: Libraries and their description

Except these libraries, some small pieces of a third party code that were used are listed in the table 1.2.

In the following chapters each of the project systems will be described from the design point of view. At the beginning of each chapter there is a UML class diagram and a paragraph about the purpose of the described system and at the end of most chapters there is a small glossary of terms used in that chapter. In the UML class diagrams, two stereotypes are used. The first one is the `<<singleton>>` stereotype used in classes derived from the *Utils::Singleton<T>* class, which is an implementation of the singleton design pattern. The second one is the `<<external>>` stereotype used in classes defined and implemented in some of the libraries described in the table 1.1.

Third party code	Description
Properties, RTTI[14]	a basic concept of entity properties and run-time type information
Tree[15]	an STL-like container class for n-ary trees
FreeList[14]	free lists / memory pooling implementation
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, script string [1]	an implementation of strings in the script engine and building more files to a script module

Table 1.2: Third party code with the description

Chapter 2

Core

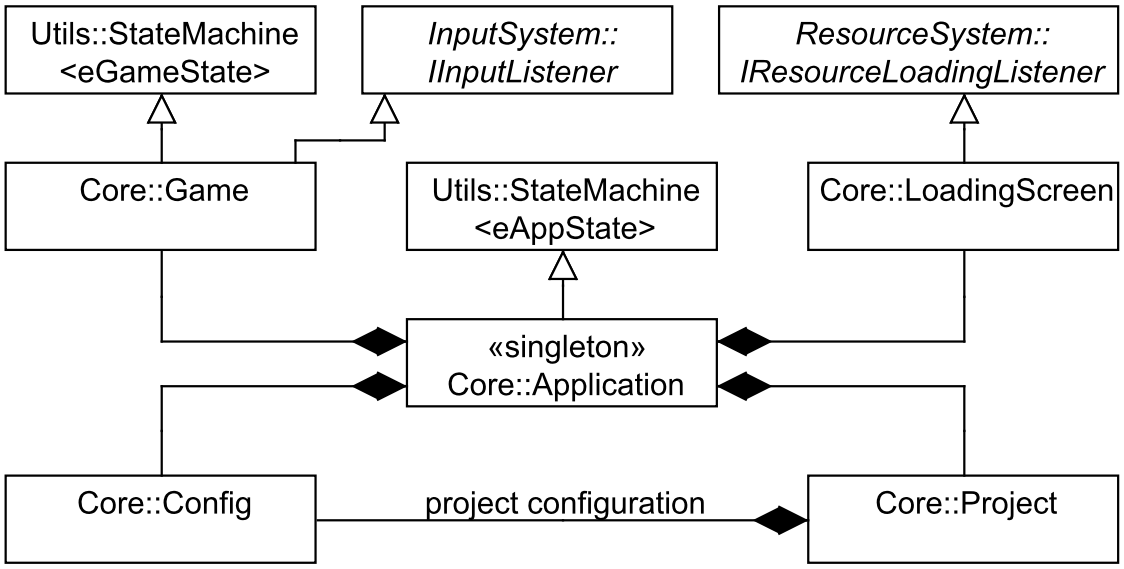

Figure 2.1: Class diagram of the Core namespace

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

2.1 Application

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

At the initialization of the application, the configuration is read (see section 2.4) and all engine systems are created and initialized as well as the loading screen and game

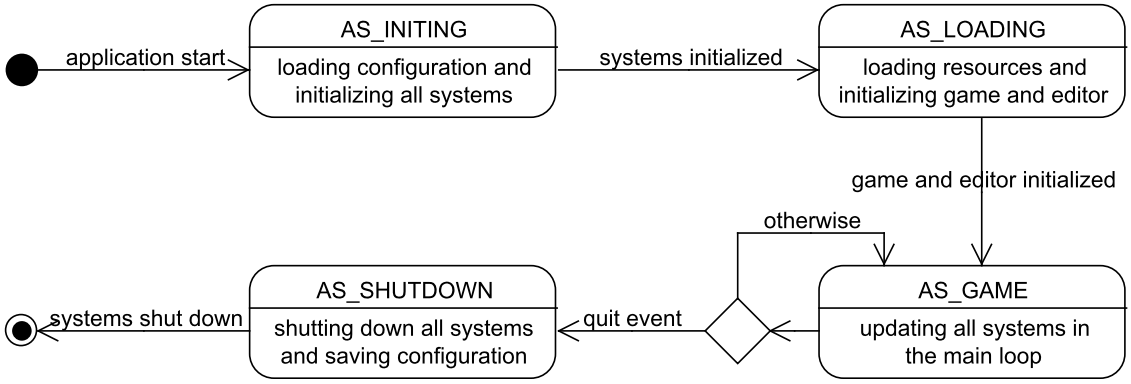

Figure 2.2: Possible states of the Application class

classes. The state of the application is changed to *loading* and the main loop runs 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 the average and the last-frame FPS number. Then there are methods for showing and hiding the debug console as well as writing message to it or a method for executing an external file in the OS shell. There are also 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. If so it means the game is running only in a small window instead of the full screen mode. From this class it is possible to get the currently opened project as well as deploy it to a specific platform.

2.2 Game

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

Before the game is initialized by the *Init* method a valid render target (a camera and a viewport, see section 4.1) must be set by the *SetRenderTarget* method. If it's not set the game will attempt to use the default render target by calling the *CreateDefaultRenderTarget* method. The render target is pasesd to the game from 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 of 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 that can be in two states – paused or running. When the action is running, the physics and the logic of the entities is being updated in the method *Update*, which means that the corresponding messages are

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

broadcasted to all entities before and after the update of the physical engine. The action can be paused, resumed and restarted to a previously saved position. There is a global timer that measures the game time (can be obtained by the method *GetTimeMillis*), when the game is running, which is used by other systems, such as the script system.

Since the class *Core::Game* registers the input listener to itself, there are callbacks where it is possible to react to the keyboard and mouse events such as the key or the mouse button press/release or a mouse movement. The corresponding information such as the 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. For more information about using dynamic properties see the section 14.3.

2.3 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 the 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 into eight parts and one of them is drawn brighter than the others. Once in 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.

2.4 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 the 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*).

2.5 Project

The *Core::Project* class manages the project and its scenes in both the editor and 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.

2.6 Glossary

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

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

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

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

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

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

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

Project – represents a single 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

Entity system

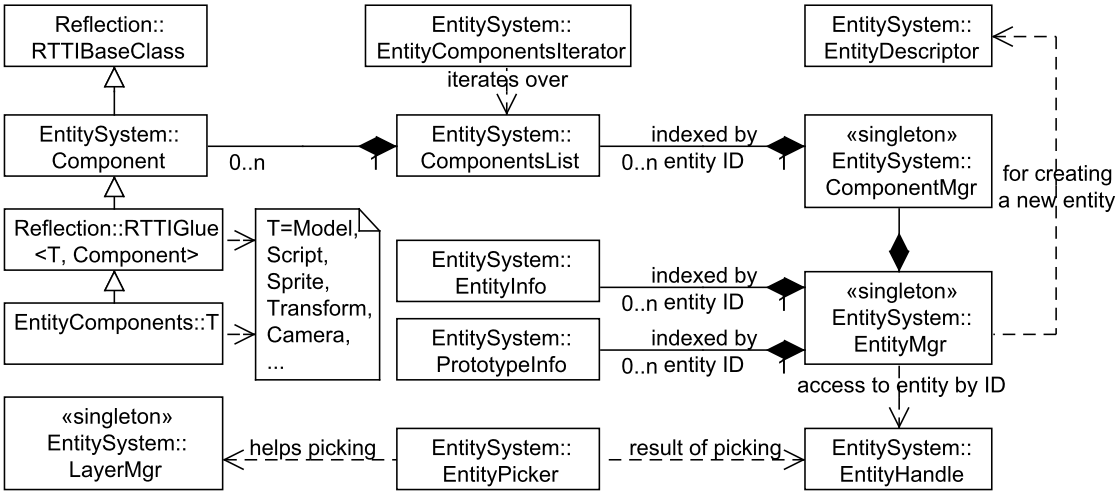

Figure 3.1: Class diagram of the Entity system namespace

The entity system creates a common interface for a definition of all game objects such as a game environment, a player character, a camera etc. and their behavior such as a drawing on a screen, an interaction with other objects etc. The object creation is based on a composition of simple functionalities that can be reused in many of them. The advantage of this unified system is an easy creating and editing of new objects from the game editor or from scripts. The disadvantage is slower access to the object properties and behavior. It cooperates with other systems like the graphical renderer to display objects or the script system to interact with the game from scripts.

3.1 Components and their manager

Every game object is represented by an entity, which is a compound of components that provides it with various functionalities. A component can have several properties (and functions), which can be read or written (called) via their getters and setters (or functions themselves), and which are accessible through their unique name. It can also

react to messages such as initialization, drawing, logic update etc. by its own behavior. Component properties and behaviors are accessible only through the entity owning the component. So, it is possible to read or write a specific property of an entity, if it contains a component with this property. It is also possible to send a message to an entity that dispatches it to all of its components that can then handle it.

The *EntitySystem::Component* class is a base class for all components used in the entity system. It inherits from the *Reflection::RTTIBaseClass* class, which provides the methods for working with RTTI (registering properties and functions of component, see section 14.2). It has methods for getting the owner entity, the component type (defined in ComponentEnums.h) and the component property from its name and for posting a message to the owner entity. It also introduces methods, which should be overridden by specific components, that are used for handling messages and the component creation and destruction (see the Extending Ocerus document).

The *EntitySystem::ComponentMgr* is a singleton class that manages instances of all entity components in the entity system (see the figure 3.2). Internally it stores mapping from all entities to lists of their components. It provides methods for adding a new component of a certain type to an entity and listing or deleting all or specific components from an entity. For passing all components of an entity the *EntitySystem::EntityComponentsIterator* iterator, which encapsulates a standard iterator, is used (for example it has the *HasMore* method which returns whether the iterator is at the end of the component list).

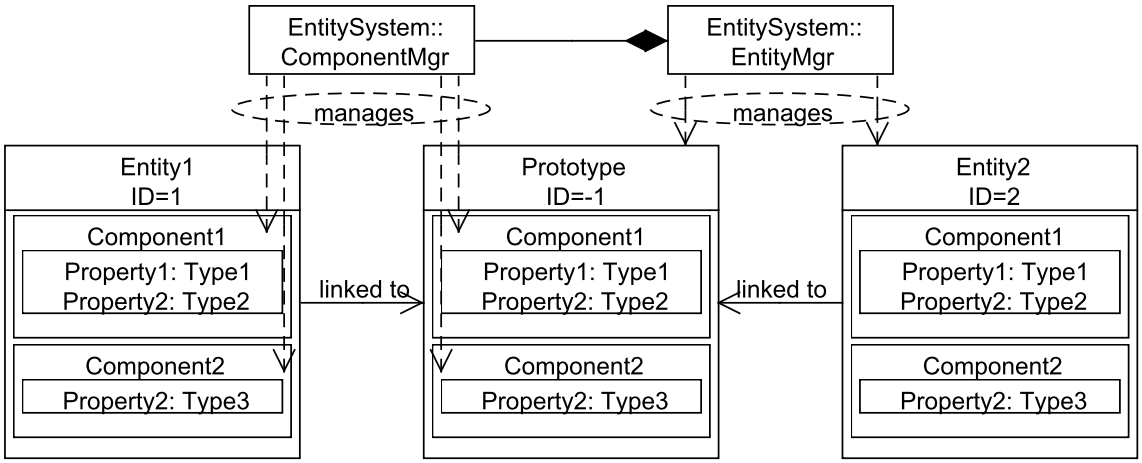

Figure 3.2: Objects managed by entity and component managers

3.2 Entities and their manager

An entity is represented by the *EntitySystem::EntityHandle* class which stores only an ID of the entity and provides methods that mostly calls corresponding methods of the entity manager with its ID. This class has also static methods that ensure that all IDs in the system are unique.

For the creation of one entity, the *EntitySystem::EntityDescription* class is used. It is basically a collection of component types. There are methods for adding a component type and setting a name and a prototype of the entity. It is also possible to set whether the created object will be an instance or a prototype of an entity. Prototypes of entities are used to propagate changes of their shared properties to the instances that are linked to them so it is possible to change properties of many entities at once. Instances must have all components that has their prototype in the same order, but they can also have own additional components that must be added after the compulsory ones.

It is possible to send messages to entities so there is the *EntitySystem::EntityMessage* structure that represents them. It consists of the message type defined in EntityMessageTypes.h and the message parameters that are an instance of the *Reflection::PropertyFunctionParameters* class. To add a parameter of any type defined in PropertyTypes.h, the *PushParameter* method can be called with a value as first argument, or the *operator<<* can be used. There is also a method that checks whether the actual parameters are of the correct types according to the definition of message type (see section the Extending Ocerus document for more information).

All entities are managed by the *EntitySystem::EntityMgr* class that stores necessary information about them in maps indexed by their ID (see the figure 3.2). The most of its methods has the entity handle as the first parameter that means it applies on the entity of the ID got from the handle. There are methods for creating entities from an entity description, a prototype, another entity or an XML resource and for destroying them. Other methods manages entity prototypes – it is possible to link/unlink an instance to/from a prototype, to set a property as (non)shared, to invoke an update of instances of a specific prototype and to create a prototype from a specific entity. Finally, there are methods for getting entity properties even of a specific component (in case of two or more properties of a same name in different components), for registering and unregistering dynamic properties, for posting and broadcasting messages to entities and for adding, listing and removing components of a specific entity (see the figure 3.3 for implementation details).

3.3 Entity picker

The entity picker implemented by the *EntitySystem::EntityPicker* class is a mechanism to select one or more entities based on their location. If the picker is used to select a single entity, all it needs is a position in the world coordinates. The query then returns the found entity or none. This feature can be used to select the entity the mouse cursor is currently hovering over. The cursor position must be translated into the world coordinates via the rendering subsystem and its viewports. If the picker is used to select more entities, a query rectangle (along with its angle) must be defined. This feature can be used to implement a multiselection using the mouse or gamepad. It is also possible to define two layers, between which the picked entities must lie.

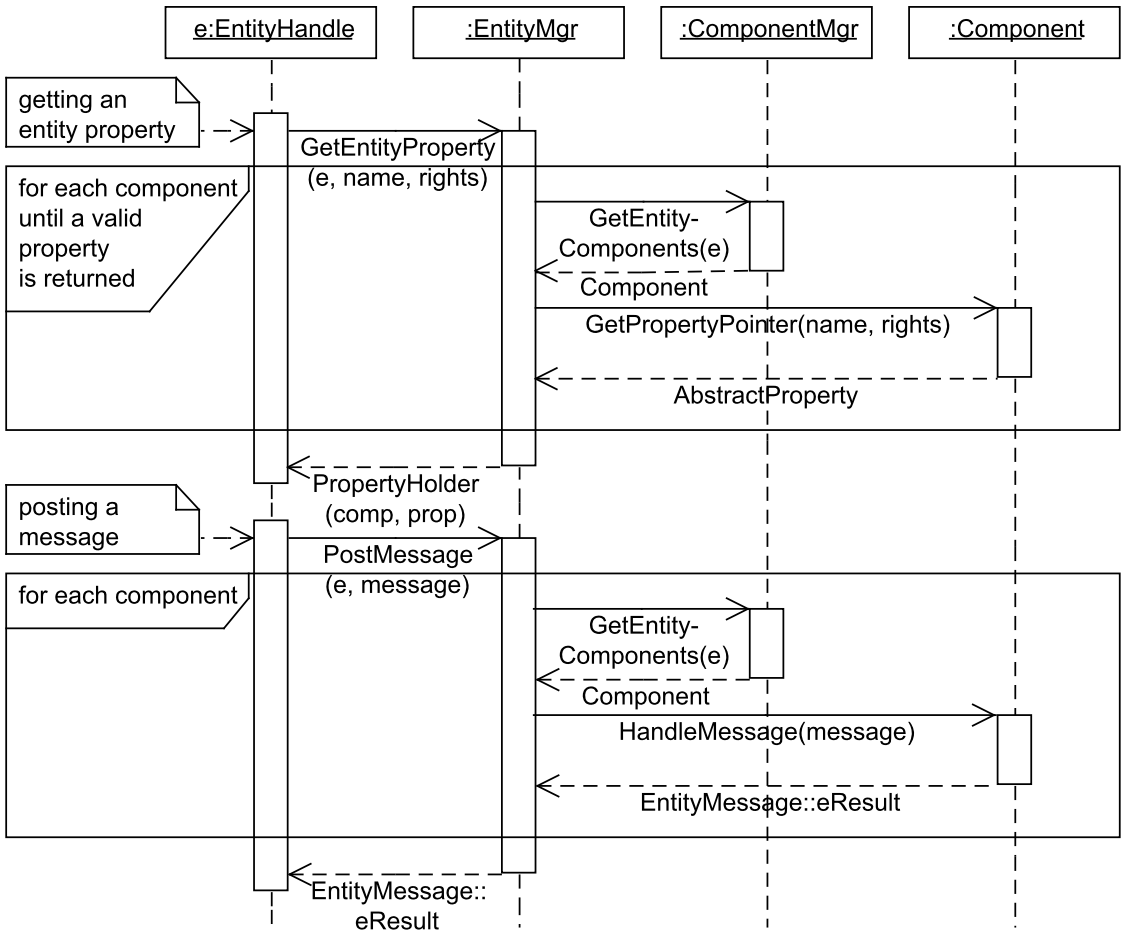

Figure 3.3: Process of getting an entity property and sending a message to an entity

3.4 Layer manager

Every entity with the *Transform* component has the layer property which is an ID of a layer from the layer manager implemented by the *EntitySystem::LayerMgr* class. This class has many methods for creating, moving and destroying layers as well as getting and setting their names and visibility, entities in a specified layer and choosing the current active layer. There are also methods for loading and saving stored information from/to a file.

There is always one initial layer with the ID equal to 0 which cannot be deleted and other layers are either in front (foreground, positive ID) or behind (background, negative ID) it.

3.5 Glossary

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

Entity property – a named pair of a getter and a setter function of a specific type with certain access rights

Entity function – a named link to a function with a *Reflection::PropertyFunctionParameters* parameter and certain access rights

Entity message – a structure that stores a message type from EntityMessageTypes.h and message parameters

Component – a class which has registered functions and properties, that can be read and written via their getters and setters, and which can handle received messages

Entity – a compound of one or more components, that provide specific functionalities, represented by a unique ID, it is possible to post a message to it

Prototype – changes of shared property values of this entity are propagated to the linked entities

Entity picker – a mechanism to select one or more entities

Layer – a number which defines a z-coordinate of an entity in a scene

Chapter 4

Gfx system

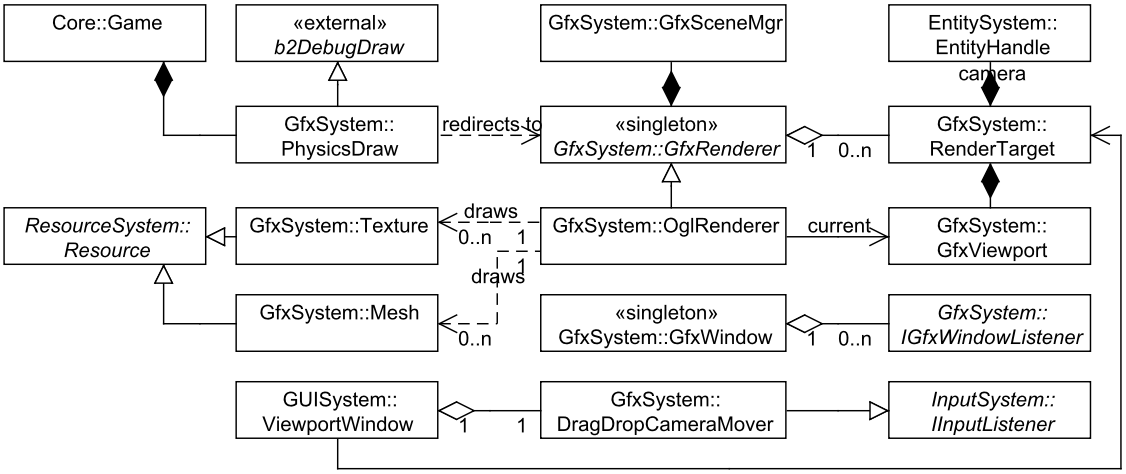

Figure 4.1: Class diagram of the GfxSystem namespace

The graphical system implements functionalities related to the rendering of game entities and the management of the application window. Note that the GUI system uses its own rendering system.

4.1 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 of the viewport within the global window. It can also be obtained from a texture. As an extra feature it carries data necessary to draw a grid used 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 and where they are rendered, so the *GfxSystem::RenderTarget* type is defined. It's 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 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.

4.2 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.

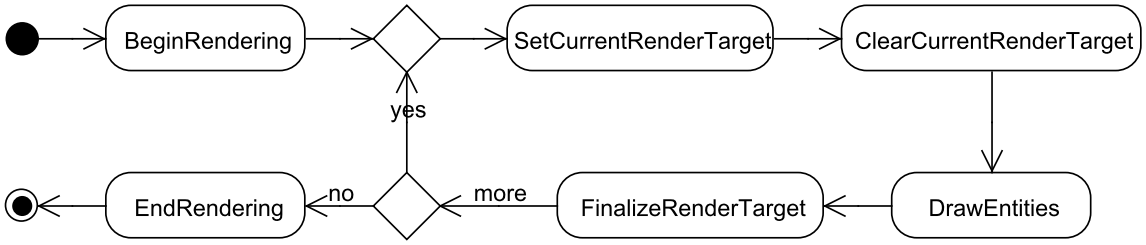

Figure 4.2: Activity diagram of the rendering process

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. As the figure 4.2 shows, the rendering must be started by the *GfxRenderer::BeginRendering* method, and 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.

4.3 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 are the window management and creating the rendering context.

4.4 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 common texture formats into the OpenGL.

For defining meshes, the Wavefront OBJ file format [20] is used with content of all material files (.mtl extension) at the beginning of the file with the .model extension. Every texture used in the model definition is automatically loaded as a resource.

4.5 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

Chapter 5

GUI system

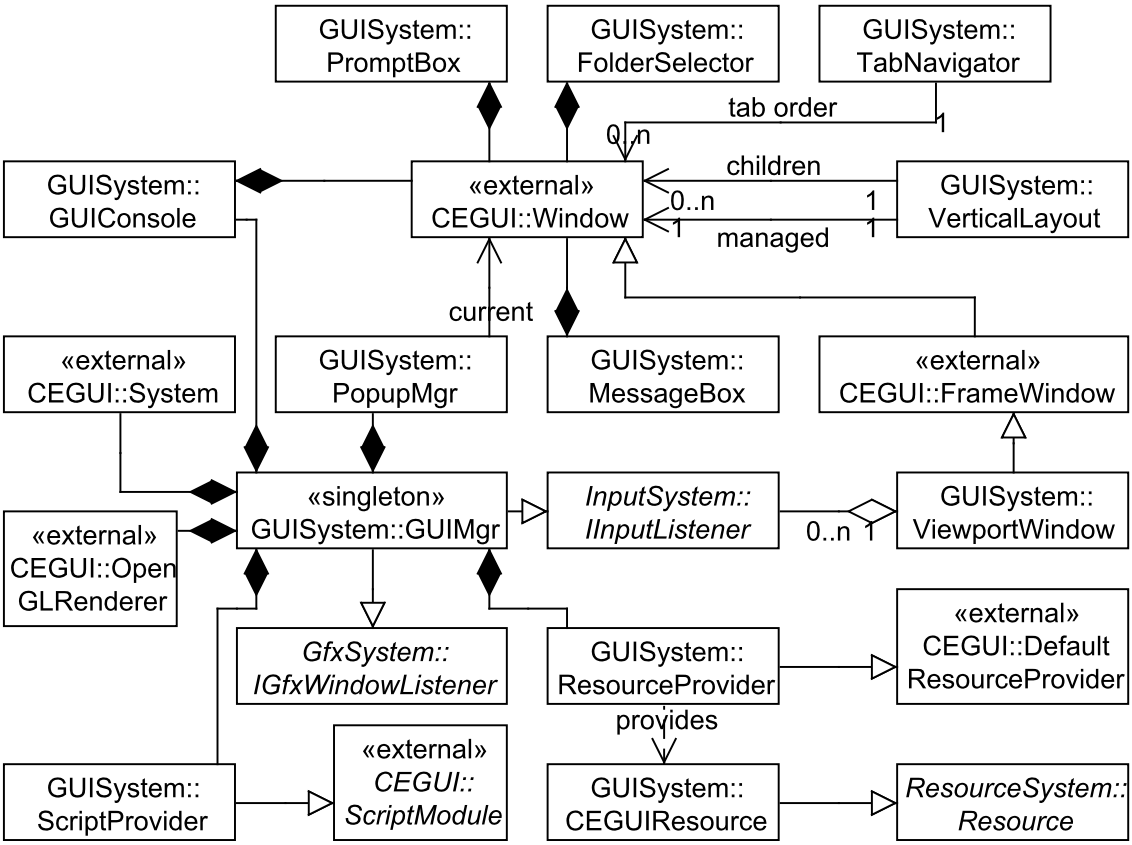

Figure 5.1: Class diagram of the GUI system namespace

The GUI system manages the graphical user interface. It is based the CEGUI library. In the engine it is used to implement the editor but the game uses it to display its own gameplay interface as well. The GUI system also takes care about the user input and how it interacts with GUI elements, element layouts, viewports and the GUI console.

5.1 GUI manager

The main class of the GUI system is *GUISystem::GUIMgr*, which is a connector for drawing, input handling and resource and script providing between the CEGUI library and the engine. On its initialization in the method *GUIMgr::Init*), it

- creates the CEGUI renderer,
- connects the resource manager with the CEGUI system via the *GUISystem::ResourceProvider* class (see section 5.2),
- connects the script manager with the CEGUI system via the *GUISystem::ScriptProvider* class (see section 5.3),
- provides itself as an input and screen listener,
- creates popup menu manager (see section 5.6) and
- creates the GUI console, which is then accessible via the *GUIMgr::GetConsole* method (see section 5.7).

In the method *GUIMgr::InitResources* it loads necessary GUI resources (schemes, image sets, fonts, layouts and looknfeels).

For loading a layout (see section 5.4) from a file, the *GUIMgr::LoadSystemLayout* and *GUIMgr::LoadProjectLayout* methods are defined. The first parameter of both methods specifies a name of a file where a layout is defined, the second one is a prefix given to all window names created by this layout. They provide a translation of all texts via the string manager. They differ in the root path and in the used string manager (the system or the project one). Similar methods are defined for loading a scheme and an imageset. For getting and setting the root layout of the application window, the *GUIMgr ::GetGUISheet* and *GUIMgr::SetGUISheet* methods are defined.

A window of any type can be created by the *GUIMgr::CreateWindow* method, got by the *GUIMgr::GetWindow* one and destroyed by the *GUIMgr::DestroyWindow* one. These methods need a name of a window as a parameter, the first one needs a type name in addition.

There are two methods called in the application main loop. First the *GUIMgr::Update* updates time of the GUI system, and then the *GUIMgr::RenderGUI* draws the whole GUI. There are several input callback methods that converts an OIS library representation of keyboard and mouse events to a CEGUI one and forwards them to the CEGUI library. It is possible to get the currently processing input event by the *GUIMgr ::GetCurrentInputEvent* method. There is also a callback method for a resolution change that forwards this information to the CEGUI library too.

5.2 GUI resources

A GUI resource is represented by the *GUISystem::CEGUIResource* class. Since the CEGUI library is not designed to allow an automatic resource unloading and reloading on

demand this class only loads raw data by the resource manager and after providing them to the CEGUI library by the *CEGUIResource::GetResource* method, it unloads them.

When the CEGUI library needs a resource, it calls an appropriate method of a resource provider class provided on an initialization of the library. In this engine it is the *GUISystem::ResourceProvider* class and its method *ResourceProvider::loadRawDataContainer* that gets the resource from the resource manager and forwards its data to the library. In the figure 5.2 there is an example of loading a GUI layout of the *GUISystem ::MessageBox* class.

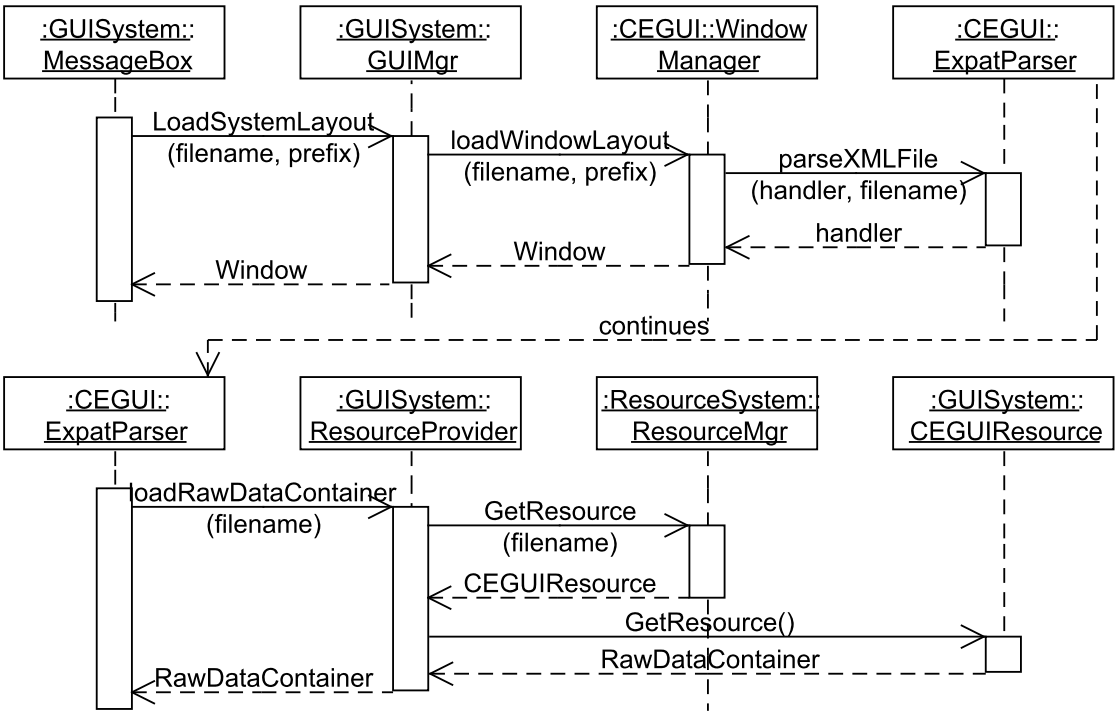

Figure 5.2: An example of loading a GUI layout

5.3 Script provider

For a connection of the CEGUI library and the script system the *GUISystem::ScriptProvider* class is introduced implementing the *CEGUI::ScriptModule* interface. The only method from this interface that truly needs an implementation is the *ScriptModule::subscribeEvent*, which provides a name of event, a name of function that should handle it and an object to which the name of event and an object with a callback method should be subscribed.

This transformation is implemented by the *GUISystem::ScriptCallback* class, that stores the function name given in its constructor and that calls an appropriate script function as a callback. It calls a function from the script module associated with the GUI

layout component of the layout, which gets the event or a function from the `GuiCallback.as` module as a default. For more information see the User Guide document.

5.4 Layouts

A GUI layout defines a composition of GUI elements including their properties such as position, size and content and their behavior. Their properties can be defined in an external XML file, but more dynamic compositions need also a lot of a code support. Their behavior can be defined in an external script file, but more complicated reactions have to be also native coded.

As an example that can be used both in the editor and in the game, the *GUISystem ::MessageBox* class was created, which provides a modal dialog for informing the user or for asking the user a question and receiving the answer. The basic layout with all possible buttons is specified in an XML file, which is loaded in the class constructor, where these buttons are mapped to the correspondent objects and displayed according to a message box type. Setting of a message text (*MessageBox::SetText*) also changes a static text GUI element specified in an XML file. The behavior after the user clicks to one of buttons is defined by a callback function that can be registered by the *MessageBox:: RegisterCallback* method and that gets the kind of the chosen button and the ID specified in the constructor parameter. For an easier usage there is the global function *GUISystem ::ShowMessageBox* that takes all necessary parameters (a text, a kind of a message box, a callback and an ID) and creates and shows an appropriate message box. A similar concept has the *GUISystem::PromptBox* class providing a modal dialog that asks for a text input from the user and the *GUISystem::FolderSelector* class providing a modal dialog for selecting a folder.

Another example is the *GUISystem::VerticalLayout* class that helps to keep GUI elements positioned in a vertical layout and automatically repositions them when one of them changes its size. In its constructor, the container in which all child elements should be managed is specified, and then the *VerticalLayout::AddChildWindow* method is used for adding them. It is also possible to set spacing between them, and there is a method for updating a layout. It is obvious that this layout is defined without any XML file. For more information about creating own layouts see the User Guide document.

5.5 Viewports

The *GUISystem::ViewportWindow* class represents a viewport window with a frame, where a scene is rendered by the graphic system. For defining a position, an angle and a zoom of a view of a scene that will be displayed in the viewport, a camera in form of an entity with a camera component must be set by the *ViewportWindow::SetCamera* method. It is possible to define whether the viewport allows a direct edit of a view and displayed entities by the *ViewportWindow::SetMovableContent* method. For example in the editor, there are two viewports – in the bottom one, the scene can be edited whereas in the top one the result is only shown. The method *ViewportWindow::AddInputListener* registers the input listener so any class can react to mouse and keyboard actions done in the viewport when it has been activated by the method *ViewportWindow::Activate*.

5.6 Popup menus

The *GUISystem::PopupMgr* class provides methods creating (*PopupMgr::CreatePopupMenu / PopupMgr::CreateMenuItem*) and destroying (*PopupMgr::DestroyPopupMenu / PopupMgr::DestroyMenuItem*) popup menu and its items as well as showing (*PopupMgr ::ShowPopup*) and hiding (*PopupMgr::HidePopup*) it and it also cares about calling a proper callback when a menu item is clicked on. The callback method is provided to the manager, when the popup menu is being opened by the *PopupMgr::ShowPopup* method.

5.7 GUI console

The *GUISystem::GUIConsole* class manages the console, which is accessible both in the game and in the editor. The console receives all messages from the log system via the method *GUIConsole::AppendLogMessage* and shows those that have an equal or higher level than the one previously set by the *GUIConsole::SetLogLevelTreshold* method. In addition, the user can type commands to the console prompt line, which are sent to the script system as a body of a method without parameters that is immediately built and run. If there is a call of the `Print` function, its content will be printed to the console via the *GUIConsole::AppendScriptMessage* method. For better usage of the console a history of previously typed commands is stored and can be revealed by up and down arrows. The console can be shown or hidden by the *GUIConsole::ToggleConsole* method.

5.8 Glossary

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

GUI – a graphic user interface

GUI element – an element from which the whole GUI is created such as label, edit box, list etc.

Layout – a composition of GUI elements including definition of their properties and behavior

Viewport – a GUI element where a scene can be rendered by the graphic system

GUI console – a window where log and script messages immediately appears and which allows an input of script commands

GUI event – a significant situation made by the user input such as a mouse click or a keyboard press

Callback – a function or method that is called as a reaction to a GUI event

Scheme – a definition of connections between a physical window type and a window look

Chapter 6

Editor

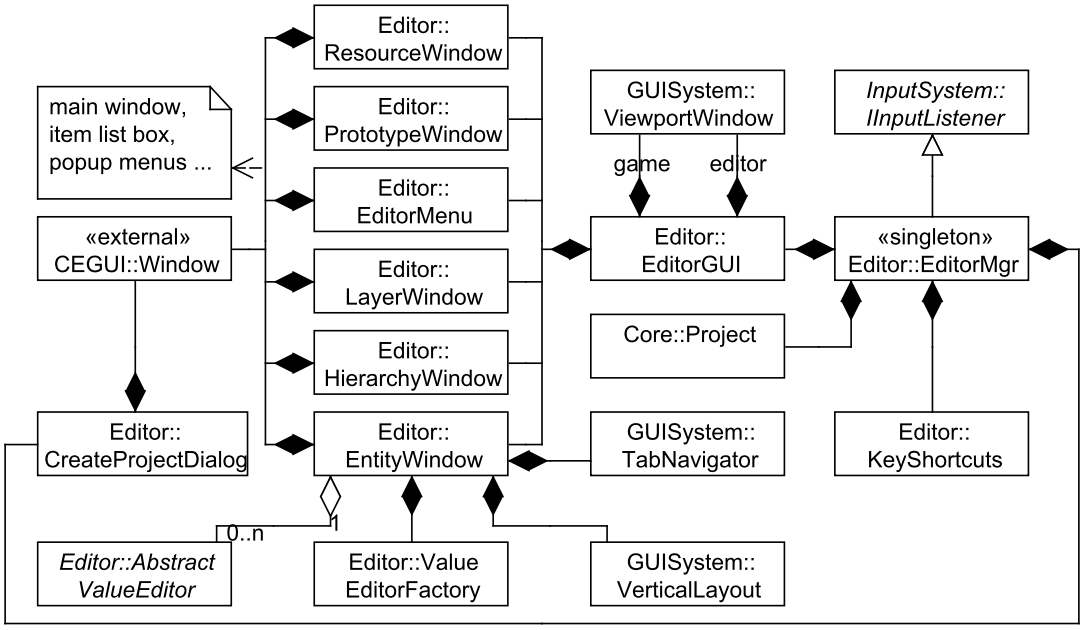

Figure 6.1: Class diagram of the Editor namespace

The editor is a graphical user interface allowing developers to visually create games using the game engine. It provides GUI elements for managing everything from the whole project to each entity in every scene. The main advantage is that every change made to a scene is immediately visible in the game window where the game action can be started anytime.

6.1 Editor manager

The *Editor::EditorMgr* class manages the logic of the editor while the *Editor::EditorGUI* class takes care of the GUI elements. The latter is instantiated by the former. Both of the classes have methods called by the application when the editor is loading or unloading.

They also have methods to update the logic of the editor as well as methods to draw itself. These are called from the main loop of the application.

The first class has methods for managing projects, scene and entities. It manages selecting entities and editing the current one, it reacts on choosing all items in the main menu such as creating a new entity, duplicating and deleting current or selected entities, adding and removing entity components, creating a prototype from a current entity, creating or loading a project or a scene and it solves changing an entity name or an entity property. It also provides a function for all edit tools such as moving, rotating or scaling of a chosen entity and for resuming, pausing and restarting the game action. Finally, there are methods for getting a reference to all editor windows and for updating them.

The second class compounds all editor GUI layouts such as the game and editor viewports, the entity, resource, prototype and layer views and the editor menu and it initializes and updates them.

6.2 Editor views

There are five classes that provide five views in the editor. The *Editor::EntityWindow* class manages the Entity view.

There are seven layout classes for editor components. All of them have initialization and update methods and callbacks for significant events and load their look from an external XML file and introduce the reactions on events and loading data from the application. The *Editor::CreateProjectDialog* represents the dialog for creating a new project, the *Editor::HierarchyWindow* manages the hierarchy of entities, the *Editor::LayerWindow* manages the layers the entities are put to, the *Editor::PrototypeWindow* manages the prototypes of entities, the *Editor::EntityWindow* manages components and properties of entities, the *Editor::ResourceWindow* manages the resources that the entities can use and the *Editor::EditorMenu* represents the main menu and toolbar.

6.3 Value editors

All value editors derive from the *Editor::AbstractValueEditor* base class (see the figure 6.2). This class has four abstract methods that every value editor must implement:

- The *AbstractValueEditor::Update* method is called when the current value of the property should be displayed in the value editor.
- The *AbstractValueEditor::Submit* method is called when the value of the property should be updated by the user input to the value editor.
- The *AbstractValueEditor::GetType* method must return the type of the implemented value editor.
- The *AbstractValueEditor::DestroyModel* method must implement the destruction of the corresponding model.

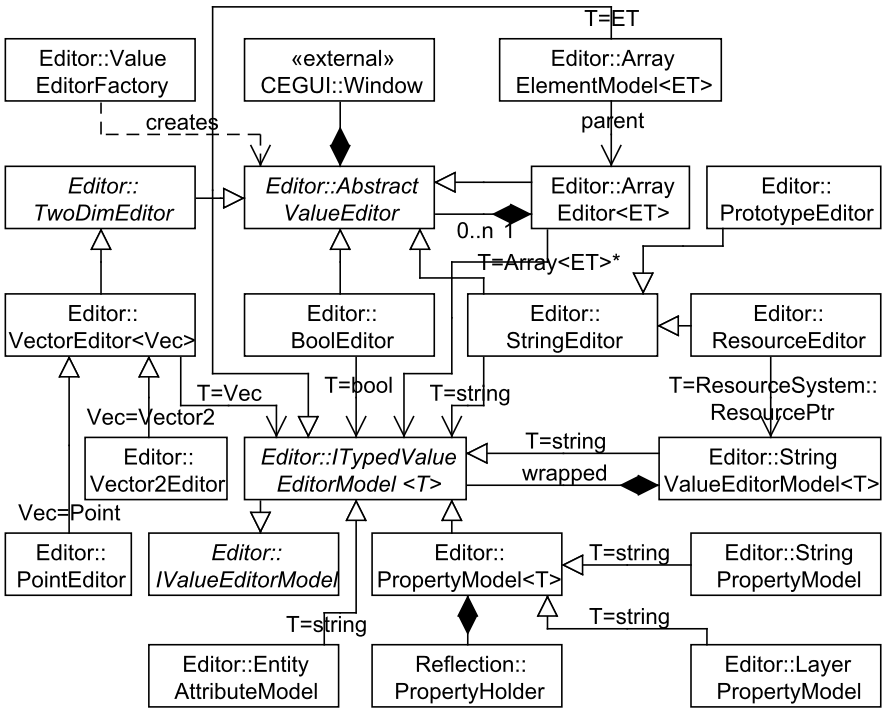

Figure 6.2: Class diagram of editor value editors

In the figure 6.3 there is an example how calling the first two methods is proceeded. There are also some methods that can help with the implementation in this class. On its initialization, each editor should create a widget for editing a required kind of value and set it to the *AbstractValueEditor::mEditorWidget* protected attribute.

For an easier implementation of value editors the model classes with the *Editor::IValueEditorModel* class as the base class were created. From the accessors of this class value editors should get the name and tool-tip for the edited property, find out whether the property is valid, read only, an element of a list, removable, shareable, shared and also they should be able to remove the property or set whether it is shared if is possible. It is recommended to derive own models from the *Editor::ITypedValueEditorModel<T>* template class, that also defines the getter and setter for the value of the edited property.

For example the *Editor::StringEditor* class derives directly from the *Editor::AbstractValueEditor* class and it implements a simple editor for properties which values are easily convertible from and to string. It contains a label with a property name, an edit box for displaying and editing the value and a remove button if the property is removable (i.e. as a part of an array). It uses a model derived from the *Editor::ITypedValueEditorModel<string>* class to setup a widget, update and submit the value which is specified in the constructor parameter. A useful implementation of this model is the *Editor::StringPropertyModel* class that operates with the *Reflection::PropertyHolder* class from which it gets all necessary information and which it can modify with a new value.

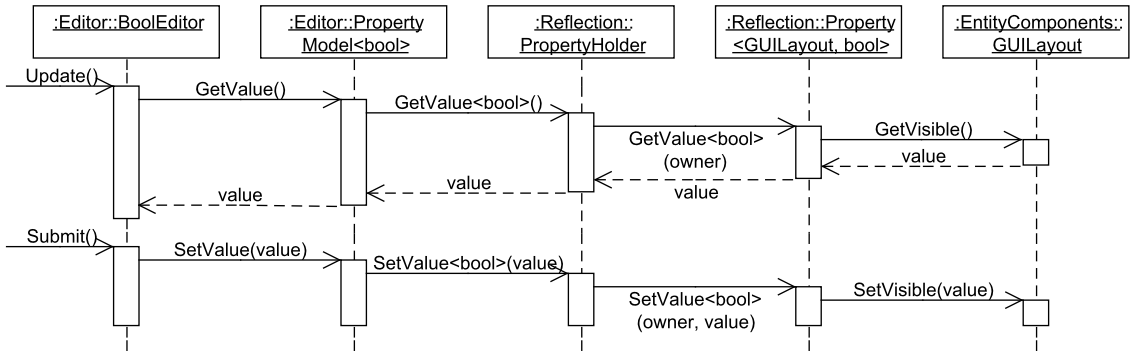

Figure 6.3: Sequence diagram of getting and setting property value in the editor (the boolean property *Visible* of the *GUILayout* component is used as an example)

There are another examples of value editors and its models in the code that can help with a creation of further ones such as editors for arrays, resources etc.

Chapter 7

Input system

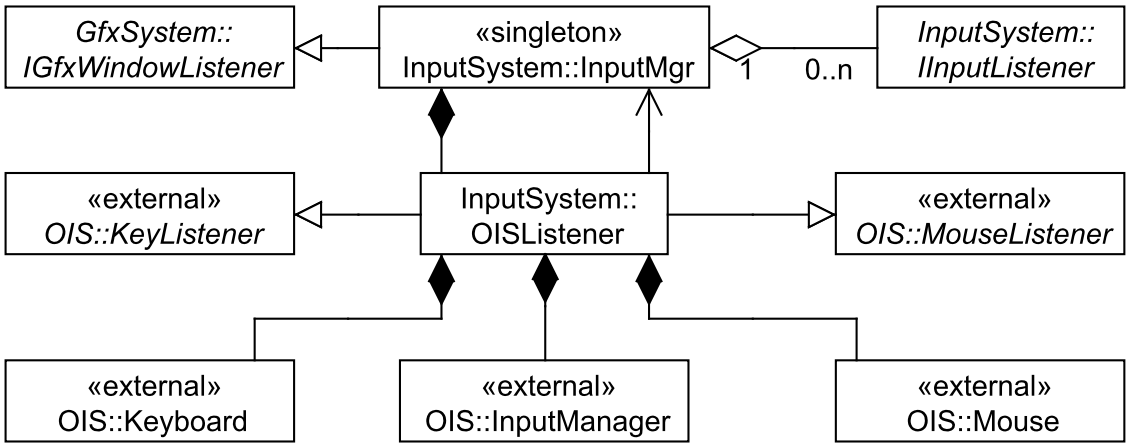

Figure 7.1: Class diagram of the Input system namespace

Since every game must somehow react to the input from the player, the input system, which handles keyboard and mouse events and forwards them to other systems, was implemented.

7.1 Input manager

The needs of games are different, but the ways they want to access the input devices are still the same. Either they want to receive a notification when something interesting happens or they want to poll the device for its current state. The *InputSystem::InputMgr* class implements both of these approaches.

The first one is provided via the *InputSystem::IInputListener* interface, which should be implemented and then registered by using the *InputMgr::AddInputListener* method for receiving the notification in callbacks of the interface.

The second approach is supported by multiple methods for querying the device state. The *InputMgr::IsKeyDown* method returns whether any specific key is currently held

down, while the *InputMgr::IsMouseButtonPressed* method does the same with the currently pressed mouse button. Finally, the *InputMgr::GetMouseState* returns a whole bunch of mouse related information such as a cursor and wheel position or pressed buttons.

To keep things synchronized, the event processing is executed in the main game thread. At the beginning of the application main loop, the *InputMgr::CaptureInput* method is called, where the events are recognized and then distributed to the listeners.

The *InputSystem::InputMgr* class is only a proxy class that calls methods of the implementation specific class *InputSystem::OISListener*, which implements the *OIS::MouseListener* and *OIS::KeyListener* interfaces from the OIS library used for the platform independent input management.

Chapter 8

Log system

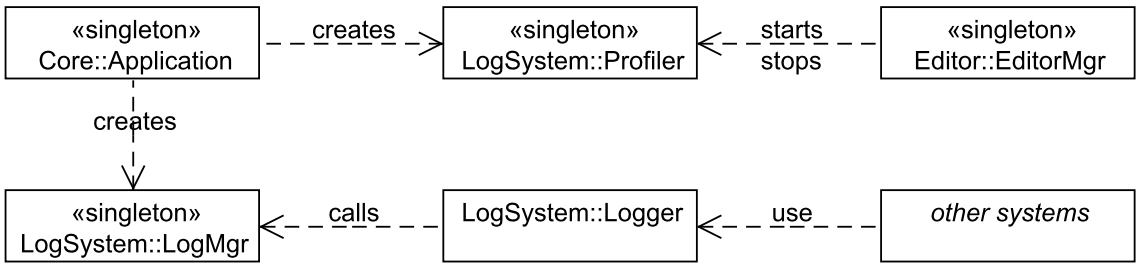

Figure 8.1: Class diagram of the Log system namespace

The log system manages internal log messages that are used for providing informations about application processes. It is useful for debugging the project. These messages can have various levels of a severity (from trace and debug messages to errors) and it is possible to set the minimal level of messages to show (i.e. only warning and errors). Another function of the log system is managing a real-time in-game profiling, useful for a location of the most time critical parts of the project that help to optimize the application effectively.

8.1 Logging messages

The main class responsible for logging messages is the *LogSystem::LogMgr*. At the application start it is initialized with a name of the file to which the messages are also written. There is only one method that logs a message of a certain severity to the file and consoles if exist. This method should not be used directly but via the class *LogSystem::Logger*.

The lifetime of the *Logger* should be only one code statement and it represents one message. In the constructor, the level of the message and whether to generate a stack trace are specified. Then a sequence of the operator << is used to build the message from strings, numbers and other common types (any user type can be supported by specifying an own operator << overloading). At the end of the statement, the destructor is automatically called (if the instance is not assigned to a variable).

For an even easier logging, macros are defined for every supported level of severity, adding the information about the file and the line, where it is logged from in case of error or warning message. Thanks to macros, it is possible to define the minimum level of severity that should be logged at the compile time, so the messages with a lower level are even not compiled to the final program, which saves time and memory. In the table 8.1, there are the macros associated with the levels of severity.

Macro	Level	Stack trace	Additional info
ocError	error	yes	yes
ocWarning	warning	no	yes
ocInfo	information	no	no
ocDebug	debug	no	no
ocTrace	trace	no	no

Table 8.1: Definitions of log macros from the most severe to the least ones

If it is for example necessary to inform that the entity (of which the handle is available) is created, the following statement should be written anywhere in the code: `ocInfo << handle << " was created.";`. When the process passed this code and the minimum level of messages to log is lower than the information one, then for example, the following message will be generated: `13:05:18: Entity(25) was created`.

8.2 Profiling functions

The profiling of a block of a code or a whole function is really easy. First the `USE_PROFILER` preprocessor directive must be globally defined, the class *LogSystem::Profiler* must be initialized at the start of the application and its method *Update* must be called in each application loop.

Then anywhere in a code the `PROFILE(name)` macro can be typed where the parameter `name` is used for identification of the corresponding results (the abbreviation `PROFILE_FNC()` uses the current function name). From that line the profiler will start measuring time and it will stop at the end of the current block or function.

Finally, when the application runs a call of *Profiler*'s method *Start* (which can be invoked with a keyboard shortcut *CTRL+F5*), the profiling is activated and a call of its methods *Stop* and *DumpIntoConsole* (another press of *CTRL+F5*) deactivates it and writes results to the text console. In addition, it is possible to ask whether the profiler is active via the method *IsRunning*.

8.3 Glossary

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

Log message – a text describing a specific action or an application state occurred at a certain time with a defined severity

Level of severity – an importance of a message to the application process

Stack trace – a list of functions that the current statement is called from right now

Logging – tracking the code execution by writing log messages to a console and a file

Console – a window where log messages immediately appear

Profiling – measuring a real time of execution of a specific code

Chapter 9

Resource system

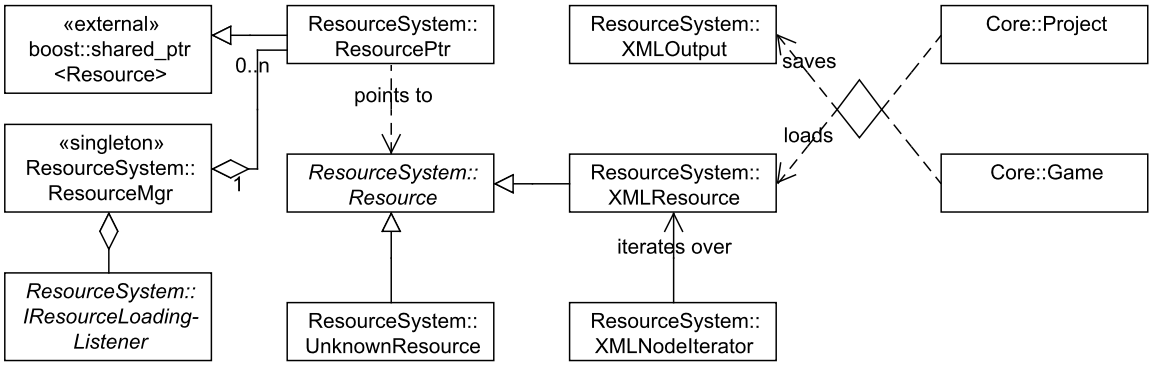

Figure 9.1: Class diagram of the Resource system namespace

Every game needs to load packs of data from external devices such as a hard drive or network. The data come in blocks belonging together and representing a unit of something usually called *resource*. Since the games work with loads of resources it is necessary to organize them both in-game and on the disk. Also, the data loaded are usually quite large and it's necessary to free them when possible to save memory. All of these tasks are implemented in the resource system as well as loading and saving scenes.

9.1 Resources

The resource is a group of data belonging together. In the engine this is represented by the abstract *ResourceSystem::Resource* class. It contains all basic attributes of a resource and allows its users to load or unload it, but the actual implementation depends on the specific type of the resource. For example, an XML file is loaded and parsed in a different way then an OpenGL texture. However, for the user it is never really necessary to know what type of the resource he is working with, so using this class as an abstraction is enough. Only the endpoint subsystem needs to work with the specific type to be able to grab the parsed data out of it. For example, the texture resource can be carried around the system as a common *ResourceSystem::Resource* class until it reaches the graphical

subsystem, which converts it to the texture resource and grabs the implementation specific texture data out of it.

9.1.1 Resource states

Each resource can be in one of the states described in the figure 9.2 at the given point of time:

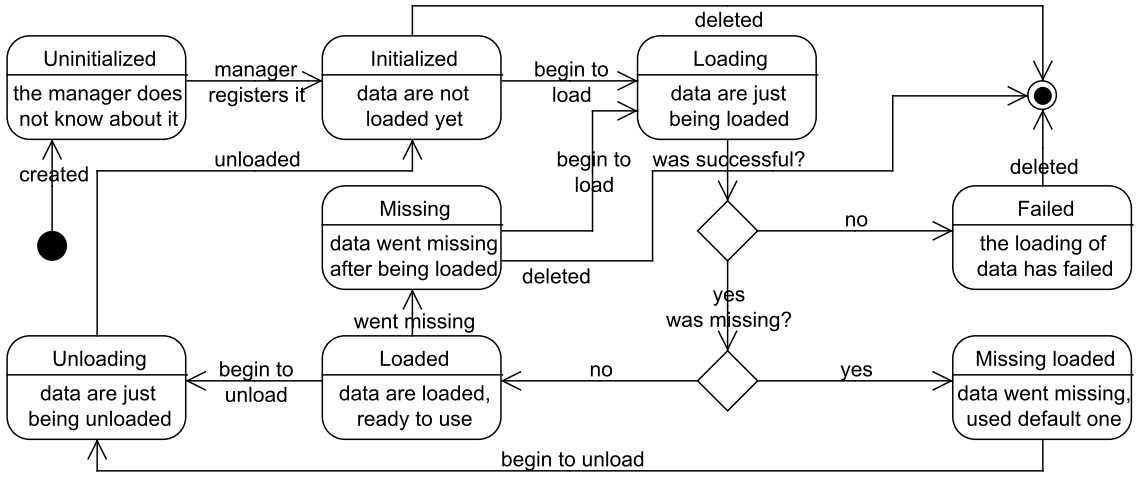

Figure 9.2: Possible resource states and their description

9.1.2 Content of the resource

Once the data for the resource are loaded, it must be parsed into the desired format. This can mean a data structure stored directly inside the resource class or just a handle to the data stored in other parts of the system. However, both of these must exist only in single instance in the whole system – in the resource which parsed the data. Otherwise the data could become desynchronized. If the resource was unloaded, a pointer to its data could still exist somewhere.

For example, the XML resource creates a tree structure for the parsed data and allows its users to traverse the tree. But nowhere in the system a pointer to the same tree or any of its parts exists. Another example is a texture. After it loads data, they are passed into the graphical subsystem, which creates a platform specific texture out of it and returns only a texture handle. This handle is then stored only in the resource.

9.1.3 Resource pointers

Since the resources are passed all around the game system, we must somehow prevent any memory leaks from appearing. Doing so is quite easy however – we simply use the shared pointer mechanism. It points to the common abstract resource (the *ResourceSystem::*

ResourcePtr class) and to specific resources as well (the *ScriptSystem::ScriptResourcePtr* or the *ResourceSystem::XMLResourcePtr* classes for example). The abstract resource pointer can be automatically converted to any specific resource pointer, but if the type does not match, an assertion fault will be raised to prevent a memory corruption. All resource pointers are defined in the `Utils/ResourcePointers.h` file.

For defining a new resource type, see the Extending Ocerus document. In the table 9.1 there are all currently existing types of resources with a brief description.

Resource type	Description and supported extensions
Texture	a bitmap picture (jpg, png, bmp, psd, dds, hdr, tga)
Mesh	a 3D object (model)
CEGUI	a GUI related resource (layout, scheme, font, imageset, looknfeel, ttf)
Text	a text in a specific language (str)
XML	an XML document (xml)
Script	a file with AngelScript script functions (as)

Table 9.1: List of defined resource types with a brief description

9.2 Resource manager

The resource manager represented by the *ResourceSystem::ResourceMgr* class takes care of organizing resources into groups and providing an interface to other parts of the system to control or grab the resources. Coupling resources into groups makes it easier to load or unload a whole bunch of them. There are methods for adding one file or a whole directory to a resource group or for getting a resource pointer to a resource of a specific name from a specific group.

To make game development easier, the source of each resource is automatically checked for an update. If it has changed, the resource is automatically reloaded if it was previously loaded. So, for example, if the user changes a currently loaded texture in an image editor and saves it, it will be immediately updated in the game.

Since gaming systems have limited memory it is possible to limit the memory used by resources. Resources usually take the biggest chunk of memory, so when lowering the memory usage it is best to start here. Hopefully, the resource manager allows defining a limit which it will try to keep. When the memory is running out, it will attempt to unload resources that were not used for a long time. These resources will remain in the system and will be ready to be loaded as soon as they are needed. However, there are certain circumstances under which the resources must not be unloaded. An example of such is rendering – no texture can be unloaded until the frame is ready. For this reason the unloading can be temporarily disabled.

If it is necessary to track the resource loading progress (for example when a scene is loading), there is the *ResourceSystem::IResourceLoadingListener* interface that should be implemented and registered by the *ResourceMgr::SetLoadingListener* method. Its methods are called when the loading of one resource or a resource group starts and ends.

9.3 Loading and saving scenes

For storing a state of a scene from the editor or from the game itself to a file, an XML format is used. There are two classes that help with saving data to a file and loading them back.

The *ResourceSystem::XMLOutput* class provides a formatted XML output to a file. First when the instance is created, a file specified in a constructor parameter is opened. Then some methods for writing XML elements and attributes are used. Finally the file is closed in the destructor or in the method *XMLOutput::CloseAndReport*. For example the file named `file.xml` with the XML structure

```
<?xml version="1.0" encoding="UTF-8"?>
<name key="value">
  <element>text</element>
</name>
```

is created by calling this sequence of orders:

```
ResourceSystem\::XMLOutput out("file.xml"); // writes header
out.BeginElementStart("name"); // writes <name
out.AddAttribute("key", "value"); // writes key="value"
out.BeginElementFinish(); // writes >
out.BeginElement("element"); // writes <element>
out.WriteString("text"); // writes text
out.EndElement(); // writes </element>
// destructor automatically closes all open elements
```

As the example shows the class does an indent, remembers names of open elements and automatically closes all open elements in the end.

The *ResourceSystem::XMLResource* class on the other hand loads an XML file and iterates over its elements and attributes. Since this class derives from the *ResourceSystem ::Resource* class first a resource pointer to the file must be get by *ResourceMgr::GetResource* method and retyped to the *ResourceSystem::XMLResourcePtr*. Then the top level elements can be iterated by the *XMLResource::IterateTopLevel* method, which returns *ResourceSystem::XMLNodeIterator* class, which serves as an iterator and which should be compared with a result of the *XMLResource::EndTopLevel* method for a detection of the end of top level elements. The iterator class has analogical methods (*XMLNodeIterator ::IterateChildren* and *XMLNodeIterator::EndChildren*) for iteration over children of the element it represents. Beside them it has also methods for getting an element's value, a value of its child or of its specific attribute. Since these methods are templates every value can be converted from a string to any chosen data type.

9.4 Glossary

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

Resource – a unit of data usually stored in an external device the game will be working with as a whole

Resource pointer – a shared pointer to a resource that can be used in the whole engine

Resource type – resources with a common type are loaded in the same way, i.e. textures, models, scripts, texts...

XML – Extensible Markup Language (XML) is a set of rules for encoding documents in machine-readable form

Chapter 10

Script system

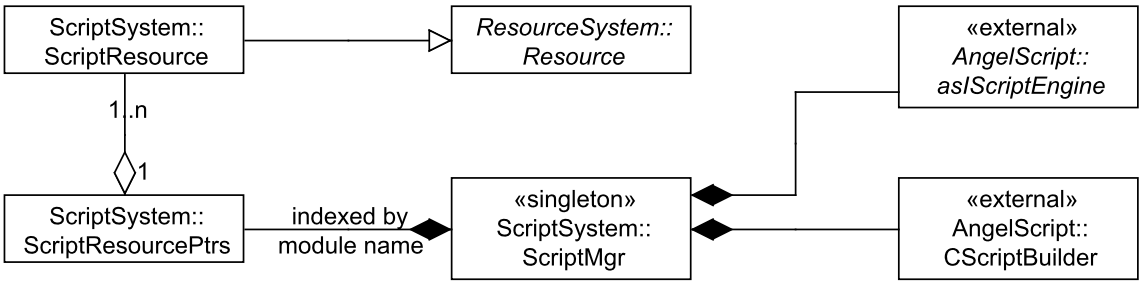

Figure 10.1: Class diagram of the Script system

The script system allows customizing reactions to application events such as messages sent to entities or GUI interactions without the recompilation of the whole application. The advantage of using scripts is that it's much easier to extend the application. Also, it can be done even by non-professional users because in the script environment they cannot do any fatal errors that could corrupt the application. Also, the engine controls the script execution – for example, there's a timeout of execution preventing cycling. The disadvantage is slower execution of scripts than a native code, so the user should decide which part of the system will be the native coded and what can be done by scripts.

10.1 Interface of the script manager

The script manager is represented by the class *ScriptSystem::ScriptMgr* that encapsulates the script engine and manages an access to the script modules. This class is a singleton and it is created when the game starts. It initializes the AngelScript engine and registers all integral classes and functions (see the Script Reference documentation) as well as all user-defined ones.

The first thing a caller of a script function must do is to obtain a function ID. It can be returned from the method *ScriptMgr::GetFunctionID*, which needs the name of the module, where the function is, and its declaration. For example, if the name of a function to be called is `IncreaseArgument` and it receives one integer argument and returns also

an integer, the declaration of it will be `int32 IncreaseArgument(int32)`. This method returns an integer that represents the desired function ID (greater than or equal to zero) or the error code (less than zero), which means the function with this declaration could not be find in the mentioned module, or this module does not exist or cannot be built from sources (see log for further information). The function ID is valid if the module exists in memory, so it can be stored for later usage.

The function mentioned above calls *ScriptMgr::GetModule* to obtain desired module. This method returns the module from a memory or loads its sources from a disc and builds it if necessary. If it is inconvenient that the module is built at the first call of its function, the *ScriptMgr::GetModule* method can be called before to get a confidence that the module is ready to use.

The calling of script functions can be done with three methods. First the caller calls the *ScriptMgr::PrepareContext*, which needs the function ID and returns context prepared for passing the argument values. Then the argument values can be passed with the *SetFunctionArgument* method, which needs the prepared context as the first argument, a parameter index as the second one and a parameter value in form of the *PropertyFunctionParameter* as the last one. After that, the *ScriptMgr::ExecuteContext* should be called with the prepared context as the first argument and a maximum time of executing script in milliseconds (0 means infinity) to prevent cycling as the second one. This method executes the script with given arguments and returns whether the execution was successful. If so, it is possible to get a return value of function with corresponding methods of context. In the end the context should be released to avoid memory leaks.

There is a possibility to call a simple script string stored in a memory by the method *ScriptMgr::ExecuteString*, which accepts this string as the first parameter. The method wraps it to the function without parameters and builds it and calls it as a part of the module specified in the second optional parameter so it is possible to declare local variables and to call functions from the module. This is useful for example for implementing a user console.

As mentioned in the User Guide document, it is possible to use a conditional compilation of scripts. The method *ScriptMgr::DefineWord* adds a preprocessor define passed as the string argument. The *ScriptMgr::UnloadModule* and *ScriptMgr::ClearModules* methods unload one/all previously loaded and built modules. All function IDs and contexts associated with these modules will be superseded so the caller should inform all objects that holds them about it (use *ResourceUnloadCallback* in ScriptRegister.cpp for specify actions done when modules are unloaded). These methods could be called when it is necessary to reload modules or free all memory used by them, but when it is better not to destroy the whole script engine.

There are also two methods for getting time. The first one (*ScriptMgr::GetGameTime*) returns the game time, which does not increase when the game action is paused, whereas the second one (*ScriptMgr::GetGlobalTime*) returns the engine time, which does increase even if the game action is paused.

10.2 The Script component

The purpose of the Script component is to provide a possibility for an entity to respond to a received message by a script. When the message is received, it should find an appropriate handler, which is a script function with a strict declaration (see EntityMessageTypes.h for entity message handler declarations), in defined modules and call it with the arguments provided in the message data.

The component has five registered properties. The first one is an array of module files that are searched for message handlers. The second one represents the maximum script execution time in milliseconds, after which the script will be aborted. For better performance, this component caches function IDs founded in modules in a map, so it is necessary to inform it when the modules are going to change by sending a message `RESOURCE_UPDATE` to its entity.

The last three properties provide a support for scripts that should be called periodically and that should remember their state. These scripts are written to the `void OnAction()` message handlers, which are called when the message `CHECK_ACTION` is received (which should be every game loop) and when the appropriate time in the `ScriptTimes` property is lower than the current game time. From these scripts it is possible to call the `int32 GetState()` function, which returns the appropriate state from the `ScriptStates` property and to call the `void SetAndSleep(int32, uint64)` function, which sets this state and sets the time to the current game time plus the second argument in milliseconds. The right time and state are got thanks to the `ScriptCurrentArrayIndex` property.

The most important method of this component is the *Script::HandleMessage*, which accepts a message structure as parameter and returns if the message was processed well or it was ignored. First it checks whether the function IDs should be updated and if the message is `RESOURCE_UPDATE` it ensures to do an update before the next message processing. Then it gets an appropriate function ID, dependent on a message type, checks whether to continue in case of the `CHECK_ACTION` message and calls the script manager to prepare a new context with this function. After that the pointer to the parent entity is stored to the context data so the `this` property can be called from a script to get the current entity handle. Then it adds additional parameters to a function call from the message data according to the message type and executes the context with defined timeout. Finally, it releases the context and returns whether the execution was successful. An example of a reaction to a message is shown in the figure 10.2.

10.3 Glossary

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

Script function – a function defined and implemented in a script file

Script file – one file containing script class and script function definitions, can include a code from other files

Script module – one or more script files connected by include directives, which are managed and built together and have a common namespace

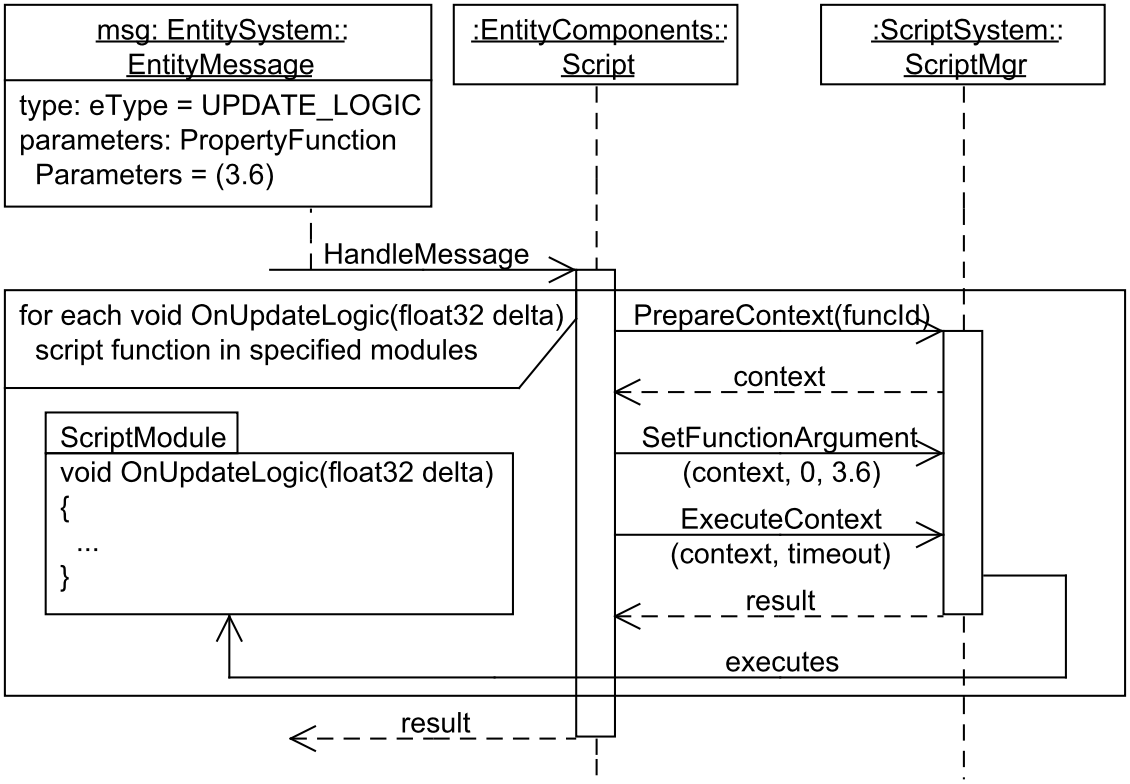

Figure 10.2: An example of reaction of the Script component to a message

Function ID – an identification of a script function based on a module name and a function declaration

Script context – an object that wraps script function calling, it must be prepared with a function ID, executed and released, function arguments can be passed and a return value can be obtained

Script engine – an object that registers C++ classes, global functions, properties etc. for usage in a script code and manages script modules and contexts

Script manager – a class that encapsulated a script engine and provides methods for managing script modules and calling script functions

Script component – a component that provides script callbacks to the messages for an entity

Chapter 11

String system

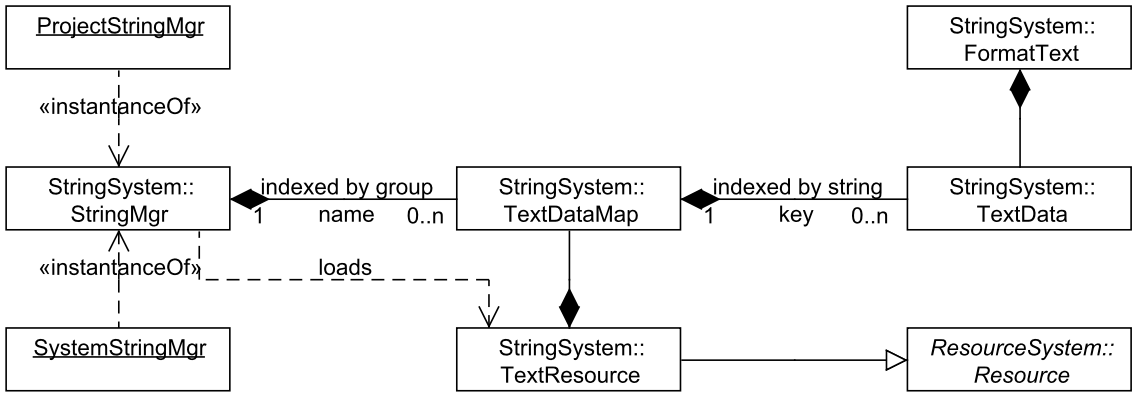

Figure 11.1: Class diagram of the String system namespace

The string system manages all texts that are visible to the user except internal log messages. It supports localization to various languages and their country-specific dialects and a switching among them on the fly.

11.1 Interface of the string manager

The string manager is represented by the class *StringSystem::StringMgr* that provides all necessary services. There are two instances of this class when the application runs. The first one manages system texts (i.e. labels in the editor, common error messages etc.). The second one provides an access to project specific texts, so they differs only in a path to the root directory where the text resources are loaded from. Both are initialized when the application starts and destroyed on the application shutdown. The class provides static methods that return these instances. Or it is possible to use defined macros.

The first method that is necessary to call before using the others is the *StringMgr:: LoadLanguagePack*, which expects the language and the country code. Both parameters can be omitted and their meaning is widely explained in the User Guide documentation. It removes old text items and loads new ones according to a specific language setting

to the memory via the *StringMgr::TextResource* class and divides them to the desired groups. This method can be called at any time during the whole execution of application but the other systems must refresh their data themselves. The accepted format of text files and necessary directory layout for loading them are described in the User Guide documentation.

There are several methods in the class for getting the text data according to its group name and key, which differs in returning a pointer to the data or the data itself and in specifying or omitting the group name (using the default one instead). If the text data of a specified group name and key does not exist, an empty string is returned and an error message is written to the log.

11.2 Using variable text

If the loaded text data contains character sequences in a form of `%x` (`x` is a number from 1 to 9), it is possible to replace them by another text using the class `StringSystem ::TextFormat`. Just construct the instance of this class with the loaded text as a parameter, then use a sequence of `<<` operators to replace all well formatted character sequences and store it to another text data variable.

The `<<` operator finds the described character sequence with the minimum number and replaces it with the text provided as the parameter. If no such sequence is found, it inserts the provided text at the end of the loaded text. For example if the code

```
StringSystem\::TextData loaded = "The %2, the %3 and the %1.";
StringSystem\::TextData result = StringSystem\::TextFormat
    (loaded) << "third" << "first" << "second";
```

is called then in the `result` variable there will be the following text:

```
The first, the second and the third.
```

11.3 Glossary

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

Key – an ID that is used for indexing a text data, must be unique within a group

Text item – a pair of a key and a text data that is contained it a text file

Group – a set of text items that can be indexed from application, one file contains text items from one group, text items from one group can be contained in several files

Language code – two-character acronym representing a world language according to ISO 639-1 [23].

Country code – two-character acronym representing a country according to ISO 3166-1 alpha-2 [24].

Chapter 12

Memory system

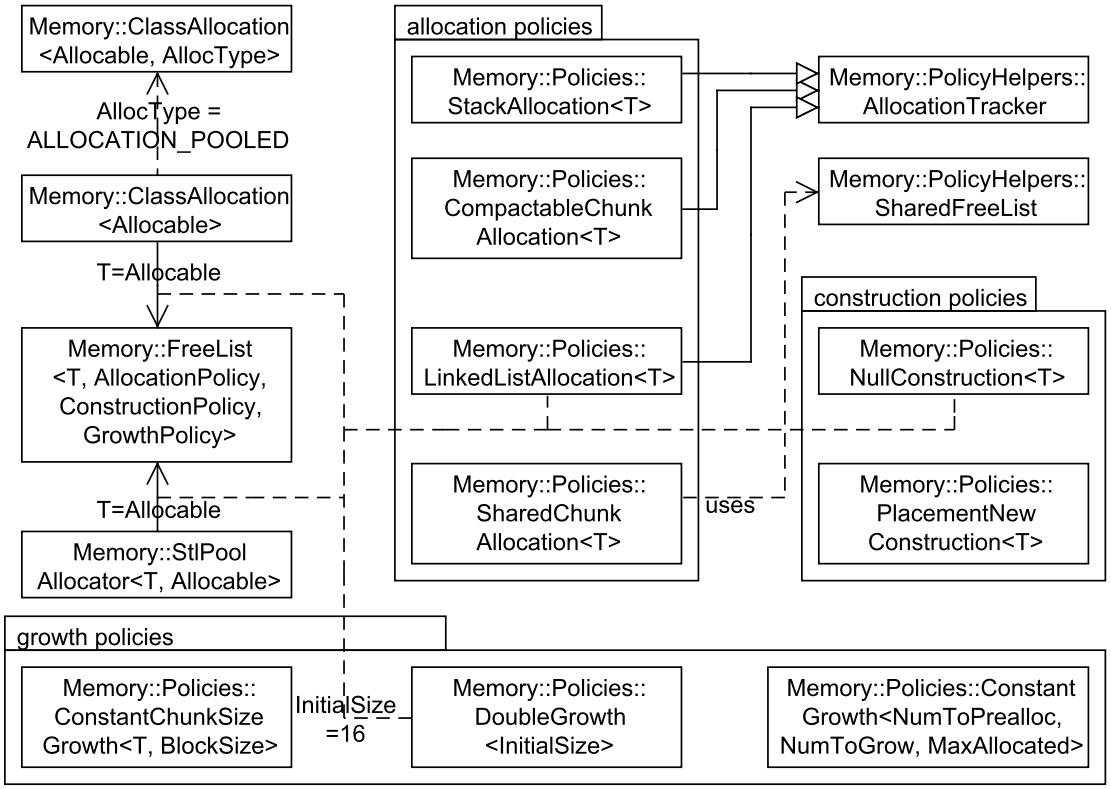

Figure 12.1: Class diagram of the Memory namespace

The memory system controls the memory allocation of the whole application. It attempts to work under the hood, so that the rest of the project does not need to know about it. It provides tools to override the default dynamic memory allocation as well as custom specialized allocators for small objects or containers.

Since games are very different from other applications, they also have specific memory requirements. However, the default managers in current compilers are targeted to common programs. The game is a real-time performance-intensive application while it also

allocates and deallocates a lot of objects on a regular basis. An example of this might be graphical effects spawned and destroyed during the play to keep the action high. In this case it is much more efficient to pool the objects instead of allocating them on the general heap. Another problem arises on game consoles where the memory available to the game is very limited and the engine must hold a tight control on the state of the memory and dynamically clean it if necessary while the action is still running.

12.1 Global allocation

The main part of this system are overridden default memory allocation functions of the compiler. The memory subsystem provides alternatives to the *malloc* and *free* functions called the *Memory::CustomMalloc* and *Memory::CustomFree*. The *new* and *delete* operators are overridden automatically, so using the functions manually is not necessary. Note that all this works in the whole application as well as in the libraries.

12.2 Free lists

The *Memory::FreeList* is a template implementation of a memory pool allocator. Its configuration is done by specifying the policies as template arguments. The *Memory::FreeList* then provides methods for allocation (*Allocate*) and deallocation (*Free)*. From the user's point of view they work the same way as *new* and *delete*, while even the constructor/destructor is called if required by the construction policy.

The allocation policy defines the structure of the pool in the memory. This may also influence the performance depending on the character of allocations/deallocations. The construction policy enables or disables the use of constructors/destructors while allocating/deallocating an object. The growth policy defines the way the memory pool is enlarged or shrunk when needed. This can also influence the performance and memory consumption.

12.3 Stl pool allocator

The *Memory::StlPoolAllocator* is a templatized STL-compliant pooled allocator. It is implemented as a wrapper on the top of *Memory::FreeList* with parameters suitable for STL containers. The allocator is completely ready to be directly used with the containers by supplying it as a template parameter into them. In the engine, this is done in the *Containers.h* file. The pooled containers are named *pooled_list*, *pooled_set*, etc.

12.4 Class allocation

As mentioned before, it is common in games to rapidly allocate and deallocate objects of the same type – a graphical effect might be an example. At the same time the programmer might not want to store the objects in a structure suitable for pooling and generally care about the memory management. In this case all he has to do is to derive the effect class from *Memory::ClassAllocation* with the correct template parameters. The class controls

the way instances of the deriving class are allocated and deallocated by overriding the *new* and *delete* operators. The usage of the class then does not change while the instance may be pooled.

Chapter 13

Platform setup

To be able to build the engine for different platforms, it was decided to concentrate all platform specific settings into one place. These include basic type definitions, macros, standard header file includes and containers. All of these can be found in the header files in the `Setup` directory under the source tree.

13.1 Specific header files

The main header file, included in most compilation units, is `Settings.h`. It contains basic settings for the platform (macros controlling the behavior of libraries for example), but more importantly it aggregates other setup headers together.

In `Platform.h` you can find macros defining the currently used platform. These macros are then used in other parts of the project to branch the code for specific platforms in compile time. In `BasicTypes.h`, there are definitions of simple types used in the whole project (integer, float, etc.). In `Containers.h`, there are definitions of the STL-like containers. And `ComplexTypes.h` contains definitions of other complex STL-like data structures.

`Memory_pre.h` and `Memory_post.h` define the memory allocation method used by the project and includes their implementation from the memory subsystem.

Chapter 14

Utilities and reflection

During the development of a game, usually several helper classes and methods are needed such as those for math or containers. This kind of functionality does not fit anywhere because it is too general. Placing it in a specific subsystem made it unusable anywhere else so they were aggregated in this namespace.

14.1 Helper classes and methods

There are several categories of utilities: containers (i.e. tree), math functions (i.e. hash) and design patterns (i.e. singleton). Here is the list of them with a brief description:

- Array – a templated representation of an array with the information about its size
- Callback – a generic callback solution that uniformly wraps pointers to functions and/or methods
- DataContainer – a class storing a pointer to a data buffer and its size
- FilesystemUtils – utility functions for working with files and directories
- GlobalProperties – globally accessible variables identified by string identifiers
- Hash – custom hash and reverse hash functions
- MathUtils – helper math functions
- ResourcePointers – a file that gathers smart pointers of all resources together so that it can be included with no overhead
- Singleton – a singleton pattern implementation
- SmartAssert – a smart assert macro implementation
- StateMachine – an implementation of a basic finite state machine
- StringConverter – a set of functions for converting different values to and from a string

- StringKey – a class that serves as a key into maps and other structures where strings are used to index data, but a high speed is necessary
- Timer – a support for time measurement
- Tree – an STL-like container class for n-ary trees
- XMLConverter – a set of functions for reading/writing different values from/to XML

14.2 RTTI

RTTI is a shortage for *Run-Time Type Information*. It is a mechanism that allows an instance of a class to know what its class is, the name of the class and other attributes. The instance is also able to create new instances of the same class, cloning itself.

RTTI is implemented using C++ templates. For using it a new class must derive from the *Reflection::RTTIGlue* class which takes two template arguments. The first one is the new class and the second one is its predecessor in the RTTI hierarchy. If there is no parent then the new class should inherit from the *Reflection::RTTIBaseClass* class. The mechanism will automatically call the static *RegisterReflection* method of the new class during the startup if defined. In this method new properties and functions can be registered for the reflection (see section 14.3) or a component dependency can be added by the *AddComponentDependency* method.

14.3 Properties

Properties are special RTTI attributes of classes that allow accessing data of their instances using string names. This greatly helps encapsulating classes and provides a uniform data access interface.

To make use of properties the class must be already using RTTI. A getter and setter method for each of data provided as properties must be defined. Then the static *RegisterReflection* function must be defined and inside its body the *RegisterProperty* method must be called for each property providing pointers to the data getter and setter. To access the data of an instance, the `GetRTTI()->GetProperty()` order on the instance is used. It returns the *Reflection::PropertyHolder* class, on which it is possible to call the template *GetValue* or *SetValue* methods.

It is also possible to set a custom function as a special property. The function must take the *PropertyFunctionParameters* class as a single parameter and return `void`. It must be registered by using the *RegisterFunction* method. The function can be then used in a similar way as the common properties, but the *CallFunction* method is used instead of the *SetValue* one.

Each object instantiated from a class using RTTI can register its own properties that are related to an object instead of a class. They are called dynamic properties and they can be registered by the *RegisterDynamicProperty* template method and unregistered by the *UnregisterDynamicProperty* method whenever during an object life cycle. The class can react on a (un)registration of a property by overriding the *DynamicPropertyChanged*

method. Dynamic properties are accessible in the same way as regular ones. If a dynamic property has a same name as some regular one, the dynamic one is returned.

14.4 Glossary

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

RTTI – a system for getting type information on run-time

Property – a class attribute accessible by its name that must be registered to the RTTI system

Dynamic property – data of any defined type associated to an object with RTTI that can be added and removed any time

Chapter 15

Conclusion

In this document the architecture of the Ocerus engine was described. Each of the chapters cover the design of a certain bigger part of the engine. It goes quite deep into describing how it works inside but the little details are left for the Doxygen reference documentation.

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

List of Tables