Contents

Chapter 1

Introduction

The project Ocerus was designed to be easily extendable on well defined places in code. It allows developers to create even more diverse games. This document serves as a cookbook and shows a sequence of steps that will lead to extension of the Ocerus. There are several ways of extending the Ocerus. Each way will be discussed in separate chapters.

This document only shows how to make things work. If you want to understand how they work, you should see the Design documentation.

Chapter 2

Components

Probably the first thing you would like to extend are the components. They can easily add functionality to game entities. Similar result can be achieved by using scripts but the logic in components is not limited to predefined script functions and runs faster. In the following sections, there will be a sequence of steps that are necessary to do in order to create and use new component.

There will be also an example of simple component that will make an entity to move up and down. It would probably be better to implement this functionality by a script but it will serve well as a simple example.

2.1 Creating Component Class

The first thing that needs to be done is creating new component class. Each component class is defined in separate header file, which are located in *EntitySystem/Components*.

The header file of a component must be defined in the *EntityComponents* namespace and must publicly inherit from the *RTTIGlue< [Component Name], Component>*. So the example component class would look like this:

```
namespace EntityComponents
{
  class ExampleComp : public RTTIGlue<ExampleComp, Component>
  {
  }
}
```

There are few methods in this class that should be overridden. Namely, *Create*, *Destroy*, *HandleMessage* and *RegisterReflection*. *Create* and *Destroy*

methods are called after the creation respectively before the destruction of the component. *HandleMessage* and *RegisterReflection* method will be described in a later sections.

Then, the component type must be registered in the *_ComponentTypes.h* header file (located in *EntitySystem/Components*). New line *COMPONENT_TYPE(ExampleCom* should be added.

Finally the header file with the component class must be included in the *_ComponentHeaders.h* header file.

2.2 Adding Component Properties

The component properties are represented by class members. But just defining a class member is not enough to make it into a regular property. It needs to be registered in the RTI. Registering properties is done in static method *RegisterReflection*. It is called during the initialization process of the *EntitySystem*. In this method, there should be called the *RegisterProperty* function. Usage of that function looks like this:

```
RegisterProperty<Property type>("Property name", &GetterMethod,
  &SetterMethod, Access flag, "Property description");
```

- The Property type is obviously the data type of the property but is must be registered in the *PropertyTypes.h* header file (located in *Utils/Properties*). All regular property types are already registered there. There is also a short description of how to register new property types.

- The property must have defined getter and setter methods. The methods signatures must look like this:

  ```
  [Property type] GetProp(void) const
  void SetZoom([Property type] value)
  ```

  Addresses of those methods must be given as parameters to the *RegisterProperty* function.

- The Access flag defines ways how the property can be accessed from the editor or a script or whether it should be saved/loaded to/from stream. The list of all flag states is in the *PropertyAccess.h* header file (located in *Utils/Properties*).

There can be also functions registered in order to be called from scripts. For that, there is the *RegisterFunction* function. Usage of that function is similar to the *RegisterProperty* and looks like this:

```
RegisterFunction("Function Name", &Function, Access flag,
  "Function description");
```

Back to the example component. Let's say it will have three properties and one function. One property for the amplitude, second for the period of moving, third for the timer and function for reseting the timer. All three properties will be of the float32 data type. Setters and getters must be defined as well as the *RegisterProperty* function.

The example component needs to access and change the position of the entity. The position is defined in the *Transform* component. To ensure that the entity have *Transform* component, the dependency must be added. For that purpose serves the *AddDependency* function. It receives a component type as a parameter (component type is defined as *CT_[Component class name]*). It will also force the dependent component to initialize later than the other component.

So the example component class will look like this:

```
namespace EntityComponents
{
  class ExampleComp : public RTTIGlue<ExampleComp, Component>
  {
  public:
    virtual void Create(void){mAmplitude = mPeriod = mTimer = 0;}
    virtual void Destroy(void){}

    void Reset(void)
    {
       mTimer = 0;
    }

    static void RegisterReflection(void)
    {
      RegisterProperty<float32>("Amplitude", &GetAmplitude,
        &SetAmplitude, PA_FULL_ACCESS, "Amplitude of moving.");

      RegisterProperty<float32>("Period", &GetPeriod,
        &SetPeriod, PA_FULL_ACCESS, "Period of moving.");
```

```
      RegisterProperty<float32>("Timer", &GetTimer,
        &SetTimer,PA_FULL_ACCESS,"Time from the start.");

      RegisterFunction("Reset", &Reset, PA_SCRIPT_WRITE,
        "Resets the timer.");

      AddComponentDependency(CT_Transform);
    }

    float32 GetAmplitude(void) const { return mAmplitude; }
    void SetAmplitude(float32 value) { mAmplitude = value; }

    float32 GetPeriod(void) const { return mPeriod; }
    void SetPeriod(float32 value) { mPeriod = value; }

    float32 GetTimer(void) const { return mTimer; }
    void SetTimer(float32 value) { mTimer = value; }

  private:
    float32 mAplitude;
    float32 mPeriod;
    float32 mTimer;
  }
}
```

2.3 Entity Messages

Entities can receive messages. They are sent by the engine when some event occurs. It can be initialization of the entity, update in physics, pressed key etc. Messages can be also sent from a script.

A message that is sent to an entity is distributed to all its components. It can also carry parameters.

2.3.1 Message Types

Message types are defined in the *EntityMessageTypes.h* header file (located in *EntitySystem/EntityMgr*). New message types can be easily added if needed. Declaration of the message type is done by macros and looks like this:

```
ENTITY_MESSAGE_TYPE([Name], "[Script function]", [Parameters] )
```

- Name is simply name of the message type. Must be unique.
- If an entity has a script component, then when it receives a message, a function in script is called. And that functions name and signature is defined by the second parameter in the *ENTITY_MESSAGE_TYPE* macro.
- The third parameter defines parameters that can be added to the message. When no parameter is needed, then *NO_PARAMS* macro should be used. Otherwise, the *Params* macro should be used. It accepts property types as parameters. So a message with float32 and int32 parameters will have *Params(PT_FLOAT32, PT_INT32)*.

  Note that the parameters must correspond to the script function signature defined in the second parameter of *ENTITY_MESSAGE_TYPE* macro. So it could look like this:

```
ENTITY_MESSAGE_TYPE(HELLO, "void Hello(float32, int32)",
  Params(PT_FLOAT32, PT_INT32) )
```

2.3.2 Handling Messages

The handling of messages is done by the *HandleMessage* method of the component class. It accepts *EntityMessage* as a parameter. The *EntityMessage* contains the message type and message parameters.

The *HandleMessage* method must return result. *RESULT_OK* if the message was accepted, *RESULT_IGNORED* when the message was ignored and *RESULT_ERROR* if an error occurred.

The example component, which makes an entity to move up and down, could have the *HandleMessage* method implemented like this:

```
EntityMessage::eResult EntityComponents::ExampleComp::
  HandleMessage( const EntityMessage& msg )
{
  switch (msg.type)
  {
  case EntityMessage::INIT:
    {
      // Get transform component
      Component* transform = gEntityMgr.
        GetEntityComponentPtr(GetOwner(), CT_Transform);
```

```
      // Get position property from component
      // mStartPosition must be declared as a member variable
      mStartPosition = transform->GetProperty("Position").
      GetValue<Vector2>();

      return EntityMessage::RESULT_OK;
    }
  case EntityMessage::SYNC_PRE_PHYSICS:
    {
      // Get first (and only) message parameter
      float32 delta = *msg.parameters.GetParameter(0).
        GetData<float32>();

      if ((delta <= 0)|| (mPeriod <= 0))
        return EntityMessage::RESULT_OK;

      mTimer += delta;

      // Calculate an offset from the start position
      float32 offset = mAmplitude *
        MathUtils::Sin(mTimer / mPeriod * 2 * MathUtils::PI);

      Component* transform =
      gEntityMgr.GetEntityComponentPtr(GetOwner(), CT_Transform);

      Vector2 pos = mStartPosition;
      pos.y += offset;

      // Set new value to the position property
      transform->GetProperty("Position").SetValue(pos);

      return EntityMessage::RESULT_OK;
    }
    default:
      return EntityMessage::RESULT_IGNORED;
  }
}
```

Chapter 3

Script System

The Script system provides quite powerful tool for programming the game logic without having to modify and compile the source code. Even if there are hundreds of registered functions, classes, operators etc., it could be useful to register some more. The Script system is prepared for this so it is not difficult to register new stuff.

Following sections will show simple example of how to register a new class with methods in the Script system. It will be described only briefly. For more details, please see the AngelScript documentation. Especially chapter *Using AngelScript/Registering the application interface*.

3.1 Registering New Class

All new classes and functions should be registered in the *RegisterAllAdditions* function in the *ScriptRegister.cpp* file (located in *ScriptSystem*).

Registering of new class looks like this:

```
r = engine->RegisterObjectType("NewClass", sizeof(NewClass),
 asOBJ_VALUE | asOBJ_APP_CLASS_C);
OC_SCRIPT_ASSERT();
```

- The method returns an integer typed result of registering. Name of the variable, the result is being assigned to, should be *r*, because it is then checked for error in the *OC_SCRIPT_ASSERT()* macro.
- The first parameter of the method defines the name of the class in a script.
- The second parameter defines the size of the class in bytes.

- The last parameter is a flag defining the behavior. For details see the AngelScript documentation. This particular combination means that the class is handled as object value (not reference) and needs to be constructed and destructed by constructor and destructor.

3.1.1 Class Methods

Class methods should be registered like this:

```
void NewClass::ClassMethod(float32 f) const
{
  // Do something
}

r = engine->RegisterObjectMethod("NewClass",
 "int32 class_method(float32) const",
  asMETHODPR(NewClass, ClassMethod, (float32), int32),
  asCALL_THISCALL);
OC_SCRIPT_ASSERT();
```

- The first parameter is the name of the methods class.
- The second parameter defines name and signature of the method in a script.
- The third parameter defines type and C++ name of the function. When the function is method of some class (as in this example), then *asMETHODPR* macro is used with parameters: name of class, name of method, parameter types and return type.
- The last parameter defines how the function should be called. *asCALL_THISCALL* means that it should be called as a class method. Details in the AngelScript documentation.

3.1.2 Constructors and Destructors

If a constructor or destructor is needed they shall be registered the following way:

```
void Constructor(void *memory)
{
  // Initialize the pre-allocated memory by calling the
```

```
  // object constructor with the placement-new operator
  new(memory) Object();
}

void Destructor(void *memory)
{
  // Uninitialize the memory by calling the object destructor
  ((Object*)memory)->~Object();
}

// Register the behaviours
r = engine->RegisterObjectBehaviour("val",
  asBEHAVE_CONSTRUCT, "void f()", asFUNCTION(Constructor),
  asCALL_CDECL_OBJLAST);
OC_SCRIPT_ASSERT();
r = engine->RegisterObjectBehaviour("val",
  asBEHAVE_DESTRUCT, "void f()", asFUNCTION(Destructor),
  asCALL_CDECL_OBJLAST);
OC_SCRIPT_ASSERT();
```

Chapter 4

Resource Types

4.1 blabla

Chapter 5

Renderer

5.1 blabla

Bibliography

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

List of Figures

List of Tables