Documentation to project Ocerus

Lukas Hermann, Ondrej Mocny, Tomas Svoboda

July 18, 2010

Contents

Chapter 1

Introduction

This document serves as both user and design documentation for the Ocerus project. In this opening chapter the purpose of this project will be revealed, the structure of this document and the recommended way of reading it will be presented and finally the project architecture will be described.

1.1 Purpose of this project

The project Ocerus implements a multiplatform game engine and editor for creating simple 2.5D games. The main focus lies on the editing tools integrated to the engine. So, every change is immediately visible in the game context and can be tested in a real time. The engine takes care of rendering, physics, customization of game object behavior by scripts, resource management and input devices. It also provides a connection between the parts of the engine. The game objects are easily expandable using new components or scripts. Scripts themselves have access to other parts of the system.

1.2 How to read this documentation

Since this product is intended to be used by game developers rather than end users and because it is necessary to understand the common ways of developing more complex games the user documentation and the design documentation are merged into a single document. This section should help users of this product to orient in the document and let them know where the important information is located.

The first section that should be read before starting development of a new project is the section 6.2. It describes the basic work with the GUI of the editor – how to create a new project, add a new scene, create entities

and manipulate with them etc. For a change of a default entity behavior it is necessary to understand the associated script language syntax (10.2) and its functions and methods (10.3.2, 10.3.3 and 10.3.4). If some behavior cannot be added by scripts or it would be inefficient new components from which entities are built can be added, which is described in section 3.3.

For adding a game menu or any text to the game the section 5.3 which explains the GUI creation. For a multilanguage support all texts in the game must not be hardcoded but the string system described in the chapter 11 should be used instead. If the game needs more advanced configuration see the section 2.5. When a new native code is added it can be useful to log debug message (8.2), profile the code (8.3), use helper classes (13) and access the new code from scripts (10.4.1).

1.3 Project architecture

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

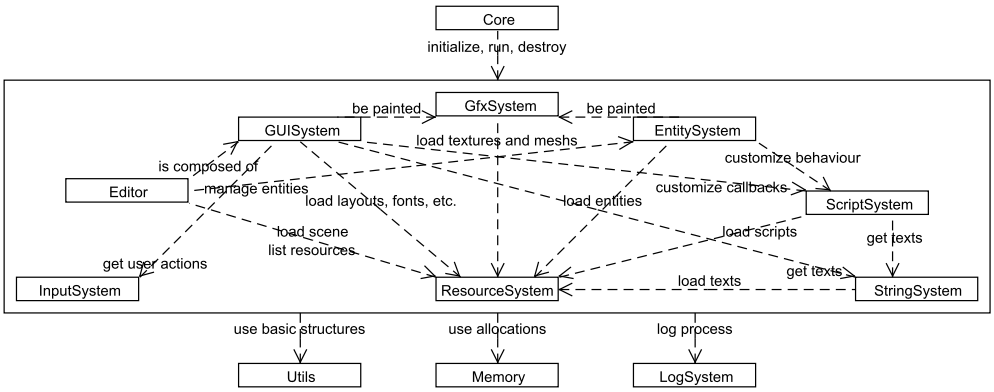

Figure 1.1: Dependencies among the systems

The project has not been created from scratch but it is based on several libraries to allow the developers to focus on important features for the end users and top-level design rather than low-level programming. All used libraries support many platforms, have free licenses and have been heavily tested in a lot of other projects. All of them are used directly by one to three

subsystems except the library for unit testing. The library dependences of each system are displayed in the picture 1.2.

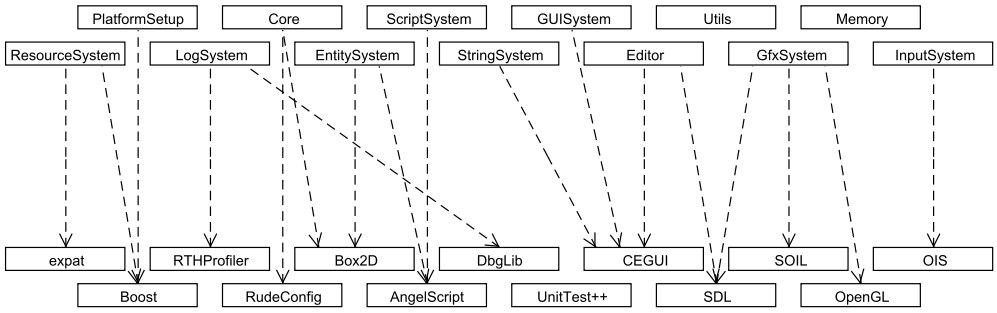

Figure 1.2: Library dependencies of the project systems

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

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

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

Chapter 2

Core

Namespaces: Core

Headers: Application.h, Config.h, Game.h, LoadingScreen.h, Project.h

Source files: Application.cpp, Config.cpp, Game.cpp, LoadingScreen.cpp, Project.cpp

Classes: Application, Config, Game, LoadingScreen, Project

Libraries used: Box2D, RudeConfig

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, invokes 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 managing 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.

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 a *loading* state) or updated and drawn (in a *game* state).

In the application class there are also methods for getting an average and last FPS statistic and methods for showing and hiding a debug console as well as writing message to it. 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.

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.

Before the game initialization at the method *Init* a valid render target must be set by method *SetRenderTarget* to know where to draw the game content. This is done for example by the editor when the game is run from it. Then physics, time, an action etc. are initialized and in the *Update* method called in the main loop they are updated.

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

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

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

Since the class *Core::Game* registers the input listener to itself there are callbacks where it is possible to react to keyboard and mouse events such as a key or mouse button press/release or a mouse move. The corresponding information such as a current mouse position is available through the callback

parameters.

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. 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 has 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). Another 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 parametrizes 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

Entity system

Namespaces: EntitySystem, EntityComponents

Headers: Component.h, ComponentEnums.h, ComponentHeaders.h, ComponentID.h, ComponentIterators.h, ComponentMgr.h, ComponentTypes.h, EntityDescription.h, EntityHandle.h, EntityMessage.h, EntityMessageTypes.h, EntityMgr.h, EntityPicker.h

Source files: Component.cpp, ComponentEnums.cpp, ComponentMgr.cpp, EntityDescription.cpp, EntityHandle.cpp, EntityMessage.cpp, EntityMgr.cpp, EntityPicker.cpp

Classes: Component, EntityComponentsIterator, ComponentMgr, EntityDescription, EntityHandle, EntityMessage, EntityMgr, EntityPicker

Libraries used: none

3.1 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 the extending of the system which will be probably the

first action when creating a new game. In the last section there is a small glossary of used terms.

3.2 Components and entities

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.

3.2.1 Components and their manager

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). 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 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 section 3.3.1).

The *EntitySystem::ComponentMgr* is a singleton class that manages instances of all entity components in the entity system. 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).

3.2.2 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 ensures 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 3.3.2 for more information).

All entities are managed by the *EntitySystem::EntityMgr* class that stores necessary information about them in maps indexed by their ID. 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 or an XML resource and for destroying them. Another methods manages entity prototypes - it is possible to link/unlink an instance to/from a prototype, to set a property as (non)shared and to invoke an update of instances of a specific prototype. 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 posting and broadcasting messages to entities and for adding, listing and removing components of a specific entity.

3.2.3 Entity picker

The entity picker 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.

3.3 Extending the entity system

The entity system will be probably the first system to extend when creating a new game. It is necessary to create new components with their specific properties and behavior from which new entities can be created in the game editor or from scripts. It is also possible to create new message types that can be sent to entities when it is needed to inform about new different events.

3.3.1 Creating new components

There are several steps that leads to creating a new component. First it is needed to create a class *ComponentName* (to be replaced by a real component name) which publicly inherits from the *Reflection::RTTIGlue<ComponentName, Component>* class and which is in the *EntityComponents* namespace. This class should override the *Create*, *Destroy* and *HandleMessage* methods for a custom behavior on creation, destruction and handling messages. In the last method the message structure is got from the first parameter so it is possible to get a message type and message parameters (the *GetParameter(index)* method is used). The method should return *EntityMessage::RESULT_OK* if the message was processed, *EntityMessage::RESULT_IGNORED* if it was ignored or *EntityMessage::RESULT_ERROR* if an error occured.

The last common method of the class will be the *RegisterReflection* method that is static and that is called automatically when the application initializes. It should register all properties and functions which the component provides by the *RegisterProperty* and *RegisterFunction* methods inherited from the base class. In case of a property the following information must be specified: a type (from PropertyTypes.h), a name (must be unique in a component), a getter (a constant function that returns value of a same type and has no parameters), a setter (a non-constant function with one constant parameter of a same type and no return value), an access flags (a disjunction of the *Reflection::ePropertyAccess* enumerations) and a comment (will be displayed in the editor). A getter and a setter can be simple functions that return or modify a private member variable or they can do a more complex computing. For example the registration of an integer property with a getter

and a setter as methods of the class and with a read and write access from the editor looks like:

```
RegisterProperty<int32>("IntProp", &ComponentName::GetIntProp,
    &ComponentName::SetIntProp, PA_INIT | PA_EDIT_READ |
    PA_EDIT_WRITE, "This is an integer property.");
```

After the creation of the class a new component type must be registered in ComponentTypes.h where a new line like `COMPONENT_TYPE(CT_COMPONENT_NAME, ComponentName)` should be added. Finally the header file where the component is declared must be added to ComponentHeaders.h, for example `#include "../Components/ComponentName.h"/`. Now the component is ready to be added to entities.

3.3.2 Adding new entity message types

If it is necessary to add a new entity message type that can be posted to entities then the new line to EntityMessageTypes.h must be added. It should look like `ENTITY_MESSAGE_TYPE(MESSAGE_NAME, "void OnMessageName(type1, type2, ...)", Params(PT_TYPE1, PT_TYPE2, ...))` where the first parameter is an enumeration constant, the second one is a declaration of script function that handles the message and the last one is a definition of message parameters where `PT_TYPE` is from PropertyTypes.h. In case the message has no parameters the `NO_PARAMS` macro should be provided as the last parameter.

3.4 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 an 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

Chapter 4

Gfx system

Namespaces: GfxSystem

Headers: GfxRenderer.h,GfxSceneMgr.h, GfxStructures.h, GfxWindow.h, IGfxWindowListener.h, OglRenderer.h, Texture.h, RenderTarget.h, GfxViewport.h, DragDropCameraMover.h, PhysicsDraw.h

Source files: GfxRenderer.cpp,GfxSceneMgr.cpp, GfxStructures.cpp, GfxWindow.cpp, OglRenderer.cpp, Texture.cpp, GfxViewport.cpp, DragDropCameraMover.cpp, PhysicsDraw.cpp

Classes: GfxRenderer, OglRenderer, GfxSceneMgr, GfxWindow, IGfxWindowListener, Texture, GfxViewport, DragDropCameraMover, PhysicsDraw

Libraries used: SDL, OpenGL, SOIL

4.1 Introduction

GfxSystem implements functionalities related to the rendering of game entities. The design of this system is influenced by the the requirement of platform independence.

Note that *GUISystem* uses its own renderering system.

4.2 Application window

GfxSystem also manages creating and handling the application window which depends on the used operating system. In the Ocerus, this functionality is implemented by the *GfxWindow* class with the usage of the SDL library.

SDL (Simple DirectMedia Layer) is a free cross-platform multi-media development API used for games, emulators, MPEG players, and other applications. The main advantage is that it supports many operating systems.

It is used for handling window events aswell. Currently, the only event that is being handled is the "change resolution" event. There is a listener implemeted for this event(*IGfxWindowListener*) so every other system that needs to be informed about a resolution change can add its own listener. This feature is used by the *InputSystem* and the *GUISystem*.

Note that SDL 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 SDL features used is Ocerus are window management and creating rendering context.

4.3 Renderer

For low-level rendering, the OpenGL library is used. OpenGL works on most platforms but not on all. *GfxSystem* is designed in a way that allows to easily implement support for other low-level graphic libraries (e.g. DirectX). It is just necessary to create new derived class and implement few virtual functions. The base class that communicates with other systems is the *GfxRender* and its derived class using the OpenGL calls is the *OglRenderer*.

4.4 Textures

For loading the textures, SOIL library is used. SOIL (Simple OpenGL Image Library) is a tiny C library used for uploading textures into the OpenGL. It supports most of the the common image formats.

Loading textures is implemented in the class *Texture*. It inherits from *Resource* class, see *ResourceSystem*.

Since SOIL depends on the OpenGL, class *Texture* uses methods from *GfxRenderer* or more precisely *OglRender*.

4.5 Process of rendering the entities

Each entity, upon its initialization, registers its graphic representing components in the *GfxSceneMgr*. Upon the request from the game loop in the *Core*, the *GfxSceneMgr* draws all registered components, using methods from the *GfxRenderer*. It is also necessary to set up the rendering target within the *GfxRenderer*. This is usually done by the *GUISystem*.

Chapter 5

GUI system

Namespaces: GUISystem

Headers: CEGUIForwards.h, CEGUIResource.h, CEGUITools.h, GUIConsole.h, GUIMgr.h, MessageBox.h, ResourceProvider.h, ScriptProvider-.h, VerticalLayout.h, ViewportWindow.h

Source files: CEGUIResource.cpp, CEGUITools.cpp, GUIConsole.cpp, GUIMgr.cpp, MessageBox.cpp, ResourceProvider.cpp, ScriptProvider-.cpp, VerticalLayout.cpp, ViewportWindow.cpp

Classes: CEGUIResource, GUIConsole, GUIMgr, MessageBox, ResourceProvider, ScriptCallback, ScriptProvider, VerticalLayout, ViewportWindow

Libraries used: CEGUI

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 an user interaction with GUI elements, element layouts, viewports and GUI console.

In the following sections the connection to the engine will be described as well as the method of creating an own GUI. In the last section there is a small glossary of used terms.

5.2 Connection to the engine

In this section all necessary parts of GUI system will be introduced with their connection to the other parts of engine.

5.2.1 GUI manager

The main class of GUI system is the *GUISystem::GUIMgr* which is a connector for drawing, input handling and resource providing between the CEGUI library and the engine. During creation it creates a CEGUI renderer, connects the resource manager with the CEGUI system via the *GUISystem::ResourceProvider* class (see section 5.2.2), provides itself as a input and screen listener and creates GUI console which is then accesible via the *GUIMgr::GetConsole* method (see section 5.2.5). On initialization (*GUIMgr::Init*) it loads necessary GUI resources (schemes, imagesets, fonts, layouts, 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 layout see section 5.2.3.

Sometimes it is needed to disconnect a callback function from an event such as a mouse click on a button. Since the CEGUI library crashes when an event is disconnected in its callback the *GUIMgr::DisconnectEvent* method is introduced which adds the event to the list of events that should be disconnected and the real disconnection will process after calling the method *GUIMgr::ProcessDisconnectedEventList* in the application main loop.

There are two more methods called in the application main loop. First the *GUIMgr::Update* updates time of GUI system, 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.

5.2.2 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.

5.2.3 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.

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, then the *VerticalLayout::AddChildWindow* method is used for adding them. It is also possible to set a 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 section 5.3.

5.2.4 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*.

5.2.5 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 *GUIConsole::AppendScriptMessage* method. For better usage of the console a history of previously type commands is stored and can be revealed by up and down arrows. The console can be shown or hide by *GUIConsole::ToggleConsole* method.

5.3 Creating GUI

This section focuses on creating a GUI to the game. First there is a description of a layout definition, then the way how handle events is introduced and finally the connection of these two parts in the GUI layout component is described.

5.3.1 Defining a layout

The GUI layout for the game should be defined in a file with the *.layout* extension and the XML internal structure. Since the CEGUI library is used for the GUI system a layout must fulfil its specification which is widely described in its documentation [1] therefore there is only a brief description it the following paragraphs.

The `<GUILayout>` element is the root element in layout XML files that must contain a single `<Window>` element representing the root GUI element. The `<Window>` element must have the `Type` attribute which specifies the type of window to be created. This may refer to a concrete window type, an alias, or a falagard mapped type (see the next paragraph). It can have also the `Name` attribute specifying a unique name of the window. This element may contain `<Property>` elements with the `Name` and `Value` attributes used to set properties of the window, `<Event>` elements with the `Name` and `Function` attributes used to create bindings between the window and script functions (see the next subsection) and another `<Window>` elements as its child windows. For supported window types, properties and events see the CEGUI documentation.

For defining connections between physical window type and window look (rendering, fonts, imagesets) the scheme file is used which is another XML file with a specific structure. Its root element is the `GUIScheme` one that can contain any number of for example `<Imageset>`, `<Font>`, `<LookNFeel>`, `<WindowAlias>` or `<FalagardMapping>` elements. For concrete usage of these elements as well as building a GUI layout with own fonts, images etc. see the CEGUI documentation.

The only specific feature of writing GUI layouts for this game engine is the translation of a text in the `Text` and `Tooltip` properties if it is surrounded by $ characters (i.e. `Text=$text$`). This text is used as a key and the word `GUI` as a group for the string manager and it is replaced by the result according to the current language when the layout is loaded to the engine.

5.3.2 Event handling

Events generated by an interaction of user with GUI elements (mouse clicking, key pressing, etc.) can be handled by script functions written to a script module. If a layout is connected with a script module via a GUI layout component (see the next section) every occurence of an event specified in the `Name` attribute of the layout element `<Event>` will be followed by calling a script function with the name equal to the `Function` attribute in the same tag with a `Window@` parameter holding a reference to the window which the

element is child of.

For example assume this is a part of a layout

```
<Window Type="CEGUI/PushButton" Name="Continue">
  ...
  <Event Name="MouseClick" Function="ContinueClick">
  ...
</Window>
```

and this is a part of a script module

```
void ContinueClick(Window@ window)
{
  Println(window.GetName());
}
```

connected via a GUI layout component contained in any entity in the scene then every click on the button named `Continue` will print the message `Continue` to the log.

5.3.3 GUI layout component

The GUI layout component serves as a connection of one GUI layout file and one script module with functions that serves as callbacks to events of GUI elements described in the layout. Beside of these two properties there is a property `Visible` indicating whether the layout should be visible and active.

It loads the layout on its initialization to the GUI system and it calls a script function with the declaration `void OnUpdateLogic(float32)` in the specified module every time it receives the *UPDATE_LOGIC* message where it is possible to update all information displayed by GUI elements (use the global function `Window@ GetWindow(string)` with the name specified in the layout file to get the reference to the window).

To show the GUI layout just add this component to any entity in the scene (or create a new one) and set the `Visible` property to true. The GUI elements are displayed over all entities and are not affected by cameras nor layers.

5.4 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

Chapter 6

Editor

Namespaces: Editor, GUISystem

Headers: EditorGUI.h, EditorMenu.h, EditorMgr.h, FolderSelector.h, LayerMgrWidget.h, PopupMenu.h, PrototypeWindow.h, ResourceWindow.h and editors of properties

Source files: EditorGUI.cpp, EditorMenu.cpp, EditorMgr.cpp, FolderSelector.cpp, LayerMgrWidget.cpp, PopupMenu.cpp, PrototypeWindow.cpp, ResourceWindow.cpp and editors of properties

Classes: EditorGUI, EditorMenu, EditorMgr, FolderSelector, LayerMgrWidget, PopupMenu, PrototypeWindow, ResourceWindow

Libraries used: CEGUI, SDL

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 usage of the editor to make a new game as well as the classes providing the editor will be described. In the last section there is a small glossary of used terms.

6.2 Using the editor

6.2.1 Managing projects

6.2.2 Managing scenes

6.2.3 Managing entities

6.2.4 Editor tools

6.3 Description of editor classes

In this section 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 subsection is focused on the logical and graphical managers of the editor whereas the second subsection is about layouts used to built the GUI of the editor.

6.3.1 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 manages the editing of entities. It manages selecting entities and editing the current one, it reacts on choosing all items in the entity submenu such as creating a new entity, duplicating and deleting current or selected entities, adding and removing entity components or creating a prototype from a current entity 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 handling all popup menus.

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.

6.3.2 Writing own value editors

All value editors derives 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 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, element of a list or removable and also they should be able to remove the property 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.

6.3.3 Used layouts

There are several layouts implemented for the editor that can be used as examples of a creation of an own layout. The *GUISystem::FolderSelector* class represents the layout for selecting a folder and has methods for show-

ing/hiding it, registering callback for the Ok/Cancel pushing and getting the current and selected folder. The *Editor::PopupMenu* class manages all popup menus in the editor and has methods for opening/closing it, for initializing it and for handling all menu choices.

There are three layout classes for editor components. All of them have initialization and update methods and callbacks for significant events. All of them loads their look from an external XML file and introduces the reactions on events and loading data from the application. The *Editor::LayerMgrWidget* manages the layers the entities are put to, the *Editor::PrototypeWindow* manages the prototypes of entities and the *Editor::ResourceWindow* manages the resources that the entities can use.

6.4 Glossary

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

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.)

Entity – represents one object in the scene with specific properties and behavior

Chapter 7

Input system

Namespaces: InputSystem

Headers: IInputListener.h, InputActions.h, InputMgr.h, OISListener.h

Source files: InputMgr.cpp, OISListener.cpp

Classes: InputMgr, IInputListener

Libraries used: OIS

7.1 Introduction

Each game must somehow react to the input from the player. It can come from a keyboard, a mouse or a gamepad. To keep things organized handling of those devices were concentrated into a single part of the engine - the Input System.

7.2 What does it do

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 *InputMgr* implements both of the approaches.

The first one is provided via the *IInputListener* interface which you may implement and then register into the *InputMgr* by using its *AddInputListener* method. You'll then receive the notification in callbacks of the interface.

The second approach is supported by multiple methods in *InputMgr* for querying the device state. For example, *IsKeyDown* will tell you if any

specific key is currently held down and *GetMouseState* will return a whole bunch of mouse related informations.

7.3 Event processing

To keep things synchronized the event processing is executed in the main game thread. At the beginning of each iteration of the game loop the *Capture* method is called. Inside the events are recognized and then distributed.

Chapter 8

Log system

Namespaces: LogSystem

Headers: Logger.h, LogMacros.h, LogMgr.h, Profiler.h

Source files: Logger.cpp, LogMgr.cpp, Profiler.cpp

Classes: Logger, LogMgr, Profiler

Libraries used: RTHProfiler

8.1 Purpose of the log system

The log system manages internal log messages that are used to provide information about application processes useful to debug the whole 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 ingame profiling useful for a location of the most time critical parts of the project which can leads to effective optimization.

In the following sections the process of logging messages will be described as well as using a profiler. In the last section there is a small glossary of used terms.

8.2 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 which 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 a 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) that called the *LogMgr*'s method with the built message.

For an even easier logging of messages log 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 information level then for example the following message will be generated: `13:05:18: Entity(25) was created.`

8.3 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*) activates a profiling 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 activate via the method *IsRunning*.

8.4 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 appears

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

Chapter 9

Resource system

Namespaces: ResourceSystem

Headers: IResourceLoadingListener.h, Resource.h, ResourceMgr.h, ResourceTypes.h, UnknownResource.h, XMLOutput.h, XMLResource.h

Source files: Resource.cpp, ResourceMgr.cpp, ResourceTypes.cpp, XMLOutput.cpp, XMLResource.cpp

Classes: Resource, ResourceMgr, ResourcePtr, UnknownResource, XMLOutput, XMLResource

Libraries used: Boost, Expat

9.1 Introduction

Every game needs to load packs of data from external devices such as the hard drive or network. The data come in blocks belonging together and representing a unit of something we usually call *resource*. Because 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.

9.2 Resources

As said before, the resource is a group of data belonging together. In the engine this is represented by the abstract *Resource* class. It contains all the basic attributes of a resource and allows its users to load it 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 and so using *Resource* 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 *Resource* until it reaches the graphical subsystem which converts it to the texture resource and grabs the implementation specific texture data out of it.

9.2.1 Resource states

Each resource can be in one of the states described in the table 9.1 at the given point of time:

A state	A description
Uninitialized	the system does not know about it; it's not registered
Initialized	the system knows about it, but the data are not loaded yet
Unloading	the data are just being unloaded
Loading	the data are just being loaded
Loaded	the resource is fully ready to be used

Table 9.1: Possible resource states and their description

9.2.2 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 exists a pointer to the same tree or any of its parts. Another example is a texture. After it loads the data they are passed into the graphical subsystem which creates a platform specific texture out of it and returns only a the texture handle. The handle is then stored only in the resource.

9.2.3 Resource pointers

Because the resources are passed all around the game system we must somehow prevent any memory leaks from appearing. Doing so is quite easy how-

ever - we simply use the shared pointer mechanism. It points to the common abstract resource (*ResourcePtr* class) and to specific resources as well (*ScriptResourcePtr* or *XMLResourcePtr* for example). The abstract resource pointer can be automatically converted to any specific resource pointer, but if the type won't match an assertion fault will be raised to prevent memory corruption. All resource pointers are defined in `ResourcePointers.h`.

9.3 Resource manager

The Resource Manager represented by the *ResourceMgr* class takes care of organizing resources into groups and providing and 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 (for a single level of the game, for example).

9.3.1 Hotloading

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 you change a currently loaded texture in an image editor and save it it will be immediately updated in the game.

9.3.2 Memory limit

Because gaming systems have limited memory we must make sure we can limit the memory used by resources. Resources usually take the biggest chunk of memory, so when lowering the memory usage it's best to start here. Hopefully, *ResourceMgr* allows us to define a limit which it will try to keep. When the memory is running out, it will attempt to unload resources which 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.

9.4 Adding custom resource types

To add a new resource type you must go through two steps. First create a class derived from *Resource* implementing all abstract methods and providing

accessors to the parsed data. Then add the corresponding shared pointer to your class into `ResourcePointers.h`. While creating the accessor keep in mind that each of them must call *EnsureLoaded* to make sure the resource is actually loaded before it is to be used!

9.5 Currently existing resource types

To see all currently existing types of resources it's best to head into the doxygen documentation. Locate the *ResourceSystem::Resource* class and see all classes derived from it.

Chapter 10

Script system

Namespaces: ScriptSystem

Headers: ScriptMgr.h, ScriptRegister.h, ScriptResource.h

Source files: ScriptMgr.cpp, ScriptRegister.cpp, ScriptResource.cpp

Classes: ScriptMgr, ScriptResource

Libraries used: AngelScript

10.1 Purpose of the script system

The script system allows customizing reactions to application events such as messages sent to entities or GUI interactions without a compilation of a whole application to the users of the engine. The advantage of using scripts is an easier extension of the application which can be done even by non-professional users because in the script environment they cannot do any fatal errors that can corrupt the application and it is possible to have a full control of the script execution – for example timeout of execution prevents cycling. The disadvantage is a slower execution of scripts than a native code so an user should decide which part of system will be native coded and which can be done by scripts.

In the following sections the basics of the script language will be described as well as the connection of the script system to the rest of the engine and there will be also mentioned a process of using and extending it. In the last section there is a small glossary of used terms.

10.2 Introduction to the used script language

This section is an introduction to principles and a syntax of the script language used for the writing scripts in this engine. The script language resembles C++ language but there is some differences and limitations which are described in the next paragraphs. For further information see AngelScript documentation [2].

It is not possible to declare function prototypes, nor is it necessary as the compiler can resolve the function names anyway. For parameters sent by reference it is convenient to specify in which direction the value is passed (`&in` for passing to function, `&out` for passing from function and `&inout` for both directions) and whether is constant (`const` before type) because the compiler can optimize the calling.

It is possible to declare script classes like in C++ but there are no visibility keywords (everything is public) and all class methods must be declared with their implementation (like in Java or C#). Operator overloads are supported. Only the single inheritance is allowed but polymorphism is supported by implementing interfaces and every class method is virtual. The automatic memory management is used so it can be difficult to know exactly when the destructor is called but it is called only once.

Because pointers are not safe in a scripted environment the object handles (`class@ object;`) are used instead. They behave like smart pointers that control the life time of the object they hold. There are initialized to `null` by default and can be compared by `is` operator (`if (a is null) @a = @b;`). Every object of a script class is created as a reference type while the objects of a registered class can be created as a value type (see section 10.4.1 for more information).

Primitive data types are same as in the game engine (for example `uint8` for unsigned 8-bit integer, `float64` for 64-bit real number). Arrays are zero based and resizeable (they have methods `length()` and `resize(uint32)`). It is also possible to define C++ like enumerations and typedefs (only for primitive types).

10.3 Linking the script system to the engine

This section is about linking the script system to the rest of the application. The script system consist of the script engine to which every class and function must be registered if it should be used from script. The engine manages script modules and contexts. The script module consists of one or more files that are connected with include directives and are built together. The name

of module is the same as the name of file that includes the rest files. The script context wraps the function calling. When the application wants to call a function from a script it must create the script context, prepare it with a function ID got from the function declaration and the name of the module where the function is, add function parameters, execute it, get the return values and release it.

10.3.1 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 10.3.4) as well as all user-defined ones (see 10.4.1).

The first thing a caller of a script function must do is to obtain a function ID. It can be get from the method *ScriptMgr::GetFunctionID* that needs the name of the module where the function is and its declaration. For example if the name of 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)". The method returns the integer that means 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 all time 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 disc and builds it if necessary. If it is inconvenient that the module is built at the first call of its function, this 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 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 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 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 section 10.2 it is possible to use a conditional compilation of scripts. The method *ScriptMgr::DefineWord* adds a pre-processor define passed as the string argument. The last two methods of this class are *ScriptMgr::UnloadModule* and *ScriptMgr::ClearModules* that unload one/all previously loaded and built modules and abort all contexts in context manager. 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 needed to reload modules or free all memory used by them but when it is better not to destroy the whole script engine.

10.3.2 Binding the entity system

The script system is binded to the entity system to provide an easy work with components, entities and theirs properties from scripts. The class *EntityHandle* is registered to the script engine as a value type with a most of its methods but the work with entity properties differs from using them from a source code. There are registered common methods of this class for all defined property types (in *PropertyTypes.h*) to get and set a simple property value, to get a non-constant or constant array, to call a property function with parameters and to register and unregister a dynamic property. Methods for working with array property values (get and set a size of array, an index operator) and property function parameters (add a simple or array value as a property function parameter) are also registered. See table 10.1 for method declarations.

For example if an entity represented by its handle, that can be got by `this` global property when script is handling a message to entity (see section 10.4.2 for details), has an integer property `Integer` then the command to save it to an integer script variable will be following: `int32 n = this.Get_int32("Integer")`. If the property `Integer` does not exist in this entity or has an other type than `int32` the error message is written to the log and the default value (for `int32` 0) is returned.

Another example is using dynamic properties. Unlike other properties

An action	A declaration
EntityHandle methods	
Get a simple property value	`type Get_type(string& property_name)`
Set a simple property value	`void Set_type(string& property_name, type value)`
Get a non-constant array property value	`array_type Get_array_type(string& property_name)`
Get a constant array property value	`const array_type Get_const_array_type (string& property_name)`
Call an entity function	`void CallFunction(string& function_name, PropertyFunctionParameters& parameters)`
Register a new dynamic property	`bool RegisterDynamicProperty_type(const string& property_name, const PropertyAccessFlags flags, const string& comment)`
Unregister a dynamic property	`bool UnregisterDynamicProperty (const string& property_name)`
array_type methods	
Get a size of array	`int32 GetSize() const`
Set a size of non-constant array	`void Resize(int32 size)`
An index for a constant array	`type operator[](int32 index) const`
An index for a non-constant array	`type& operator[](int32 index)`
PropertyFunctionParameters methods	
Add a simple property as a property function parameter	`PropertyFunctionParameters operator<< (const type& value)`
Add an array property as a property function parameter	`PropertyFunctionParameters operator<< (const array_type& value)`

Table 10.1: Declarations of methods for working with entity properties where type means simple property type from *PropertyTypes.h*

dynamic ones are connected with an instance of an entity not with an entity class, they have not getters and setters and they must be registered before using. They can be used for an extension of information about the entity that should be shared by different calls of scripts. They are stored in a component Script (see section 10.4.2) which therefore must be a part of the entity. For registering an additional boolean property `Bool` on the entity held by `handle` with an access from scripts the command will be following:

```
handle.RegisterDynamicProperty_bool("Bool", PA_SCRIPT_READ |
    PA_SCRIPT_WRITE, "comment");
```

For managing entities the *EntityMgr* class is registered as no-handle object which means that only way to use it is calling methods on the return value from `gEntityMgr` global property. For example if the script should destroy the entity represented by the variable `handle`, the code will be following: `gEntityMgr.DestroyEntity(handle)`. For creation of entities the *EntityDescription* class is registered as a value type, where it is possible to specify contained components, name, kind etc., as well as method *CreateEntity* of *EntityMgr* which has this class as parameter and returns handle of created entity.

10.3.3 Binding the other systems

Beside the classes from the entity system mentioned in the section 10.3.2, some classes from other systems are also registered to the script engine. If the state of the mouse or the keyboard is needed the `gInputMgr` global property returns the input manager with the registered methods *IsKeyDown* and *GetMouseState* so for example this code returns true if the left arrow key is pressed: `gInputMgr.IsKeyDown(KC_LEFT)`. For opening another scene the *Core::Project* class is registered with *OpenScene* method that is accessible via the `gProject` global property. For getting the text data from the string manager use the `GetTextData` global functions (one accepts the group and the key parameters, another only the key parameter and it loads the text from the default group).

For handling GUI events some of GUI elements are registered to the script engine as basic reference types (see 10.4.1). The class *Window* is registered with the most necessary methods as well as its most used descendants like *ButtonBase*, *Checkbox*, *PushButton*, *RadioButton* and *Editbox*. The instances of these classes can be got from the `Window@ GetWindow(string)` global function (returns the window with the specified name), the `Window@` parameter in a GUI callback function or from some methods of these classes like `Window@ GetParent()`. When it is necessary to cast the returned handle to other class use the cast<> operator. For example when the `Window@ window` variable contains the handle to the *Editbox* class, the `editbox` variable in the code `Editbox@ editbox = cast<Editbox>(window)` will contain the cast instance, otherwise it will contain `null` handle (that can be tested by `editbox !is null` condition).

10.3.4 Another registered classes and functions

Few more base classes and functions are registered to the script engine. For storing a text data there are two classes registered as a value type. The first one is a standard C++ *string* class registered with a most of its method and =, +=, + operators for all property types from *PropertyTypes.h*, the second one is a class *StringKey* which is an integer representation of text for faster comparing and assignment and which can be construct from *string* and which can be cast to *string* by the *ToString* method.

The `void Println(const type& message)` function for all property types, which prints the string in the parameter to the log or the console so it is useful for debugging scripts, is also registered as well as some math functions and other additional types. For a complete reference of registered classes, global functions etc. see ScriptRegister.pdf.

10.4 Using and extending the script system

This section is about using the script system and extending it. If a new property type is added to the entity system or a new system which should be accessible from scripts is created then it is necessary to register new classes and functions to the script system. For better understanding of the script system the sample component which provides a script handling of messages to entities was created and it is described in this section.

10.4.1 Registering new classes and functions

If it is necessary to register a new class with its methods or a global function, the registration function should be implemented in the *ScriptRegister.cpp* file and called by the global function *RegisterAllAdditions* with the pointer to script engine as a parameter because a registration is done by calling methods of it.

The registration function should first declare the integer variable `int32 r` to which should be assign all return values from calling registration methods and after each calling the `OC_SCRIPT_ASSERT()` should be used to check if the registration is successful (it checks the variable `r` to be positive). First it is necessary to register the whole class which is done by the *RegisterObjectType* method. The first parameter is a name of class in a script, the second is a size of class (use `sizeof(class)` for register a class as value, `0` otherwise) and the last parameter is flag. See table 10.2 or documentation of AngelScript [2] for some flag combination.

A class type	Flags
A primitive value type without any special management	`asOBJ_VALUE | asOBJ_POD | asOBJ_APP_CLASS_CA`
A value type needed to be properly initialized and uninitialized	`asOBJ_VALUE | asOBJ_APP_CLASS_CA`
A basic reference type	`asOBJ_REF`
A single-reference type (singleton)	`asOBJ_REF | asOBJ_NOHANDLE`

Table 10.2: Flags in register function method

After the registration the whole class it is possible to register methods of it with the *RegisterObjectMethod* method. The first parameter is the name of the registered class, the second is the declaration of method in a script, the third is a pointer to the C++ function that should be called and the last is the calling convention. The pointer to C++ function should be get from one of the four macros described in the table 10.3 (`PR` variants must be used when functions or methods are overloaded) and the possible calling conventions are listed in the table 10.4. For example the registering of method

bool EntityHandle::IsValid() const as a method of the `EntityHandle` script class the code will be following:

```
r = engine->RegisterObjectMethod("EntityHandle", "bool IsValid() const",
  asMETHOD(EntityHandle, IsValid), asCALL_THISCALL); OC_SCRIPT_ASSERT();
```

A macro declaration	An example
`asFUNCTION(global_function_name)`	`asFUNCTION(Add)`
`asFUNCTIONPR(global_function_name, (parameters), return_type)`	`asFUNCTIONPR(Add, (int), void)`
`asMETHOD(class_name, method_name)`	`asMETHOD(Object, Add)`
`asMETHODPR(class_name, method_name, (parameters), return_type)`	`asMETHOD(Object, Add, (int), void)`

Table 10.3: Used macros for a function and method registration

A call convention	An use case
`asCALL_CDECL`	`A cdecl function (for global functions)`
`asCALL_STDCALL`	`A stdcall function (for global functions)`
`asCALL_THISCALL`	`A thiscall class method (for class methods)`
`asCALL_CDECL_OBJLAST`	`A cdecl function with the object pointer as the last parameter`
`asCALL_CDECL_OBJFIRST`	`A cdecl function with the object pointer as the first parameter`

Table 10.4: Used calling conventions for a function and method registration

Operators are registered as common methods but they have a special script declaration name such as `opEquals` or `opAssign`. See AngelScript documentation [2] for further operator names.

If the class has some special constructors or destructor, they need to be registered by the *RegisterObjectBehaviour* method which has same parameters as the *RegisterObjectMethod* except an inserted second parameter that means which behavior is registered (see table 10.5 for possible values). They must be registered by proxy functions that take a pointer to an object as the last argument (so the `asFUNCTION` macro and the `asCALL_CDECL_OBJLAST` calling convention are used) and fill it with a constructed object or call a correct destructor on it.

If the class is registered as a basic reference type, it needs to have a reference counter, a factory function and add-reference and release methods. These methods are registered by *RegisterObjectBehaviour* method as well as constructor and destructor. The single-reference type does not need these methods but it cannot be passed as function parameter or stored to variable.

It is also possible to register class properties by the *RegisterObjectProperty* method but more portable is to register theirs getters and setters as class methods with strict declarations (`type get_propname() const` and `void set_propname(type)`) so they can be used almost as registered directly (`object.propname`).

A C++ function declaration	A behavior
`void ObjectDefaultConstructor(Object* self)`	`asBEHAVE_CONSTRUCT`
`void ObjectCopyConstructor(Object& other, Object* self)`	`asBEHAVE_CONSTRUCT`
`void ObjectDestructor(Object* self)`	`asBEHAVE_DESTRUCT`
`Object *ObjectFactory()`	`asBEHAVE_FACTORY`
`void Object::Addref()`	`asBEHAVE_ADDREF`
`void Object::Release()`	`asBEHAVE_RELEASE`

Table 10.5: Possible behaviors for the *RegisterObjectBehaviour* method

Global functions are registered by the *RegisterGlobalFunction* method that has almost same parameters as the *RegisterObjectMethod* (does not have the first), global properties by *RegisterGlobalProperty* method. There are also methods for registering enumerations (*RegisterEnum*, *RegisterEnumValue*) and typedefs for primitive types (*RegisterTypedef*) that are easy to use. Further information about registering types can be found at the AngelScript documentation [2], examples of using are in the *ScriptRegister.cpp* file.

10.4.2 Sample component

As an example how to link the script and the entity system the Script component was created. The purpose of this 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 an 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 names that are searched for message handlers, the second represents the maximum script execution time in milliseconds after that 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 provides a support for scripts that should be called periodically and that should remember theirs 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 `Times` 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 `States` 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 mil-

liseconds. The right time and state are got thanks to the `CurrentArrayIndex` property.

The most important method of this component is the *Script::HandleMessage* that accept 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 depended 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 `GetCurrentEntityHandle` function 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.

10.5 Glossary

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

Value type – a primitive type (integer or real number, boolean value, enumeration or string) or an object that is copied on an assignment or a passing to or from a function

Reference type – an object that is assigned or passed to a function only through a pointer on it

Object handle – an equivalent of a reference in C++ that counts references on an object

Script class – a class defined and implemented in a script file, it is always used as a reference type

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 with include directives, which are managed and built together and have a common namespace

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

Script context – an object that wraps a 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 use 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

Chapter 11

String system

Namespaces: StringSystem

Headers: FormatText.h, StringMgr.h, TextData.h, TextResource.h

Source files: FormatText.cpp, StringMgr.cpp, TextResource.cpp

Classes: FormatText, StringMgr, TextResource

Libraries used: CEGUI

11.1 Purpose of the string system

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

In the following sections the required format of text files and the directory layout will be described as well as switching the languages and loading and formating the desired localized text. In the last section there is a small glossary of used terms.

11.2 Format of text files

One text file consists of lines of text items and comments. The pattern of a text item is `key=Value`, where `key` is an ID that is used for indexing a text represented by `Value` from the application. The first text item in a file defines a group for the following text items if its `key` is `group`, otherwise the group is set to the default one. An ID must be unique within a group, must

not begin with the # character and contain the = character and the same ID in the same group should represented the same text in each language.

A Value part can contain %x character sequences where x is a number from 1 to 9. When a text with these sequences is loaded it is possible to replace them with another text in order according to the specific number. See the section 11.5 for more information.

A comment is a text that begins with the # character and it is ignored by the application. Every text item and each comment must be on it's own line without any leading white characters. Here is an example of a correct text file:

```
# the name of the group to which the following texts belong
group=ExampleGroup
# comment to the text item below
example_key=Example text.
another_key=Another text.
```

The encoding of text files must be ASCII or UTF8. It is possible to end lines in Windows (\r\n) or UNIX (\n) style.

11.3 Directory layout

The directory in which the text files are stored must have a specific layout. In the root directory there should be files with default language texts. If any ID of a text item of a specific group is not contained in a chosen language subdirectory, the text item of a default language will be used. In this directory there should be also subdirectories representing language specific texts. It is recommended that their names correspond with ISO 639-1 codes [3] of their language (i.e. en for English, fr for French etc.). These subdirectories should contain the same files as the root directory which should consist of the language specific text items that will be preferably used if the corresponding language is chosen.

If any text item varies among various countries that use a common language it should be placed to the same file in subdirectories of a language directory. It is recommended that their names correspond with ISO 3166-1 alpha-2 codes [4] of a country they represent (i.e. US for United States, GB for United Kingdom etc.). Here is an example of a directory layout:

```
-en
 -GB
  -textfile.str (1)
```

```
 -US
  -textfile.str (2)
 -textfile.str  (3)
-fr
 -textfile.str  (4)
-textfile.str   (5)
```

Assume that the files from the above example have a following content:

```
(1)
group=Group
country_specific=Country
(3)
group=Group
language_specific=Language
country_specific=Language
(5)
group=Group
default_string=Default
language_specific=Default
country_specific=Default
```

The texts given from invoking the following IDs of group `Group` for specific languages are shown in the table 11.1.

Text ID	Text value
Language: en-GB	
default_string	Default
language_specific	Language
country_specific	Country
Language: en	
default_string	Default
language_specific	Language
country_specific	Language
Language: default	
default_string	Default
language_specific	Default
country_specific	Default

Table 11.1: Texts given from IDs with various language settings

11.4 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 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 described in the section 11.3. Both are initialized when the application starts and destroyed on an application shutdown. The class provides static methods returning 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* that expects the language and the country code (both can be omitted) which meaning is widely explained in the section 11.3. It removes old text items and loads new ones according to a specific language setting to the memory and divides them to the desired groups. This method can be called during the whole execution of application but the other systems must refresh their data themselves.

There are several methods in the class for getting the text data according to its group and ID 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 and ID does not exist, an empty string is returned and an error message is written to the log.

11.5 Using a 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 with 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 formated 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 then 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.
```

11.6 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 [3].

Country code – two-character acronym representing a country according to ISO 3166-1 alpha-2 [4].

ASCII – an encoding of text that uses only numbers, characters of English alphabet and a few symbols and non-printing control characters

UTF8 – a backward compatible ASCII encoding extension that is able to represented any character in the Unicode standard in 1 to 4 bytes

Chapter 12

Platform setup

Namespaces: none

Headers: BasicTypes.h, ComplexTypes.h, Containers.h, Memory_post.h, Memory_pre.h, Platform.h, Settings.h

Source files: ComplexTypes.cpp

Classes: none

Libraries used: none

12.1 Introduction

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.

12.2 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 behaviour 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.h` defines the memory allocation method used by the project and includes their implementation from the memory subsystem (the `Memory` directory).

Chapter 13

Utils

Namespaces: Utils, Reflection

Headers: the Utils directory

Source files: the Utils directory

Classes: Array, DataContainer, GlobalProperties, Singleton, StateMachine, StringKey, Timer, AbstractProperty, Property, PropertyFunctionParameter, PropertyFunctionParameters, PropertyHolder, PropertyList, TypedProperty, RTTI, RTTINullClass, RTTIBaseClass, RTTIGlue

Libraries used: none

13.1 Introduction

During the development of a game you usually need several helper classes and methods such as those for math or containers. This kind of stuff does not fit anywhere because it is too general. Placing it in a specific subsystem would make it unusable anywhere else. So it is best to aggregate this in one place.

13.2 Categories

There are several categories of utilities: containers (i.e. tree), math functions (i.e. hash) and design patterns (i.e. singleton). The utilities are usually simple enough to understand, so it's best to read the code and/or the doxygen documentation. However, a few of them are a bit more complicated. That's why they're described in the following sections.

13.2.1 RTTI

RTTI is a shortage for *Run-Time Type Information*. It is a mechanism which 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. To make a use of it you must derive your class from *RTTIGlue* which takes two template arguments. The first one is your class and the second one is its predecessor in the RTTI hierarchy. If there is none or you do not want to specify that inherit *RTTIBaseClass* instead of *RTTIGlue*. The mechanism will automatically call the static *RegisterReflection* function of your class if defined. The function is called only once during the startup. Here you may adjust attributes of the class or add new ones.

13.2.2 Properties

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

To make use of *Properties* your class must be already using *RTTI*. Create a getter and setter method for each of your data you want to provide as properties. Define the static *RegisterReflection* function (see the RTTI section for details) and inside its body call *RegisterProperty* providing pointers to the data getters and setters. For details about this function see the doxygen docs. To access the data of an instance use `GetRTTI()->GetProperty()` on the instance. It will give you *PropertyHolder* on which you can finally call the templatized *GetValue* or *SetValue* methods.

You can also set a custom function as a special property. The function must take *PropertyFunctionParameters* as a single parameter and return `void`. You can register it using *RegisterFunction*. The function can be then used in a similar way as the common properties, but you use *CallFunction* instead of *SetValue*.

Bibliography

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

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

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

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

List of Figures

List of Tables

Appendix A

Script system registered object

A.1 Purpose of this document

This document is an enumeration of classes, global functions etc. that are registered to the script engine so they can be used from scripts. It is divided to the section about objects from the game engine and to the section about additional user-defined objects.

A.2 Integral registered objects

This section is about registered objects from the game engine so it should not be edited until the game engine is changed.

A.2.1 Classes

StringKey

This class serves as a key into maps and other structures where we want to index data using strings, but we need high speed as well. The string value is hashed and the result is then used as a decimal representation of the string. This class is registered as a value type.

Constructors

- StringKey() – default constructor
- StringKey(const StringKey &in) – copy constructor
- StringKey(const string &in) – constructs the key from a standard string

Operators and methods

- bool operator=(const StringKey &in) const – assignment operator
- StringKey& operator==(const StringKey &in) – equality operator
- string ToString() const – converts the key to a string

array_T

This is a group of classes parametrized by a type T from PropertyTypes.h that serve as an array of values of a type T corresponding to properties of the Array<T>* type. For example an array of 32-bit signed integer will be declared as `array_int32`. These classes are not compatible with an script array defined as for example `int32[]` but they have similar methods. This class is registered as a value type.

Constructors

- array_T() – default constructor that should not be used, instances of these classes are got from Get_array_T and Get_const_array_T methods of EntityHandle class

Operators and methods

- T& operator[](int32) – write accessor to an array item
- T operator[](int32) const – read accessor to an array item
- int32 GetSize() const – returns a size of the array
- void Resize(int32) – resize an array to a new size

PropertyFunctionParameters

This class represents generic parameters passed to a function accessed via the properties system. Thanks to the << operators the function parameter can be passed as `PropertyFunctionParameters() << param1 << param2 ...`. This class is registered as a value type.

Constructors

- PropertyFunctionParameters() – constructs new empty parameters

Operators and methods

- PropertyFunctionParameters& operator=(const PropertyFunctionParameters &in) – assignment operator
- bool operator==(const PropertyFunctionParameters &in) const – equality operator
- PropertyFunctionParameters operator<<(const T &in) const – add parameter of type T from PropertyTypes.h
- PropertyFunctionParameters operator<<(const array_T &in) const – add parameter of type array_T

EntityHandle

This class represents one unique entity in the entity system. This class is registered as a value type.

Constructors

- EntityHandle() – default constructor will initialize the handle to an invalid state
- EntityHandle(const EntityHandle &in) – only the copy constructor is enabled, new entities should be added only by the EntityMgr

Operators and methods

- EntityHandle& operator=(const EntityHandle &in) – assignment operator
- bool operator==(const EntityHandle &in) – equality operator
- bool IsValid() const – returns true if this handle is valid (not null)
- bool Exists() const – returns true if this entity still exists in the system
- EntityID GetID() const – returns the internal ID of this entity
- string GetName() const – returns the name of this entity
- EntityTag GetTag() const – returns the tag of this entity
- void SetTag(EntityTag) – sets the tag of this entity

- eEntityMessageResult PostMessage(const eEntityMessageType, PropertyFunctionParameters = PropertyFunctionParameters()) – sends a message to this entity (message type, parameters)
- void CallFunction(string &in, PropertyFunctionParameters &in) – calls a function on an entity of a specific name and with specific parameters
- T Get_T(string &in) – gets a value of an entity property of a specific name and type T from PropertyTypes.h
- void Set_T(string &in, T) – sets a value of an entity property of a specific name and type T from PropertyTypes.h
- array_T Get_array_T(string &in) – gets a non-constant value of an entity property of a specific name and type of array_T
- const array_T Get_const_array_T(string &in) – gets a constant value of an entity property of a specific name and type of array_T
- bool RegisterDynamicProperty_T(const string &in, const PropertyAccessFlags, const string &in) – registers an entity dynamic property of a specific name, access and comment
- bool UnregisterDynamicProperty(const string &in) const – unregisters an entity dynamic property of a specific name

EntityDescription

This class contains all info needed to create one instance of an entity. It is basically a collection of component descriptions. This class is registered as a value type.

Constructors

- EntityDescription() – default constructor

Operators and methods

- void Reset() – clears everything, call this before each subsequent filling of the description
- void AddComponent(const eComponentType) – adds a new component specified by its type

- void SetName(const string &in) – sets a custom name for this entity
- void SetKind(const eEntityDescriptionKind) – sets this entity to be an ordinary entity or a prototype
- void SetPrototype(const EntityHandle) – sets the prototype the entity is to be linked to
- void SetPrototype(const EntityID) – sets the prototype the entity is to be linked to
- void SetDesiredID(const EntityID) – sets a desired ID for this entity, it does not have to be used by EntityMgr

EntityMgr

This class manages all game entities. It is registered as a no-handle reference so the only way to use it is to get reference from a global property.

Operators and methods

- EntityHandle CreateEntity(EntityDescription &in) – creates a new entity accordingly to its description and returns its handle
- EntityHandle InstantiatePrototype(const EntityHandle, const string &in) – creates a new entity from a prototype with a specific name
- EntityHandle DuplicateEntity(const EntityHandle, const string &in) – duplicates an entity with a specific new name
- void DestroyEntity(const EntityHandle) – destroys a specified entity if it exists
- bool EntityExists(const EntityHandle) const – returns true if the entity exists
- EntityHandle FindFirstEntity(const string &in) – returns EntityHandle to the first entity of a specified name
- EntityHandle GetEntity(EntityID) const – return EntityHandle of the entity with specified ID
- bool IsEntityInited(const EntityHandle) const – returns true if the entity was fully initialized

- bool IsEntityPrototype(const EntityHandle) const – returns true if the entity is a prototype
- void LinkEntityToPrototype(const EntityHandle, const EntityHandle) – assigns the given entity to the prototype
- void UnlinkEntityFromPrototype(const EntityHandle) – destroys the link between the component and its prototype
- bool IsPrototypePropertyShared(const EntityHandle, const StringKey) const – returns true if the property of the prototype is marked as shared (and thus propagated to instances)
- void SetPrototypePropertyShared(const EntityHandle, const StringKey) – marks the property as shared among instances of the prototype
- void SetPrototypePropertyNonShared(const EntityHandle, const StringKey) – marks the property as non shared among instances of the prototype
- void UpdatePrototypeInstances(const EntityHandle) – propagates the current state of properties of the prototype to its instances
- bool HasEntityProperty(const EntityHandle, const StringKey, const PropertyAccessFlags) const – returns true if the entity has the given property
- bool HasEntityComponentProperty(const EntityHandle, const ComponentID, const StringKey, const PropertyAccessFlags) const – returns true if the component of the entity has the given property
- void BroadcastMessage(const eEntityMessageType, PropertyFunctionParameters = PropertyFunctionParameters()) – sends a message to all entities
- bool HasEntityComponentOfType(const EntityHandle, const eComponentType) – returns true if the given entity has a component of the given type
- int32 GetNumberOfEntityComponents(const EntityHandle) const – returns the number of components attached to the entity
- ComponentID AddComponentToEntity(const EntityHandle, const eComponentType) – adds a component of the specified type to the entity, returns an ID of the new component

- void DestroyEntityComponent(const EntityHandle, const ComponentID) – destroys a component of the entity

MouseState

This structure contains the state of the mouse device at a specific point of time. This class is registered as a simple value type.

Properties

- int32 x – x coordinate
- int32 y – y coordinate
- int32 wheel – wheel position
- uint8 buttons – pressed buttons

InputMgr

This class manages the game input. It is registered as a no-handle reference so the only way to use it is to get reference from a global property.

Operators and methods

- void CaptureInput() – updates the state of the manager and processes all events
- bool IsKeyDown(const eKeyCode) const – returns true if a specified key is down
- bool IsMouseButtonPressed(const eMouseButton) const – returns true if a specified button of the mouse is pressed
- MouseState& GetMouseState() const – returns the current state of the mouse

Project

This class represents the current project. It is registered as a no-handle reference so the only way to use it is to get reference from a global property.

Operators and methods

- bool OpenScene(const string &in) – opens the scene with given filename

- bool OpenSceneAtIndex(int32) – opens the scene at the given index in the scene list
- uint32 GetSceneCount() const – returns the number of scenes
- int32 GetSceneIndex(const string &in) const – returns the index of the scene with a specific name, -1 if does not exist
- string GetOpenedSceneName() const – returns name of the opened scene, or empty string if no scene is opened
- string GetSceneName(int32) const – returns name of the scene at the given index in the scene list

CEGUIString

This class represents string in the CEGUI library. It is registered as a value type and it is implicitely cast from/to a common string so no other methods are needed.

Window

This class is the base class for all GUI elements. It is registered as a basic reference and its handle can be cast to all of its desceants.

Operators and methods

- const CEGUIString& GetName() const – returns the name of this window
- const CEGUIString& GetType() const – returns the type name for this window
- bool IsDisabled() const – returns whether the window is currently disabled (does not inherit state from ancestor windows)
- bool IsDisabled(bool) const – returns whether the window is currently disabled (specify whether to inherit state from ancestor windows)
- bool IsVisible() const – returns true if the window is currently visible (does not inherit state from ancestor windows)
- bool IsVisible(bool) const – returns true if the window is currently visible (specify whether to inherit state from ancestor windows)

- bool IsActive() const – returns true if this is the active window (may receive user inputs)
- const CEGUIString& GetText() const – returns the current text for the window
- bool InheritsAlpha() const – returns true if the window inherits alpha from its parent(s)
- float32 GetAlpha() const – returns the current alpha value set for this window (between 0.0 and 1.0)
- float32 GetEffectiveAlpha() const – returns the effective alpha value that will be used when rendering this window
- Window GetParent() const – returns the parent of this window
- const CEGUIString& GetTooltipText() const – returns the current tooltip text set for this window
- bool InheritsTooltipText() const – returns whether this window inherits tooltip text from its parent when its own tooltip text is not set
- void SetEnabled(bool) – sets whether this window is enabled or disabled
- void SetVisible(bool) – sets whether this window is visible or hidden
- void Activate() – activates the window giving it input focus and bringing it to the top of all windows
- void Deactivate() – deactivates the window
- void SetText(const CEGUIString& in) – sets the current text string for the window
- void SetAlpha(float32) – sets the current alpha value for this window
- void SetInheritsAlpha(bool) – sets whether this window will inherit alpha from its parent windows
- void SetTooltipText(const CEGUIString& in) – sets the tooltip text for this window
- void SetInheritsTooltipText(bool) – sets whether this window inherits tooltip text from its parent when its own tooltip text is not set

ButtonBase

This class is the base class for all button GUI elements. It is registered as a basic reference and its handle can be cast to all of its desceants and the Window class. Beside these methods it has all methods of the Window class.

Operators and methods

- bool IsHovering() const – returns true if user is hovering over this widget
- bool IsPushed() const – returns true if the button widget is in the pushed state

Checkbox

This class representes the checkbox GUI element. It is registered as a basic reference and its handle can be cast to the Window and ButtonBase classes. Beside these methods it has all methods of the Window and ButtonBase classes.

Operators and methods

- bool IsSelected() const – returns true if the checkbox is selected (has the checkmark)
- void SetSelected(bool) – sets whether the checkbox is selected or not

PushButton

This class representes the push button GUI element. It is registered as a basic reference and its handle can be cast to the Window and ButtonBase classes. It has all methods of the Window and ButtonBase classes and none more.

RadioButton

This class representes the radio button GUI element. It is registered as a basic reference and its handle can be cast to the Window and ButtonBase classes. Beside these methods it has all methods of the Window and ButtonBase classes.

Operators and methods

- bool IsSelected() const – returns true if the checkbox is selected (has the checkmark)
- void SetSelected(bool) – sets whether the checkbox is selected or not
- uint32 GetGroupID() const – returns the group ID assigned to this radio button
- void SetGroupID(uint32) – sets the group ID for this radio button

Editbox

This class representes the editbox GUI element. It is registered as a basic reference and its handle can be cast to the Window class. Beside these methods it has all methods of the Window class.

Operators and methods

- bool HasInputFocus() const – returns true if the editbox has input focus
- bool IsReadOnly() const – returns true if the editbox is read-only
- bool IsTextMasked() const – returns true if the text for the editbox will be rendered masked
- bool IsTextValid() const – returns true if the editbox text is valid given the currently set validation string
- const CEGUIString& GetValidationString() const – returns the currently set validation string
- uint32 GetMaskCodePoint() const – returns the utf32 code point used when rendering masked text
- uint32 GetMaxTextLength() const – returns the maximum text length set for this editbox
- void SetReadOnly(bool) – specifies whether the editbox is read-only
- void SetTextMasked(bool) – specifies whether the text for the editbox will be rendered masked
- void SetValidationString(const CEGUIString &in) – sets the text validation string

- void SetMaskCodePoint(uint32) – sets the utf32 code point used when rendering masked text
- void SetMaxTextLength(uint32) – sets the maximum text length for this editbox

A.2.2 Global functions

Declaration and comment

- Window@ GetWindow(string) – returns the window with the given name or null if such a window does not exist
- void Println(const T &in) – writes a message to the log or the console (converts all types T from PropertyTypes.h to string)
- const string GetTextData(const StringKey &in, const StringKey &in) – returns a text data specified by a given search key and group
- const string GetTextData(const StringKey &in) – returns a text data specified by a given search key from a default group

A.2.3 Global properties

Declaration and comment

- EntityHandle this – returns the handle of entity that calls the current function, invalid handle if no entity calls it
- EntityMgr& gEntityMgr – the reference to the entity manager on which it is possible to call methods, this reference cannot be stored to a variable or passed to a function
- InputMgr& gInputMgr – the reference to the input manager on which it is possible to call methods, this reference cannot be stored to a variable or passed to a function
- Project& gProject – the reference to the current project on which it is possible to call methods, this reference cannot be stored to a variable or passed to a function

A.2.4 Enumerations

Type { values } – comment

- eEntityMessageType { ... } – this is user-defined, the types are got from EntityMessageTypes.h
- eEntityMessageResult { RESULT_IGNORED, RESULT_OK, RESULT-_ERROR } – result receives after sending out a message to entities (message is ignored, processed well or an error occured)
- eEntityDescriptionKind { EK_ENTITY, EK_PROTOTYPE } – kind of an entity (an ordinary entity, a prototype)
- eComponentType { ... } – this is user-defined, the types are got from _ComponentTypes.h
- ePropertyAccess { PA_EDIT_READ = 1<<1, PA_EDIT_WRITE = 1<<2, PA_SCRIPT_READ = 1<<3, PA_SCRIPT_WRITE = 1<<4, PA_INIT = 1<<5, PA_FULL_ACCESS = 0xff } – restrictions of access which can be granted to a property (the property can be read/written from editor/scripts/during the component initialization or full access is granted)
- eKeyCode { ... } – this is defined in KeyCodes.h
- eMouseButton { MBTN_LEFT, MBTN_RIGHT, MBTN_MIDDLE, MBTN_UNKNOWN } – all possible buttons of the mouse device

A.2.5 Typedefs

New type = old type

- float32 = float
- float64 = double
- EntityID = int32
- EntityTag = uint16
- ComponentID = int32
- PropertyAccessFlags = uint8

A.3 Additional registered objects

This section is about additional registered objects so it should be edited when new objects are registered to the script engine.

A.3.1 Classes

Vector2

This class represents 2D vector and it is registered as a value type.

Properties

- float32 x – x coordinate
- float32 y – y coordinate

Constructors

- Vector2() – default constructor (x = y = 0)
- Vector2(const Vector2 &in) – copy constructor
- Vector2(float32 x, float32 y) – constructor using coordinates

Operators and methods

- void operator+=(const Vector2 &in) – add a vector to this vector
- void operator-=(const Vector2 &in) – subtract a vector from this vector
- void operator*=(float32) – multiply this vector by a scalar
- Vector2 operator-() const – negate this vector
- bool operator==(const Vector2 &in) const – equality operator
- Vector2 operator+(const Vector2 &in) const – returns a sum of this and argument vector
- Vector2 operator-(const Vector2 &in) const – returns a difference of this and argument vector
- Vector2 operator*(float32) const – returns a product of a scalar and this vector

- float32 Length() const – get the length of this vector (the norm)
- float32 LengthSquared() const – get the length squared
- void Set(float32, float32) – set this vector to some specified coordinates
- void SetZero() – set this vector to all zeros
- float32 Normalize() – converts this vector into a unit vector and returns the length
- bool IsValid() const – returns whether this vector contain finite coordinates
- float32 Dot(const Vector2 &in) const – returns a scalar product of this and argument vector

Color

This class represents 32-bit color and it is registered as a value type.

Properties

- uint8 r – red component of color
- uint8 g – green component of color
- uint8 b – blue component of color
- uint8 a – alpha component of color

Constructors

- Color() – default constructor (r = g = b = 0, a = 255)
- Color(uint8 r, uint8 g, uint8 b, uint8 a = 255) – parameter constructor
- Color(uint32 color) – construct the color from 32-bit number

Operators and methods

- bool operator==(const Color &in) const – equality operator
- uint32 GetARGB() const – return 32-bit representation of the color

A.3.2 Global functions

Declaration and comment

- float32 Random(const float32, const float32) – returns the random real number between the first and the second parameter
- float32 Abs(const float32)
- int32 Abs(const int32)
- float32 Min(const float32, const float32)
- int32 Min(const int32, const int32)
- Vector2 Min(const Vector2, const Vector2)
- float32 Max(const float32, const float32)
- int32 Max(const int32, const int32)
- Vector2 Max(const Vector2, const Vector2)
- int32 Round(const float32)
- int64 Round(const float64)
- int32 Floor(const float32)
- int32 Ceiling(const float32)
- float32 Sqr(const float32)
- float32 Sqrt(const float32)
- float32 Distance(const Vector2 &in, const Vector2 &in)
- float32 DistanceSquared(const Vector2 &in, const Vector2 &in)
- float32 AngleDistance(const float32, const float32)
- float32 Sin(const float32)
- float32 Cos(const float32)
- float32 Tan(const float32)

- float32 ArcTan(const float32)
- float32 ArcSin(const float32)
- float32 Dot(const Vector2 &in, const Vector2 &in)
- float32 Cross(const Vector2 &in, const Vector2 &in)
- float32 Clamp(const float32, const float32, const float32) – returns the first parameter if it is between the second and the third one, the second one if the first one is lesser than the second one and the third one if the first on is greater than the third one
- float32 ClampAngle(const float32) – calls Clamp(param, 0, 2*PI)
- bool IsAngleInRange(const float32, const float32, const float32)
- float32 Wrap(const float32, const float32, const float32) – substracts the difference of the third and the second parameter from the first one until it is lesser than the third one and adds the difference of the third and the second parameter to the first one until it is greater than the second one
- float32 WrapAngle(const float32) – calls Wrap(param, 0, 2*PI)
- float32 Angle(const Vector2 &in, const Vector2 &in)
- float32 RadToDeg(const float32)
- Vector2 VectorFromAngle(const float32, const float32) – creates the vector with the angle from the first parameter and the size from the second one
- bool IsPowerOfTwo(const uint32)
- float32 ComputePolygonArea(Vector2[] &in)
- int32 GetState() – returns the state of script when it is called from the OnAction() handler, an error otherwise
- void SetAndSleep(int32, uint64) – sets the state of script to the first argument and the time of the next execution to the current time plus the second argument in milliseconds when it is called from the OnAction() handler, throws an error otherwise

A.3.3 Global properties

Declaration and comment

- float32 PI - the PI constant