Documentation of the Ocerus project

Lukas Hermann, Ondrej Mocny, Tomas Svoboda, Michal Cevora

February 15, 2011

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 the section 3.3. This can leads to defining new kinds of resources that can be created according to the section 9.2.4.

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 (14) 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.

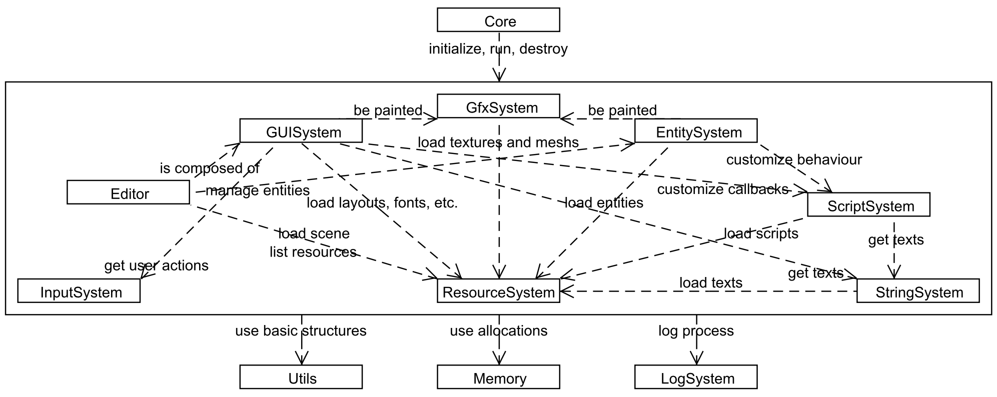

Figure 1.1: Dependencies among the systems

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.

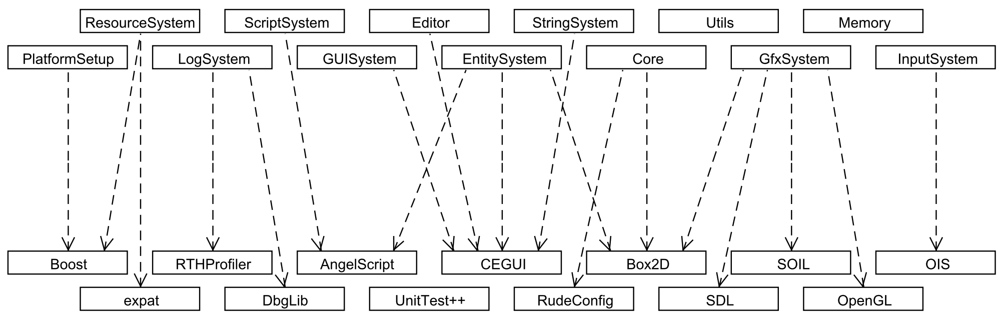

Figure 1.2: Library dependencies of the project systems

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

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

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

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

In the following chapters each of the project systems will be described from 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, invoke their update and draw methods in the main loop and in the end correctly finalize them.

In the following sections the class representing the application as well as the classes corresponding to the application states (loading screen, game), configuration and project management will be introduced. In the last section there is a small glossary of used terms.

2.2 Application

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

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

2.3 Game

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

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.

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

2.4 Loading screen

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

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

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

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

2.5 Configuration

The *Core::Config* class allows storing a configuration data needed by various parts of the program. 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, LayerMgr.h and entity components

Source files: Component.cpp, ComponentEnums.cpp, ComponentMgr.cpp, EntityDescription.cpp, EntityHandle.cpp, EntityMessage.cpp, EntityMgr.cpp, EntityPicker.cpp, LayerMgr.cpp and entity components

Classes: Component, EntityComponentsIterator, ComponentMgr, EntityDescription, EntityHandle, EntityMessage, EntityMgr, EntityPicker, LayerMgr and entity components

Libraries used: Box2D, AngelScript, CEGUI

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, a prototype, another entity 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, to invoke an update of instances of a specific prototype and to create a prototype from a specific entity. Finally there are methods for getting entity properties even of a specific component (in case of two or more properties of a same name in different components), for registering and unregistering dynamic properties, for posting and broadcasting messages to entities and for adding, listing and removing components of a specific entity.

3.2.3 Entity picker

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

3.2.4 Layer manager

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

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

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 lead 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 *Get-Parameter(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.

Already implemented components are documented along with the corresponding systems that use them.

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

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

Chapter 4

Gfx system

Namespaces: GfxSystem

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

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

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

Libraries used: SDL, OpenGL, SOIL, Box2D

4.1 Purpose of the graphic system

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

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

4.2 Graphic viewport and render target

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

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

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

4.3 Renderer and scene manager

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

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

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

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

4.4 Application window

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

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

4.5 Mesh and texture

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

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

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

4.6 Glossary

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

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

Render target – a pair or a viewport and a camera

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

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

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

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

Chapter 5

GUI system

Namespaces: GUISystem

Headers: CEGUICommon.h, CEGUIForwards.h, CEGUIResource.h, FolderSelector.h, GUIConsole.h, GUIMgr.h, MessageBox.h, PopupMgr.h, PromptBox.h, ResourceProvider.h, ScriptProvider.h, TabNavigator.h, VerticalLayout.h, ViewportWindow.h

Source files: CEGUIResource.cpp, FolderSelector.cpp, GUIConsole.cpp, GUIMgr.cpp, MessageBox.cpp, PopupMgr.cpp, PromptBox.cpp, ResourceProvider.cpp, ScriptProvider.cpp, TabNavigation.cpp, VerticalLayout.cpp, ViewportWindow.cpp

Classes: CEGUIResource, FolderSelector, GUIConsole, GUIMgr, MessageBox, PopupMgr, PromptBox, 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 the GUI system is the *GUISystem::GUIMgr* which is a connector for drawing, input handling and resource and script providing between the CEGUI library and the engine. During its creation it creates a CEGUI renderer, connects the resource manager with the CEGUI system via the *GUISystem::ResourceProvider* class (see section 5.2.2), connects the script manager with the CEGUI system via the *GUISystem::ScriptProvider* class (see section 5.2.3), provides itself as an input and screen listener and creates the GUI console which is then accesible via the *GUIMgr::GetConsole* method (see section 5.2.7). 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 layouts see section 5.2.4.

There are two methods called in the application main loop. First the *GUIMgr::Update* updates time of GUI system, then the *GUIMgr::Render-GUI* 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 Script provider

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

This transformation is implemented by the *GUISystem::ScriptCallback* class which stores the function name given in its constructor and which calls an appropriate script function as an callback. It calls a function from the script module associated with the GUI layout component of the layout which gets the event or a function from the `GuiCallback.as` module as a default. For more information see section 5.3.2.

5.2.4 Layouts

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

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

ID) and creates and shows an appropriate message box. A similar concept have the *PromptBox* class providing a modal dialog that asks for a text input from the user and the *GUISystem::FolderSelector* class providing a modal dialog for selecting a folder.

Another example is the *GUISystem::VerticalLayout* class that helps to keep GUI elements positioned in a vertical layout and automatically repositions them when one of them changes its size. In its constructor the container in which all child elements should be managed is specified, 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.

5.2.5 Viewports

The *GUISystem::ViewportWindow* class represents a viewport window with a frame where a scene is rendered by the graphic system. For defining a position, an angle and a zoom of a view of a scene 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*.

5.2.6 Popup menus

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

5.2.7 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 typed commands is stored and can be revealed by up and down arrows. The console can be shown or hide by *GUIConsole::ToggleConsole* method.

5.3 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 fulfill its specification which is widely described in its documentation [21] 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 a physical window type and a window look (rendering, fonts, imagesets) a 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 the 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 occurrence 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 are the property `Visible` indicating whether the layout is visible, the property `Enabled` indicating whether layout elements reacts on an user input and the property `Scheme` referring to the GUI scheme file that is loaded before the layout file.

On its initialization it loads the scheme and the layout to the GUI system and it calls script functions with the declarations `void OnInit()` and `void OnPostInit()` in the specified module. Every time it receives the *UPDATE_LOGIC* message it calls a script function with the declaration `void OnUpdateLogic(float32)` 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). It can also globally reacts on a key pressing and releasing by the `void OnKeyPressed/Released(eKeyCode, uint32)` functions when the layout is enabled.

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

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

Chapter 6

Editor

Namespaces: Editor

Headers: CreateProjectDialog.h, EditorGUI.h, EditorMenu.h, EditorMgr-.h, EntityWindow.h, HierarchyWindow.h, KeyShortcuts.h, LayerWindow.h, PrototypeWindow.h, ResourceWindow.h and editors of properties

Source files: CreateProjectDialog.cpp, EditorGUI.cpp, EditorMenu.cpp, EditorMgr.cpp, EntityWindow.cpp, HierarchyWindow.cpp, KeyShortcuts.cpp, LayerWindow.cpp, PrototypeWindow.cpp, ResourceWindow-.cpp and editors of properties

Classes: CreateProjectDialog, EditorGUI, EditorMenu, EditorMgr, EntityWindow, HierarchyWindow, KeyShortcuts, LayerWindow, PrototypeWindow, ResourceWindow and editors of properties

Libraries used: CEGUI

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

In this section the editor user interface will be described. First the managing of whole projects will be introduced as well as managing of their scenes. Then working with entities will be described including their organization and editing with editor tools. Finally the rest of editor features will be revealed.

6.2.1 Managing projects

After the editor application starts you should either create a new project, or open an old one. For the creation of a new project choose *File→Create Project* from the main menu. To the displyed dialog fill the project name and choose the location where the project will be stored and click to the *Ok* button. Then the new project with the default configuration and the selected name will be created.

For opening of an old project choose *File→Open Project* from the main menu. Then find and select the project directory and click to the *Ok* button. If everything is alright the project will be loaded and the default scene will be opened, otherwise the error message will be displayed.

When the project is finished the game can be deployed for running without the editor on computers without this game engine installed. This can be done by selecting a target platform from the *File→Deploy Project* submenu list. Then a directory where game files will be stored should be selected in the shown dialog. Finally a message box indicates whether the deploy was successful.

It is also possible to close project by the *File→Close Project* choice or quit application by clicking on the *File→Quit* menu item.

6.2.2 Managing scenes

Every project contains one or more scenes which are loaded separately, have own lists of entities (game objects) and layers but a common list of entity prototypes (templates) and resources.

For creating a new scene choose *Scene→New Scene* from the main menu, select the location of it, type its name and click to the *Ok* button. For opening another scene choose *Scene→Open Scene* from the main menu and select the desired scene from the list. There are also options for saving and closing the current scene in the main menu.

6.2.3 Editor windows

Every scene consists of many entities that consist of components and that can be organized in the hierarchy, in the layers, by the position in the scene and by the prototype they are linked to.

The window with the *Hierarchy* caption is used to manage entities in the entity tree (one entity can have one parent entity and none or many child entities). It is possible to change a parent of an entity by drag and drop it to another entity that should be its new parent or after the last entity to get it to the top of the hierarchy. For moving an entity up or down on the same hierarchy level use the options in the popup menu that can be activated by right mouse button click on the required entity.

Every entity with the *Transform* component is presented in the layer manager (window with the *Layers* caption) in its layer. The layers are sorted from the foreground to the background so entities in the first layer are rendered after (are before) entities in the second one. To show the list of the entities in some layer click on the + sign before its name, to hide it click on the - sign at the same place. To show/hide all entities in some layer in the viewports click on the picture with an eye behind its name. To make some layer active (new entities are created to it and the entities in viewport are selected from it) doubleclick to its name that will become bolder. To move the entity to another layer drag and drop it to its name or drag the layer name to the *Layer* property in the entity editor window (see more in the subsection 6.2.4). To move the layer in front of/behind another one drag and drop it to its name. The last two actions can be done also by selecting the corresponding option in the popup menu activated on the layer name as well as renaming or removing it. To add a new layer click on the *New Layer* menu item in the popup menu and fill a new name to the prompt dialog.

Every entity with the *Transform* component and the *Sprite* or the *Model* component is visible in the editor viewport (the bottom window) on its current position. In this viewport it is possible to select entities and manipulate with them by editor tools (see the subsection 6.2.6) or to zoom in or zoom out the editor camera by the mouse wheel for showing various parts of the scene. In the game viewport (the top window) the entities are shown as it will look in the game (see the subsection 6.2.6).

In the window with the *Prototypes* caption entity prototypes of the current project are listed. Further information for managing them is mentioned in the subsection 6.2.5.

6.2.4 Managing entities

There are several ways to create a new entity. Choose *Edit→New Entity* from the main menu for creating an entity with the *Transform* component at the active layer, in the center of the editor viewport and at the top of the entity hierarchy. Choosing the *Add entity* option in the popup menu activated on some entity in the hierarchy window will do the same except this entity will be a parent of the new one. Another possibility is to create an entity from a prototype by popup menu on the prototype window or dragging an item from the prototype list to the hierarchy list or to the editor viewport. It creates an entity with the same components as the chosen prototype has and which is linked to it so it shares some property values with it (see the subsection 6.2.5).

For editing entity properties choose an entity from the hierarchy window, from the layer window or from the editor viewport (only entities in the active layer can be selected) by the left mouse button click. Then the window with *Entity Editor* caption will be filled with general information about the current entity (an ID, a name, a tag and a linked prototype) and with all of its components with their properties. A new component can be added to the entity by choosing the desired one from the *Edit→New Component* list in the main menu or from the similar list in the popup menu activated on the current entity in the hierarchy window. An existing component can be removed from the entity by clicking on the button in the top right corner of it in the entity editor window if it is possible (components can be dependent on each other).

Every writable property of the current entity can be changed in the entity editor window by the property editor displayed near the property name. For every property type there can be a different editor. For example the vector property has two edit boxes for x and y coordinates, while the boolean property can be changed by (un)checking the check box. The properties representing a resource (a texture, a model, a script etc.) cannot be edited directly but the resource file name must be dragged from the resource window (with the *Resources* caption) and dropped on the corresponding edit box in the entity editor. The array properties have several editors for each array item under them that can be added by the *Add* button and removed by its *RM* button. The changes in the array property must be confirmed or reverted by the *Save* or *Rvrt* button.

The current entity can be duplicated or deleted by selecting the *Edit→Duplicate Entity* or *Edit→Delete Entity* from the main menu or the corresponding option of the hierarchy window popup menu. More than one entity from the active layer can be selected by holding the *Shift* key and dragging over them in the editor viewport. This selection can be duplicated or deleted by selecting the *Edit→Duplicate Selected Entities* or *Edit→Delete Selected Entities* from the main menu.

6.2.5 Managing prototypes

An entity can be linked to a prototype with the same components, which means it shares some of prototype's properties. This can be done by dropping the prototype dragged from the prototype window to the corresponding cell in the general information section of the entity editor window where the name and ID of the current prototype are shown and where it is also possible to unlink them by clicking on the *RM* button. A new prototype can be created from the current entity by selecting *Edit→Create Prototype* or by the *Add Prototype* option of the prototype window popup menu where it is also possible to delete another one by the *Delete Prototype* option.

Components and properties of a prototype can be also edited in the entity editor after the selection in the prototype window. The prototype editor looks almost the same as the entity one but before each property there is a check box which indicates whether it should be shared or not. If some property is shared the change of its value in the prototype is propagated to all entities linked to it and it is not possible to edit it in the linked entities (this is indicated by a lock picture behind the property editor).

6.2.6 Editor tools

There is an editor toolbar at the top of the screen with buttons for running action and easier manipulating with entities. The first three buttons serve for resuming, pausing and restarting the game action. When the action is running the entities are affected by physics and scripts so it is possible to see in the game viewport how the scene will look like in the real game. The action can be paused and resumed by the first two buttons while the third button returns the scene to the state before the first resuming of the action (including all changes made to the entities).

The last four buttons serve for moving, rotating and scaling selected entities. Just select the tool (by pressing the corresponding button) and one or more entities in the editor viewport, hold the left mouse button and move with the mouse pointer. The first tool moves the entities, the second rotates them along the z coordinate (orthogonal to the computer screen), the third rotates the entities with the *Model* component along the y coordinate and the last changes the x and y scale.

For opening this file with the user documentation or the file with possible shortcuts or for showing the information about the editor select the corresponding item in the *Help* list in the main menu.

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

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

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

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

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

6.3.3 Used layouts

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

6.4 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

Component – a part of an entity which adds some properties and behavior to it

Prototype – a template entity from which is possible to set properties of all linked entities en bloc

Viewport – a part of editor where a scene is shown

Chapter 7

Input system

Namespaces: InputSystem

Headers: IInputListener.h, InputActions.h, InputMgr.h, KeyCodes.h, OIS-Listener.h

Source files: InputActions.cpp, InputMgr.cpp, OISListener.cpp

Classes: InputMgr, IInputListener, OISListener

Libraries used: OIS

7.1 Purpose of the input system

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

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

7.2 Input manager

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

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

The second approach is supported by multiple methods for querying the device state. The *InputMgr::IsKeyDown* method returns whether any specific key is currently held down while the *InputMgr::IsMouseButtonPressed* method does the same with the currently pressed mouse button. Finally the *InputMgr::GetMouseState* returns a whole bunch of mouse related information such as a cursor and wheel position or pressed buttons.

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

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

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 Purpose of the resource manager

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 usually called *resource*. Since the games work with loads of resources it is necessary to organize them both in-game and on the disk. Also, the data loaded are usually quite large and it's necessary to free them when possible to save memory. All of these tasks are implemented in the resource system.

In the following sections the mapping of a resource to the engine will be described as well as the resource manager. Also a way of loading and saving scenes will be introduced. In the last section there is a small glossary of used terms.

9.2 Resources

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

9.2.1 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
Failed	the resource failed to load

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

Since the resources are passed all around the game system we must somehow prevent any memory leaks from appearing. Doing so is quite easy however – we simply use the shared pointer mechanism. It points to the common abstract resource (the *ResourceSystem::ResourcePtr* class) and to specific resources as well (the *ScriptSystem::ScriptResourcePtr* or the *ResourceSystem::XMLResourcePtr* classes for example). The abstract resource pointer can be automatically converted to any specific resource pointer, but if the type does not match an assertion fault will be raised to prevent a memory corruption. All resource pointers are defined in the `Utils/ResourcePointers.h` file.

9.2.4 Adding custom resource types

There are two steps for adding a new resource type. The first one is creating a class derived from the *ResourceSystem::Resource* class implementing all abstract methods (*Resource::LoadImpl* and *Resource::UnloadImpl*) and providing accessors to the parsed data. The methods *Resource::Open/CloseInputStream* or *Resource::GetRawInputData* can help with this task. Each accessing method must call the *Resource::EnsureLoaded* method to make sure the resource is actually loaded before it is to be used. The second step is adding the corresponding shared pointer to the new class into the `Utils/ResourcePointers.h` file.

To see all currently existing types of resources it is best to locate the *ResourceSystem::Resource* class and see all classes derived from it.

9.3 Resource manager

The resource manager represented by the *ResourceSystem::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. There are methods for adding one file or a whole directory to a resource group or for getting a resource pointer to a resource of a specific name from a specific group.

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

Since gaming systems have limited memory it is possible to limit the memory used by resources. Resources usually take the biggest chunk of memory, so when lowering the memory usage it is best to start here. Hopefully, the resource manager allows 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.

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

9.4 Loading and saving scenes

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

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

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

is created by calling this sequence of orders:

```
ResourceSystem::XMLOutput out("file.xml"); // writes header
```

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

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

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

9.5 Glossary

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

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

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

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

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

Chapter 10

Script system

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 [22].

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 script 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 in milliseconds (0 means infinity) to prevent cycling as the second one. This method executes the script with given arguments and returns whether the execution was successful. If so it is possible to get a return value of function with corresponding methods of context. In the end the context should be released to avoid memory leaks.

There is a possibility to call a simple script string stored in a memory by the method *ScriptMgr::ExecuteString* which accepts this string as the first parameter. The method wraps it to the function without parameters and builds it and calls it as a part of the module specified in the second optional parameter so it is possible to declare local variables and to call functions from the module. This is useful for example for implementing an 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. 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 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: `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 an array	`int32 GetSize() const`
Set a size of a 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 a 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 a no-handle object which means that the 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 the method *EntityMgr::CreateEntity* 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 the Script system registered object chapter.

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 its methods.

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 [22] 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 a 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 [22] 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*, *RegisterEnum-Value*) and typedefs for primitive types (*RegisterTypedef*) that are easy to use. Further information about registering types can be found at the AngelScript documentation [22], 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 files 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 their state. These scripts are written to the `void OnAction()` message handlers which are called when the message `CHECK_ACTION` is received (which should be every game loop) and when the appropriate time in the `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 milliseconds. 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 `this` property can be called from a script to get the current entity handle. Then it adds additional parameters to a function call from the message data according to the message type and executes the context with defined timeout. Finally it releases the context and returns whether the execution was successful.

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 the 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 [23] 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 [24] 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 [23].

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

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

Memory system

Namespaces: Memory

Headers: ClassAllocation.h, FreeList.h, FreeListAllocationPolicies.h, FreeListConstructionPolicies.h, FreeListGrowPolicies.h, FreeListPolicyHelpers.h, GlobalAllocation.h, GlobalAllocation_c.h, StlPoolAllocator.h

Source files: FreeListPolicyHelpers.cpp, GlobalAllocation.cpp

Classes: ClassAllocation, FreeList, LinkedListAllocation, StackAllocation, CompactableChunkAllocation, SharedChunkAllocation, PlacementNewConstruction, NullConstruction, AllocationTracker, SharedFreeList, StlPoolAllocator

Libraries used: none

12.1 Purpose of the memory system

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

Since games are very different from other applications they also have specific memory requirements. However, the default managers in current compilers are targeted to common programs. The game is a real-time performance-intensive application while it also allocates and deallocates a lot of objects on a regular basis. An example of this might be graphical effects spawned and destroyed during the play to keep the action high. In this case it is much more efficient to pool the objects instead of allocating them on the general heap. Another problem arises on game consoles where the memory available to the game is very limited and the engine must hold a tight control on the state of the memory and dynamically clean it if necessary while the action is still running.

In the following sections various forms of memory allocation will be described.

12.2 Global allocation

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

12.3 Free lists

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

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

12.4 Stl pool allocator

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

12.5 Class allocation

As mentioned before it is common in games to rapidly allocate and deallocate objects of the same type – a graphical effect might be an example. At the same time the programmer might not want to store the objects in a structure suitable for pooling and generally care about the memory management. In this case all he has to do is to derive the effect class from *Memory::ClassAllocation* with the correct template paramters. The class controls the way instances of the deriving class are allocated and deallocated by overriding the *new* and *delete* operators. The usage of the class then does not change while the instance may be pooled.

Chapter 13

Platform setup

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

13.1 Purpose of the platform setup

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

In the following section the content of these files will be described.

13.2 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 behavior of libraries for example), but more importantly it aggregates other setup headers together.

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

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

Chapter 14

Utilities and reflection

Namespaces: Utils, Reflection

Headers: Array.h, Callback.h, DataContainer.h, FilesystemUtils.h, GlobalProperties.h, Hash.h, MathConsts.h, MathUtils.h, Properties.h, ResourcePointers.h, Singleton.h, SmartAssert.h, StateMachine.h, StringConverter.h, StringKey.h, Timer.h, Tree.h, XMLConverter.h, AbstractProperty.h, Property.h, PropertyAccess.h, PropertyEnums.h, PropertyFunctionParameters.h, PropertyHolder.h, PropertyList.h, PropertyMap.h, PropertySystem.h, PropertyTypes.h, TypedProperty.h, ValuedProperty.h, RTTI.h, RTTIBaseClass.h, RTTIGlue.h

Source files: FilesystemUtils.cpp, GlobalProperties.cpp, Hash.cpp, MathUtils.cpp, SmartAssert.cpp, StringConverter.cpp, StringKey.cpp, Timer.cpp, XMLConverter.cpp, AbstractProperty.cpp, PropertyEnums.cpp, PropertyFunctionParameters.cpp, PropertyHolder.cpp, PropertyMap.cpp, PropertySystem.cpp, RTTI.cpp

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

Libraries used: none

14.1 Purpose of utilities

During the development of a game usually several helper classes and methods are needed 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 they were aggregated in this namespace.

In the following sections there is a brief description of helper classes and methods as well as a description of the RTTI and property system. In the last section there is a small glossary of used terms.

14.2 Helper classes and methods

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

- Array – a templated representation of an array with the information about its size
- Callback – a generic callback solution that uniformly wraps pointers to functions and/or methods
- DataContainer – a class storing a pointer to a data buffer and its size
- FilesystemUtils – utility functions for working with files and directories
- GlobalProperties – globally accessible variables identified by string identifiers
- Hash – custom hash and reverse hash functions
- MathUtils – helper math functions
- ResourcePointers – a file that gathers smart pointers of all resources together so that it can be included with no overhead
- Singleton – a singleton pattern implementation
- SmartAssert – a smart assert macro implementation
- StateMachine – an implementation of a basic finite state machine
- StringConverter – a set of functions for converting different values to and from a string
- StringKey – a class that serves as a key into maps and other structures where strings are used to index data, but a high speed is necessary
- Timer – a support for time measurement

- Tree – an STL-like container class for n-ary trees
- XMLConverter – a set of functions for reading/writing different values from/to XML

14.3 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. For using it a new class must derive from the *Reflection::RTTIGlue* class which takes two template arguments. The first one is the new class and the second one is its predecessor in the RTTI hierarchy. If there is no parent then the new class should inherit from the *Reflection::RTTIBaseClass* class. The mechanism will automatically call the static *RegisterReflection* method of the new class during the startup if defined. In this method new properties and functions can be registered for the reflection (see section 14.4) or a component dependency can be added by the *AddComponentDependency* method.

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

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

14.5 Glossary

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

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

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

Bibliography

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

[2] Boost – http://www.boost.org

[3] Box2D – http://www.box2d.org

[4] CEGUI – http://www.cegui.org.uk

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

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

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

[8] OpenGL – http://www.opengl.org

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

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

[11] SDL – http://www.libsdl.org

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

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

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

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

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

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

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

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

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

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

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

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

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

List of Figures

List of Tables

Appendix A

Script system registered objects

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

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

EntityPicker

This class allows the game to pick an entity based on provided input data. This class is registered as a value type.

Constructors

- EntityPicker(const Vector2 &in, const int32, const int32) – creates a picker for selecting an entity under the current mouse cursor

Operators and methods

- EntityHandle PickSingleEntity() – runs the picking query, the result is returned directly
- void PickMultipleEntities(EntityHandle[] &out, const Vector2 &in, const float32) – runs the picking query, the result is filled into the given array, the query is defined by a rectangle between the last cursor position and the given cursor position, the rectangle is rotated by the given angle

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

Game

This class represents the current game. 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 ClearDynamicProperties() – clears the dynamic property list
- bool HasDynamicProperty(const string &in) const – returns whether the specified dynamic property exists
- bool DeleteDynamicProperty(const string &in) – deletes the specified dynamic property
- bool LoadFromFile(const string &in) – loads the game from the specified file
- void PauseAction() – pauses the game action until resumed again
- void ResumeAction() – resumes the game action if paused
- bool SaveToFile(const string &in) – saves the game to the specified file
- void Quit() – quits the game
- uint64 GetTime() – returns the current game time in miliseconds
- bool GetFullscreen() – returns whether the game is in the fullscreen mode
- void SetFullscreen(const bool) – sets whether the game is in the fullscreen mode
- T Get_T(string &in) – gets a value of a dynamic property of a specific name and type T from PropertyTypes.h
- void Set_T(string &in, T) – sets a value of a dynamic property of a specific name and type T from PropertyTypes.h

CEGUIString

This class represents string in the CEGUI library. It is registered as a value type and it is implicitly 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

GUIMgr

This class manages the graphic user interface of the game. 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 LoadScheme(const string &in) – loads the project scheme file
- void LoadImageset(const string &in) – loads the project imageset file
- Window@ LoadLayout(const CEGUIString &in, const CEGUIString &in) – loads the project layout file and returns it (the layout filename, widget names in layout file are prefixed with given prefix)

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
- Game& game – the reference to the current game on which it is possible to call methods, this reference cannot be stored to a variable or passed to a function
- GUIMgr& gGUIMgr – the reference to the graphic user interface manager 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

Point

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

Properties

- int32 x – x coordinate
- int32 y – y coordinate

Constructors

- Point() – default constructor (x = y = 0)
- Point(const Point &in) – copy constructor
- Point(int32 x, int32 y) – constructor using coordinates

Operators and methods

- Point operator-() const – negate this point
- void Set(int32, int32) – set this point to some specified coordinates

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 &in, const Vector2 &in)

- float32 Max(const float32, const float32)
- int32 Max(const int32, const int32)
- Vector2 Max(const Vector2 &in, const Vector2 &in)
- 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