Ocerus Extension Guide

http://ocerus.sourceforge.net

February 13, 2011

Contents

1 Introduction **3**

2 Components **4**

2.1 Creating a component class 4
2.2 Adding component properties 5
2.3 Entity messages 7
2.3.1 Message types 7
2.3.2 Handling messages 8
2.4 Conclusion 9

3 Script system **10**

3.1 Registering a new class 10
3.1.1 Class Methods 11
3.1.2 Constructor and Destructor 11
3.2 Global function 12
3.3 Usage of the new class in a script 12
3.4 Conclusion 12

4 Resources **13**

5 Renderer **15**

6 Conclusion **16**

Bibliography **17**

Chapter 1

Introduction

The Ocerus project was designed to be easily extendable at only few points in the engine. It allows developers to create more diverse games and meet the specific needs their project might have. This document serves as a cookbook and shows a sequence of steps that will implement sample extensions of the engine. The following chapters are divided by the points in the engine where it can be extended as stated before.

This document only shows how to make things work. If you want to understand the engine from inside you should read the Design Documentation.

Chapter 2

Components

Every game object is represented by an entity which is a compound of components that provide 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). They are accessible through their unique name. The component can also react to messages such as initialization, drawing, logic update etc. which are send from a component to entities or broadcastet by the engine to all entities.

Ocerus provides several predefined components that implement basic features such as the visual representation, physics, scripts etc. But it is also possible to define new components. They can add new functionality to the game entities. Similar result can be achieved by using scripts but the logic in components is not limited to predefined script functions and it also runs faster. In the following sections, there will be a sequence of steps that are necessary to create and use a new component.

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

2.1 Creating a component class

The first thing that must be done is to create a new component class. Each component class is defined in a separate header file. These files are located in *EntitySystem/Components*.

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

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

There are few methods in this class that should be overridden: *Create*, *Destroy*, *HandleMessage* and *RegisterReflection*. *Create* and *Destroy* methods are called after the creation and before the destruction of the component, respectively. The *HandleMessage* and *RegisterReflection* methods will be described in later sections.

After that, the component type must be registered in the *_ComponentTypes.h* header file (located in *EntitySystem/Components*). To do so, this line should be added:

```
COMPONENT_TYPE(ExampleComp)
```

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 member variables. But defining a class member variable is not enough to turn it into a regular property. It needs to be registered in the RTTI. Registering properties is done in the static method *void RegisterReflection()*. It is automatically called during the initialization process of the *EntitySystem*. In this method the registration is done by calling the *RegisterProperty* function like this:

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

- The Property type is the data type of the property but is must exist 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 the getter and setter methods defined. The methods signatures must look like this:

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

  Addresses of these methods are then passed 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 a stream. The list of all flag states is in the *PropertyAccess.h* header file (located in *Utils/Properties*).

Also component functions can be registered so they can be called from a script. To do so, there is the *RegisterFunction* function. The usage of this function is similar to *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. A property for the amplitude, another one for the period of movement and another one for the timer. The function will reset the timer. All the properties will be of the *float32* data type. Setters and getters must be defined as well for the *RegisterReflection* function.

The example component will access and change the position of the entity. The position is defined in the *Transform* component. To ensure that the entity has the *Transform* component, a component dependency must be declared. To do so there's the *AddDependency* function. It takes a component type as a parameter (the component type is defined as *CT_[Component class name]*). It will also force the dependant 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:
    // Component construction. Sets all properties to 0.
    virtual void Create(void) {mAmplitude = mPeriod = mTimer = 0;}
    // Component destruction. Nothing needs to be destructed.
    virtual void Destroy(void) {}

    // Function for resetting the timer.
    void ResetTimer(void)
    {
       mTimer = 0;
    }

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

      // Register the Period property
      RegisterProperty<float32>("Period", &GetPeriod,
        &SetPeriod, PA_FULL_ACCESS, "Period of moving.");

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

      // Register the ResetTimer function
      RegisterFunction("ResetTimer", &ResetTimer,
        PA_SCRIPT_WRITE, "Resets the timer.");

      // Add dependency on the Transform component
      AddComponentDependency(CT_Transform);
    }

    // Getter and setter for the Amplitude property
    float32 GetAmplitude(void) const { return mAmplitude; }
    void SetAmplitude(float32 value) { mAmplitude = value; }

    // Getter and setter for the Period property
    float32 GetPeriod(void) const { return mPeriod; }
    void SetPeriod(float32 value) { mPeriod = value; }

    // Getter and setter for the Timer property
    float32 GetTimer(void) const { return mTimer; }
    void SetTimer(float32 value) { mTimer = value; }

  private:
    // Definition of member variables that are used as properties.
    float32 mAplitude;
    float32 mPeriod;
    float32 mTimer;
  }
}
```

2.3 Entity messages

Entities can receive messages. They are sent by the engine when an event occurs. Or they can be send from an entity to another entity to inform it of a change. It can be an initialization of the entity, an update in physics, a pressed key etc. Messages can also be sent from a script.

A message that is sent to an entity is distributed to all of its components. Some of them may 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. The declaration of the message type is done by macros and looks like this:

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

- Name is simply the 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 function 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 the *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. *EntityMessage* contains the message type and the message parameters.

The *HandleMessage* method must return a 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 move up and down can have the *HandleMessage* method implemented like this:

```
EntityMessage::eResult EntityComponents::ExampleComp::
  HandleMessage( const EntityMessage& msg )
{
  switch (msg.type)
  {
  // Called when the component is initialized
  case EntityMessage::INIT:
    {
      // Get the transform component
      Component* transform = gEntityMgr.
        GetEntityComponentPtr(GetOwner(), CT_Transform);
      // Get the position property
      // mStartPosition must be declared as a member variable
      mStartPosition = transform->GetProperty("Position").GetValue<Vector2>();

      return EntityMessage::RESULT_OK;
    }
  // Called when the logic is updated (once per frame)
  case EntityMessage::UPDATE_LOGIC:
    {
      // Get the first message parameter
      // It is the time from the last logic update
      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);

      // Get the transform component
      Component* transform = gEntityMgr.
        GetEntityComponentPtr(GetOwner(), CT_Transform);

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

      // Set the new value to the position property
      transform->GetProperty("Position").SetValue(pos);
      return EntityMessage::RESULT_OK;
    }
    default:
      return EntityMessage::RESULT_IGNORED;
  }
}
```

2.4 Conclusion

After finishing the steps above the new component is ready to be used. It will appear in the editor and component properties and functions will be accessible by a script.

Note that most features are easier to implement with scripts. The main advantage (and also the main disadvantage) of creating a new component is that it is defined in the source code. The new component can directly access the engine but the system has to be recompiled with every change to the component.

The scripts, on the other hand, can be changed real-time, when the editor is running. It makes developing scripts much easier. The disadvantage of scripts is that it has only a limited list of functions. But that list can be extended, see chapter *Script system*.

Chapter 3

Script system

The Script system provides quite a 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 features.

The following section will show a simple example of how to register a new class. Some details will be skipped for the sake of readability. You can find the details in the AngelScript documentation [1]. Especially chapter *Using AngelScript/Registering the application interface.*

3.1 Registering a new class

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

To register a new class (named *NewClass*) we can do something like this:

```
class NewClass
{
  ...
};

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

- The method returns an integer result. The name of the variable where the result is assigned should be r because it is then checked for an 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 [1]. This particular combination means that the class is handled as object value (not reference) and is created and destroyed by a constructor and a destructor.

3.1.1 Class Methods

Class methods should be registered like this:

```
class NewClass
{
  int32 ClassMethod(float32 f)
  {
    // Do something
  }
  ...
};

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

- The first parameter is the name of the class the method belongs to.
- The second parameter defines the name and the signature of the method in a script.
- The third parameter defines the C++ signature of the function. When the function is a method of a class (as in this example), then *asMETHODPR* macro is used with the following parameters: name of the class, name of the method, the parameter types and the return type.
- The last parameter defines how the function should be called. *asCALL_THISCALL* means that it should be called as a class method. See details in the AngelScript documentation[1].

3.1.2 Constructor and Destructor

If a constructor or a destructor is needed, it is to be registered in the following way:

```
void Constructor(NewClass *memory)
{
  // Initialize the pre-allocated memory by calling the
  // object constructor with the placement-new operator
  new(memory) NewClass();
}

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

// Register the constructor
r = engine->RegisterObjectBehaviour("NewClass", asBEHAVE_CONSTRUCT, "void f()",
  asFUNCTION(Constructor), asCALL_CDECL_OBJLAST); OC_SCRIPT_ASSERT();

// Register the destructor
r = engine->RegisterObjectBehaviour("NewClass", asBEHAVE_DESTRUCT, "void f()",
  asFUNCTION(Destructor), asCALL_CDECL_OBJLAST); OC_SCRIPT_ASSERT();
```

3.2 Global function

Similarly, a global function can be added.

```
r = engine->RegisterGlobalFunction("float32 NewGlobFnc(int32)",
  asFUNCTIONPR(GlobFnc, (int32), float32), asCALL_CDECL); OC_SCRIPT_ASSERT();
```

The parameters are the same as in the *RegisterObjectMethod* method, only the first one is missing.

It is also possible to register enumerations (*RegisterEnum, RegisterEnumValue*) and typedefs for primitive types (*RegisterTypedef*). The usage is quite easy and can be found in the AngelScript documentation[1].

3.3 Usage of the new class in a script

After the new class was registered, it can be used in a script just like in the C++ code.

```
  NewClass c;
  c.class_method(0);
```

3.4 Conclusion

Scripts should be used for most of the game logic including AI, GUI control, input etc. Registering new functions to the Script system can add new functionality but it could also help to boost the performance. The logic in scripts runs slower than the native code. For example the pathfinding could be too slow when implemented in a script. A faster solution would be to write functions that compute the pathfinding in the native code and then register the functions that return the result to the script.

Chapter 4

Resources

All resources in Ocerus are managed by the Resource manager. Every resource has its type. There are some basic resource types defined in the Resource manager but sometimes it would be nice to define a new one.

New resource types are defined as a class derived from the *Resource* class. It is necessary to add a new resource type to the *eResourceType* enum and a resource type name to the *ResourceTypeNames* string array. Both are defined in the *ResourceType.h* header.

The new resource class has to override some methods:

- Constructor and destructor.
- Static *CreateMe()* method that returns a special shared pointer (*ResourceSystem ::ResourcePtr*) that points to the new resource class instance. This pointer allows type-checked casting to specific resource pointers. So it should look like this:

  ```
  ResourceSystem::ResourcePtr NewResource::CreateMe()
  {
    return ResourceSystem::ResourcePtr(new NewResource());
  }
  ```

- Static GetResourceType() method that returns *eResourceType.* That is the type you added to the enum.

  ```
  static ResourceSystem::eResourceType GetResourceType()
  {
    return ResourceSystem::RESTYPE_NEWRESOURCE;
  }
  ```

- Protected abstract *LoadImpl.* This method is called when the resource is being loaded. It should parse the data from the file stream into something useful. It should use the *DataContainer* class and the *GetRawInputData* method to get the data from the source.

  So, the *LoadImpl* method will look like this:

```
size_t NewResource::LoadImpl()
{
  DataContainer dc;
  size_t dataSize = 0;

  // loads the data from the input to the data container
  // returns true if successful
  if (GetRawInputData(dc))
  {
    dataSize = dc.GetSize();

    // get a pointer to the data byte stream
    uint8* data = dc.GetData();

    // parse the data
    // save the data into its own data structure

    dc.Release();
  }
  return dataSize;
}
```

The *LoadImpl* method must return the size of the resource in memory in bytes. If 0 is returned, it means that the resource failed to load.

- Protected abstract method *UnloadImpl*. This method is called when the resource is being unloaded. It should release all the data the resource was using.
- The resource class can have additional methods for accessing the resource type specific structures. Those method should always call *EnsureLoaded()* as the first thing to do. It makes sure the resource is loaded. Otherwise the data could be invalid.

Defining new resource types is necessary when adding functionality that requires data from external files. For example, a new component that adds sounds to an entity would need a new resource type for the sound files. The component should have a *ResourcePtr* typed property, through which it could access the resource data, and it should define and export functions that will allow to play the sound from a script.

Chapter 5

Renderer

Ocerus uses the OpenGL renderer by default. But it is possible to implement a different renderer that uses another graphical library.

The only part of the code which uses OpenGL directly is the *OglRenderer* class. It derives from the *GfxRenderer* class and implements its abstract methods. There are about twenty methods that must be implemented by the new renderer. They are described in the Doxygen documentation.

After the new renderer class is implemented, the initialization of the renderer must be changed. In the *Application.cpp* file (located in *Core*), the line:

```
GfxSystem::GfxRenderer::CreateSingleton<GfxSystem::OglRenderer>();
```

should be replaced by:

```
GfxSystem::GfxRenderer::CreateSingleton<GfxSystem::NewRenderer>();
```

Chapter 6

Conclusion

TODO.

Bibliography

[1] AngelScript documentation –
http://www.angelcode.com/angelscript/sdk/docs/manual/index.html