Design documentation of the Ocerus project

Lukas Hermann, Ondrej Mocny, Tomas Svoboda, Michal Cevora

December 30, 2010

Contents

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.

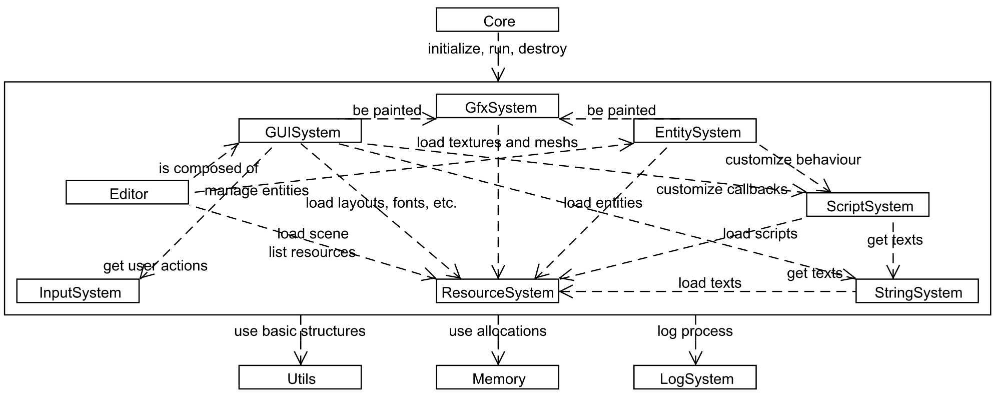


Figure 1.1: Dependencies among the systems

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.

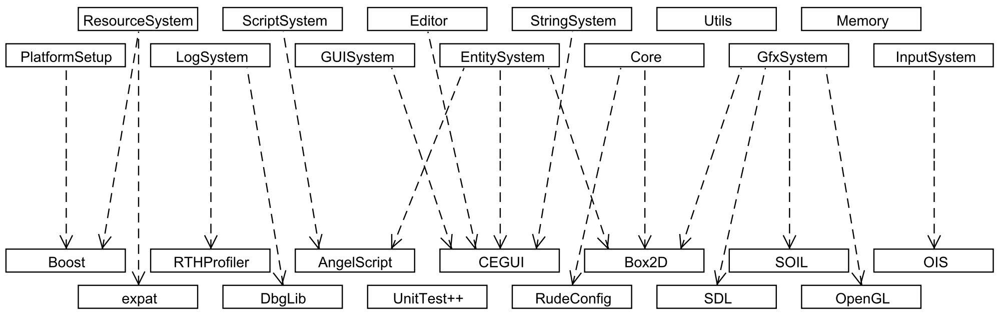


Figure 1.2: Library dependencies of the project systems

In the table 1.1 a brief description of all used libraries is provided.

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: Used libraries with the description

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

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

Third party code	Description
Properties, 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
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

at the end of most chapters there is a small glossary of terms used in that chapter.

Chapter 2

Core

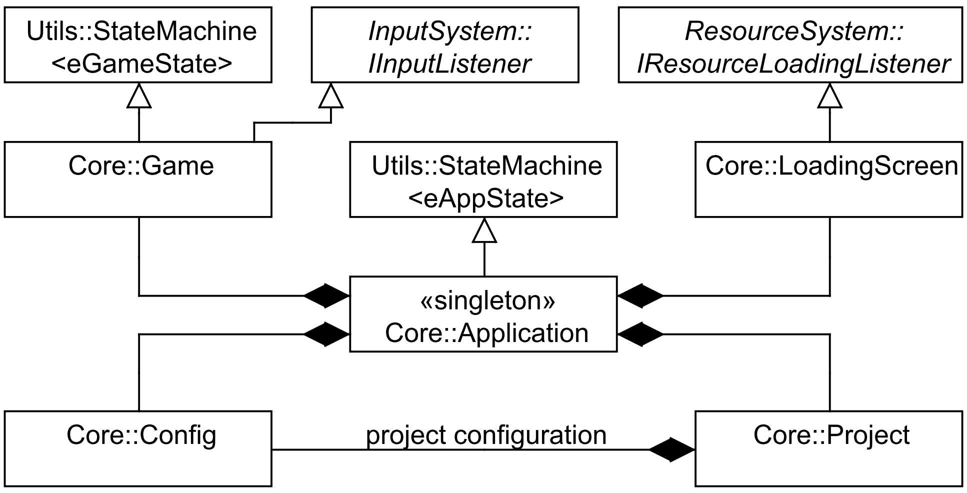

Utils::StateMachine <eGameState>
InputSystem:: IInputListener
ResourceSystem:: IResourceLoadingListener
Core::Game
Utils::StateMachine <eAppState>
Core::LoadingScreen
«singleton» Core::Application
Core::Config
project configuration
Core::Project

Figure 2.1: Class diagram of the Core namespace

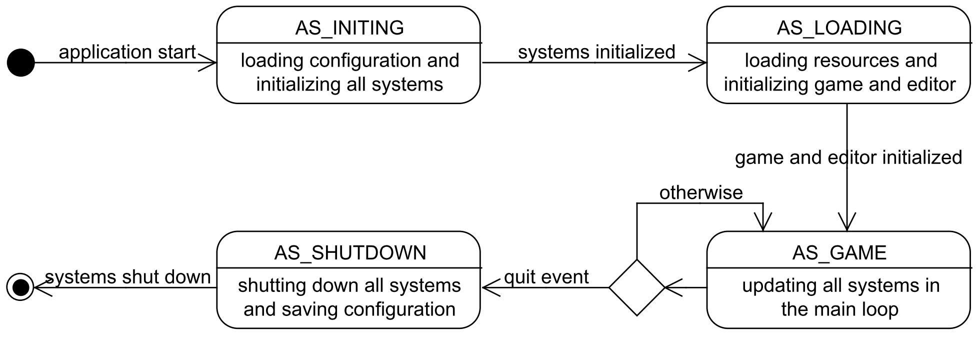
2.1 Purpose of the core

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).



AS_INITING
loading configuration and initializing all systems
application start
systems initialized
AS_LOADING
loading resources and initializing game and editor
game and editor initialized
otherwise
AS_GAME
updating all systems in the main loop
quit event
AS_SHUTDOWN
shutting down all systems and saving configuration
systems shut down

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 which can be in two states – paused or running. 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. 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.

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

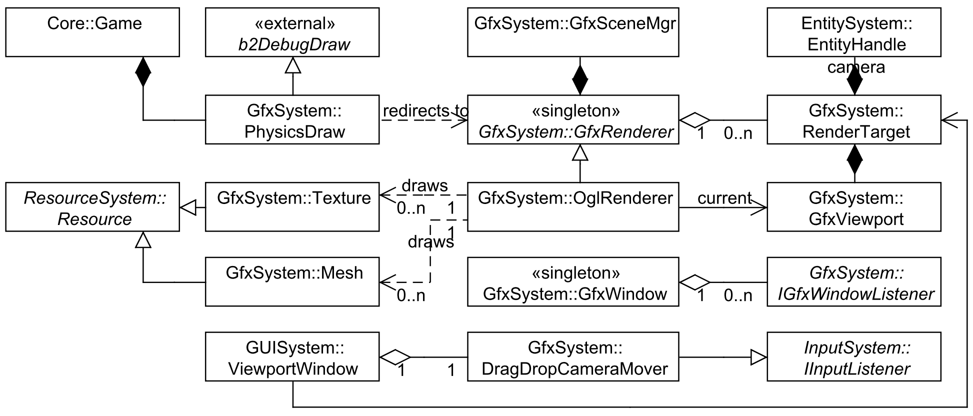


Figure 3.1: Class diagram of the GfxSystem namespace

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 *GfxSystem::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.

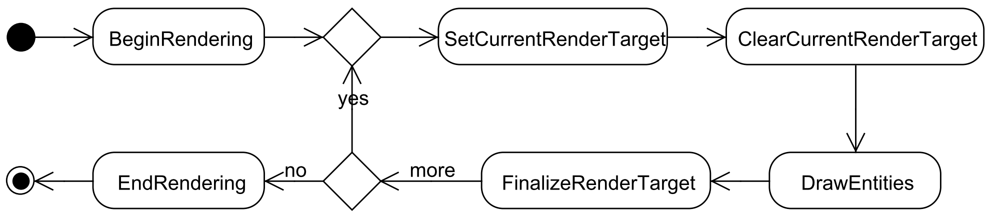


Figure 3.2: Activity diagram of the rendering process

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 diagram 3.2 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.

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

Chapter 4

Entity system

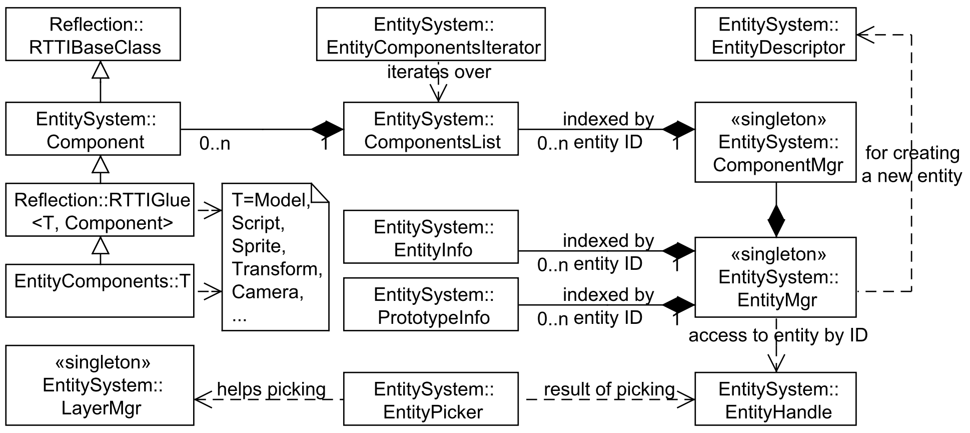


Figure 4.1: Class diagram of the Entity system namespace

4.1 Purpose of the entity system

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 a slower access to the object properties and behavior. It cooperates with the other systems like the graphics one for displaying objects or the script one for an interaction from scripts.

In the following sections the system of components and entities will be described as well as picking entities and organizing them in layers. In the last section there is a small glossary of used terms.

4.2 Components and their manager

Every game object is represented by an entity which is a compound of components that provide it 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 sent messages such as an initialization, a drawing, a logic update etc. by its own behavior. Component properties and behaviors are accessible only through an owner entity, so it is possible to read or write a specific property of an entity if it contains a component with this property and it is also possible to send a message to an entity which dispatches it to all its components that can react on 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). 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 that should be overridden by specific components 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 4.2). 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 is used that encapsulates a standard iterator (for example it has the *HasMore* method which returns whether the iterator is at the end of the component list).

4.3 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

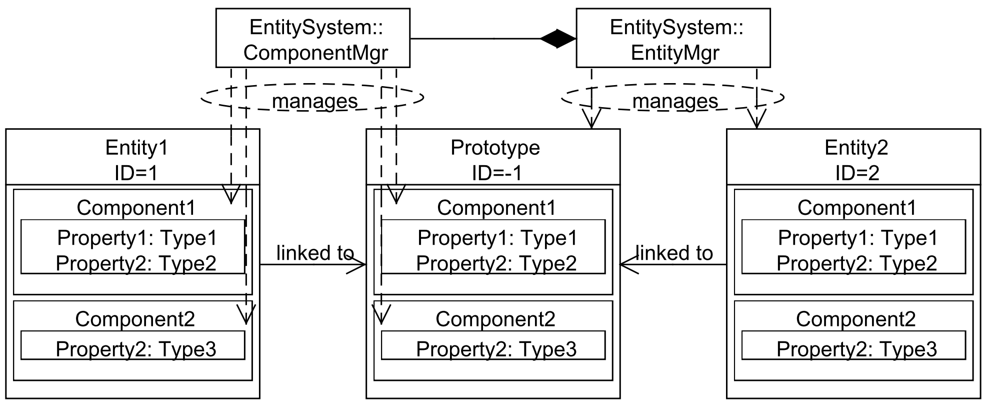


Figure 4.2: Objects managed by entity and component managers

methods of the entity manager with its ID. This class has also static methods that ensure all IDs in the system are unique.

For the creation of one entity the *EntitySystem::EntityDescription* class is used that 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 if 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 an 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 4.2). 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 4.3 for implementation details).

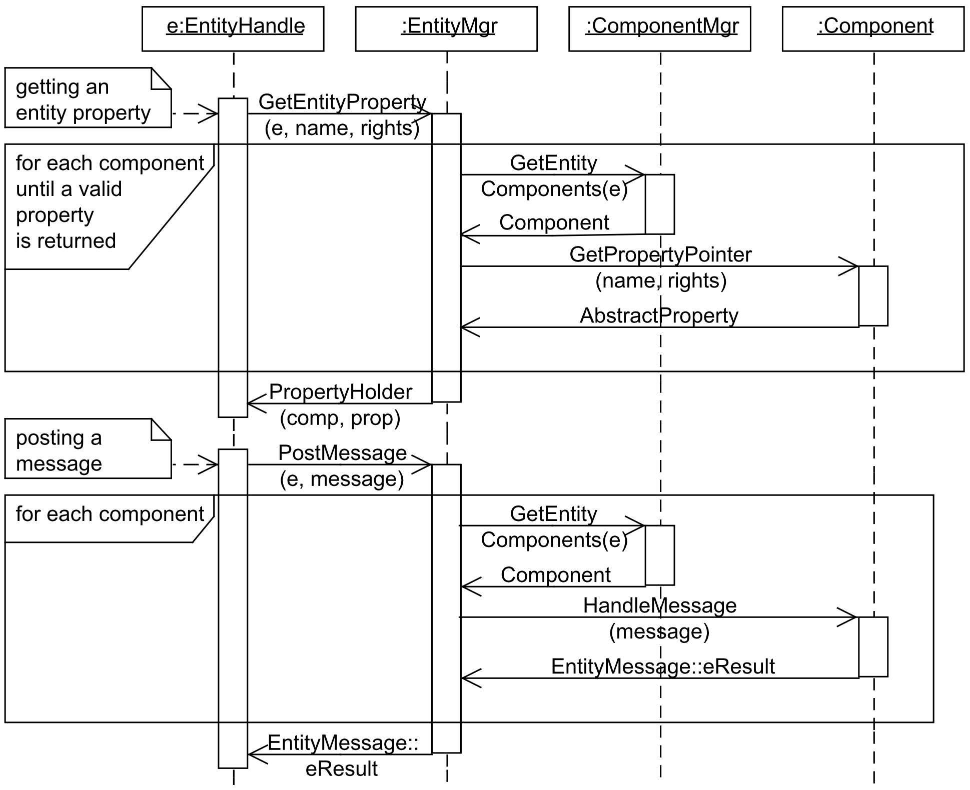


Figure 4.3: Process of getting an entity property and sending a message to an entity

4.4 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.

4.5 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 before (foreground, positive ID) or behind (background, negative ID) it.

4.6 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

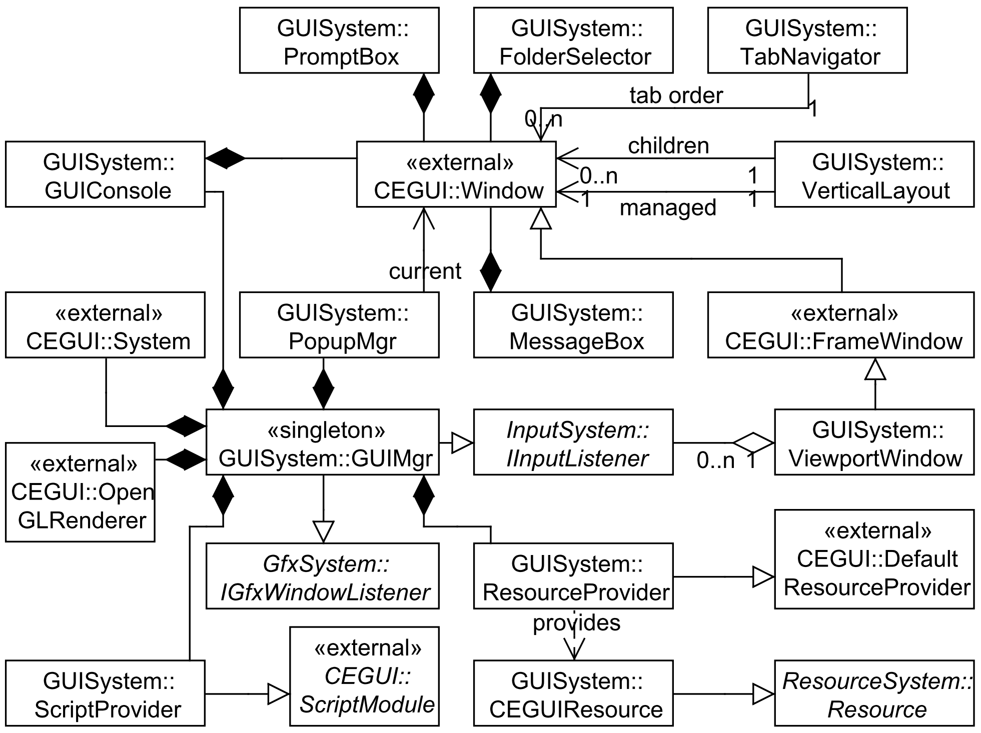
Prototype – changes of shared property values of this entity are propagated to the linked entities

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 5

GUI system



GUISystem:: PromptBox
GUISystem:: FolderSelector
GUISystem:: TabNavigator
tab order
0..n
1
GUISystem:: GUIConsole
«external» CEGUI::Window
children
0..n
1
1
managed
1
GUISystem:: VerticalLayout
current
«external» CEGUI::System
GUISystem:: PopupMgr
GUISystem:: MessageBox
«external» CEGUI::FrameWindow
«singleton» GUISystem::GUIMgr
InputSystem:: IInputListener
0..n
1
GUISystem:: ViewportWindow
«external» CEGUI::OpenGLRenderer
GfxSystem:: IGfxWindowListener
GUISystem:: ResourceProvider
«external» CEGUI::Default ResourceProvider
provides
GUISystem:: ScriptProvider
«external» CEGUI:: ScriptModule
GUISystem:: CEGUIResource
ResourceSystem:: Resource

Figure 5.1: Class diagram of the GUI system namespace

5.1 Purpose of the GUI system

The GUI system provides creating and drawing a graphic user interface based on the CEGUI library for the editor and the game itself. It also manages a user interaction with GUI elements, element layouts, viewports and GUI console.

In the following sections the GUI manager and its resource and script providers will be described as well as concept of layouts, viewports, popup menus and the GUI console. In the last section there is a small glossary of used terms.

5.2 GUI manager

The main class of the GUI system is the *GUISystem::GUIMgr* which is a connector for drawing, input handling and resource and script providing between the CEGUI library and the engine. During its creation it creates a CEGUI renderer, connects the resource manager with the CEGUI system via the *GUISystem::ResourceProvider* class (see section 5.3), connects the script manager with the CEGUI system via the *GUISystem::ScriptProvider* class (see section 5.4), provides itself as an input and screen listener and creates the GUI console which is then accessible via the *GUIMgr::GetConsole* method (see section 5.8). On its initialization (*GUIMgr::Init*) it loads necessary GUI resources (schemes, image sets, fonts, layouts and looknfeels) and creates a root window.

For loading a root layout from a file the method *GUIMgr::LoadRootLayout* is provided with only one parameter specifying a name of a file where a layout is defined. This method calls the *GUIMgr::LoadWindowLayout* which is a common method for loading a window layout from a file that also provides a translation of all texts via the string manager. There are also methods for unloading and getting the current root layout. For more information about layouts see section 5.5.

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.

5.3 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.

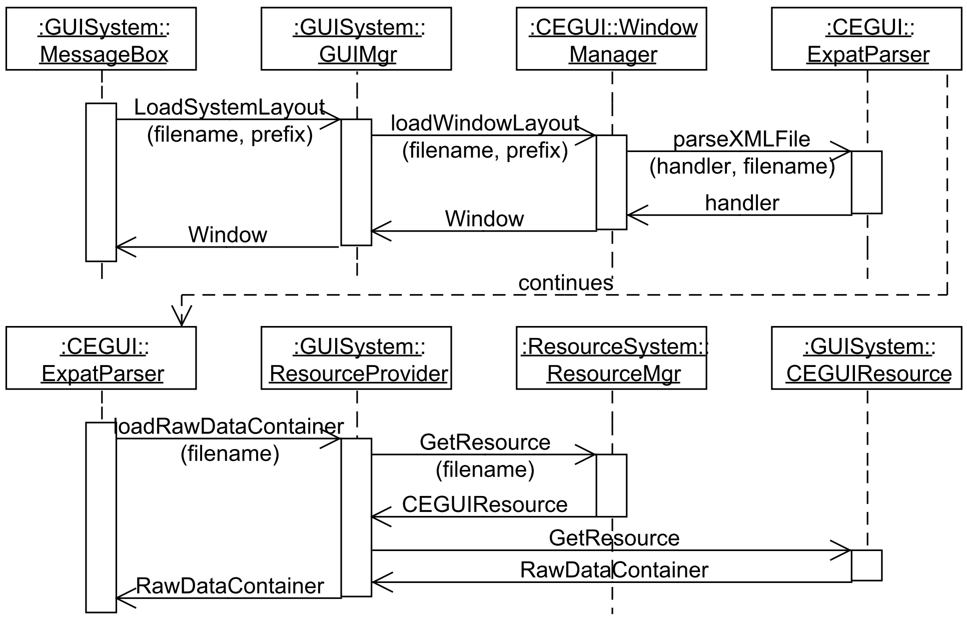

:GUISystem::MessageBox
:GUISystem::GUIMgr
:CEGUI::WindowManager
:CEGUI::ExpatParser
LoadSystemLayout (filename, prefix)
loadWindowLayout (filename, prefix)
parseXMLFile (handler, filename)
handler
Window
Window
continues
:CEGUI::ExpatParser
:GUISystem::ResourceProvider
:ResourceSystem::ResourceMgr
:GUISystem::CEGUIResource
loadRawDataContainer (filename)
GetResource (filename)
CEGUIResource
GetResource
RawDataContainer
RawDataContainer

Figure 5.2: An example of loading a GUI layout

5.4 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 which 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 which stores the function name given in its constructor and which 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.

5.5 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 a 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 a 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.

5.6 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 which 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*.

5.7 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.

5.8 GUI console

The *GUISystem::GUIConsole* class manages the console 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 ones which have an equal or higher level than 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 and 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 hide by *GUIConsole::ToggleConsole* method.

5.9 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

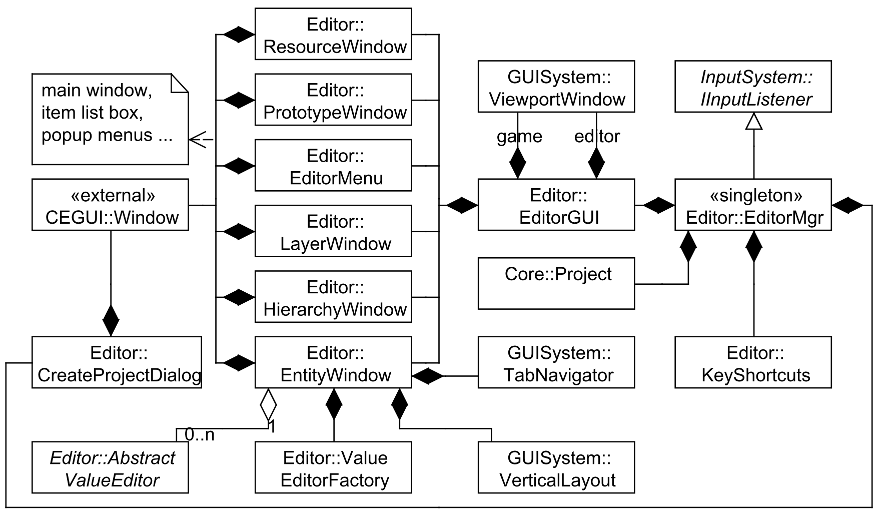


Figure 6.1: Class diagram of the Editor namespace

6.1 Purpose of the editor

The editor is used for an easy creation of a new game based on this engine. It provides a well-arranged graphic user interface for managing everything from the whole project to each entity in several scenes. 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.

In the following sections the classes providing the editor will be described. They do not have to be contained in the final distribution of the game if an editor support should not be allowed. First section is focused on the logical and graphical managers of the editor whereas the second section is about layouts used to build the GUI of the editor. In the last section a concept of value editors is introduced.

6.2 Editor managers

The *Editor::EditorMgr* class manages the logical part of the editor and it owns an instance of the *Editor::EditorGUI* class that focuses to the GUI of the editor. Both of them have methods called at the loading and the unloading of the editor and also methods for updating logic and drawing in the application loop.

The first mentioned 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 reference to all editor windows and for updating them.

The second class compounds all editor layouts such as the game and editor viewports, the resource, prototype and layer windows and the editor menu and initializes and updates them. It directly updates the entity editor window according to the current entity that uses the vertical layout for positioning entity components and various value editors for showing and editing all kinds of entity properties such as strings, vectors, resources and arrays.

6.3 Used layouts

There are six 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 and the *Editor::ResourceWindow* manages the resources that the entities can use.

6.4 Value editors

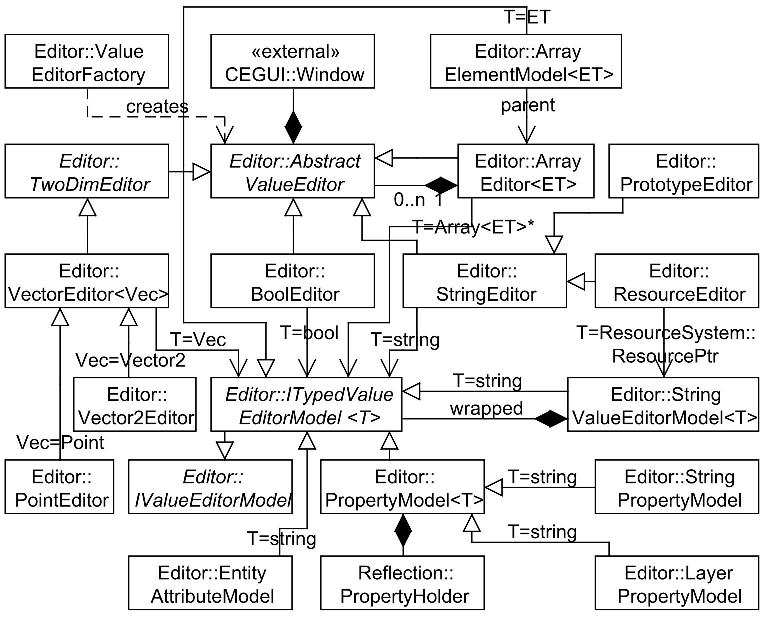

Editor::Value EditorFactory
creates
«external» CEGUI::Window
T=ET
Editor::Array ElementModel<ET>
parent
Editor:: TwoDimEditor
Editor::Abstract ValueEditor
Editor::Array Editor<ET>
Editor:: PrototypeEditor
0..n
1
T=Array<ET>*
Editor:: VectorEditor<Vec>
Editor:: BoolEditor
Editor:: StringEditor
Editor:: ResourceEditor
T=Vec
T=bool
T=string
T=ResourceSystem:: ResourcePtr
Vec=Vector2
Editor:: Vector2Editor
Editor::ITypedValue EditorModel <T>
T=string
wrapped
Editor::String ValueEditorModel<T>
Vec=Point
Editor:: PointEditor
Editor:: IValueEditorModel
Editor:: PropertyModel<T>
T=string
Editor::String PropertyModel
T=string
T=string
Editor::Entity AttributeModel
Reflection:: PropertyHolder
Editor::Layer PropertyModel

Figure 6.2: Class diagram of editor value editors

All value editors derive from the *Editor::AbstractValueEditor* base class. This class has three abstract methods that every value editor must implement and some methods that can help with the implementation. The first is the *AbstractValueEditor::CreateWidget* method with the parameter representing a string that should be used as a prefix for a name of every created component. This method should create a widget for editing a required kind of value and return it. The second is the *AbstractValueEditor::Update* method which is called when the current value of the property should be displayed in the value editor and the last is the *AbstractValueEditor::Submit* method which is called when the value of the property should be updated by the user input to the value editor.

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, 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 implements a simple editor for properties which values are easily convertible from and to string which 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 create 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.

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

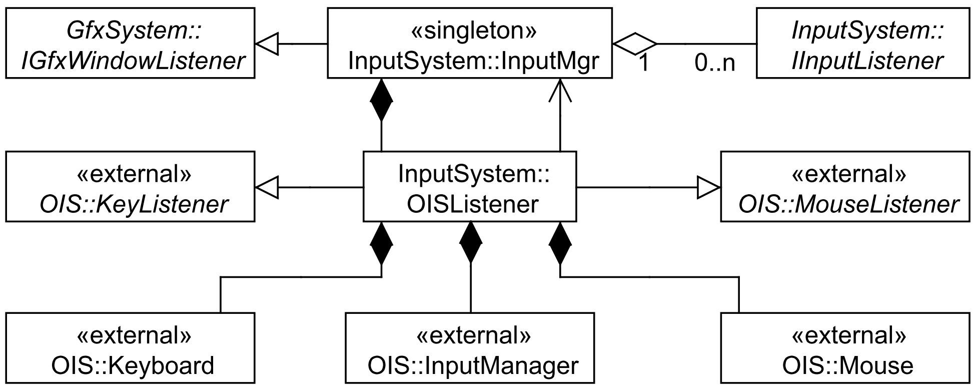


Figure 7.1: Class diagram of the Input system namespace

7.1 Purpose of the input system

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

In the following section the way of receiving and distributing input events is described.

7.2 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 the 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 *Input-Mgr::CaptureInput* method is called where the events are recognized and then distributed to the listeners.

The *InputSystem::InputMgr* class is only a proxy class which 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.

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