SENTINEL

MANUAL

Table of Contents

[Naming Conventions 3](#_Toc373224873)

[Model Exporter 3](#_Toc373224874)

[Controls 4](#_Toc373224875)

[Level Editor 4](#_Toc373224876)

[Sentinel Game 4](#_Toc373224877)

[Create Custom Program 5](#_Toc373224878)

[DLL Import / Export 5](#_Toc373224879)

[Serializable 5](#_Toc373224880)

[Widgets 6](#_Toc373224881)

[Renderer 6](#_Toc373224882)

[Shaders 6](#_Toc373224883)

# Naming Conventions

All member class variables start with m,

e.g. ColorRGBA\* mColor;

All macros and static variables are in all capitals,

e.g. static const double DESIRED\_FRAME\_RATE;

All member functions start with capital letters,

e.g. mGameWorld->Startup();

All local variables start with lowercase letters,

e.g. MeshBuilder meshBuilder;

# Model Exporter

The custom *3ds Max* exporter solution resides within the **Sentinel\_Exporter** folder. The resulting **Sentinel\_Exporter.dle** outputs to the local *3ds Max 2012* folder, e.g. C:\Program Files\Autodesk\3ds Max 2012\plugins, through the “**ADSK\_3DSMAX\_x64\_2012**” environment variable. An additional environment variable, “**MAX2012SDK**”, references the SDK. Generally, the file folder for *3ds Max* resides within a protected folder, therefore, to compile the program, *Visual Studio* should be opened in *Administrator* mode.

The exporter only exports the model within the scene, i.e. no camera, lights, etc. The textures automatically copy into the same folder as the exported model. Save the file as a native file format to *3ds Max 2012* in order to import the model.

# Controls

## Level Editor

The world viewing area can rotate its view by holding the right mouse button.

Strafe by holding the middle mouse button.

Move forward and backward by scrolling the middle mouse wheel.

To create a hierarchy with the **Objects**, drag and drop them onto each other.

All values within the **Inspector** modify the objects immediately, i.e. no need to reload the scene or object to see the changes.

## Sentinel Game

|  |  |
| --- | --- |
| **Keys** | **Function** |
| WASD  Space / C  ESC | Move  Up / Down  Exit |

# Create Custom Program

Open “**Sentinel\_Test.sln**”

Use “**Sentinel\_Test.cpp**” as a reference.

Each header file contains instructions on its usage.

**Sentinel\_Game** loads “**Default.MAP**” to test functionality.

## DLL Import / Export

The **Sentinel\_DLL** project properties contain a preprocessor variable named **BUILD\_DLL**, which indicates that the preprocessor definition SENTINEL\_DLL contains the function required to either export or import functions or classes.

Classes:

class SENTINEL\_DLL SomeClass

{

…

};

Global functions:

UINT SENTINEL\_DLL HashString( const char\* str );

## Serializable

Serializable classes input / output to files through the Archive class. Serialize only derived classes because there is no way to determine what that class may be while loading the required data, e.g. GameComponent or Widget. Those particular classes have many variations of derivations, and loading, i.e. cloning, them into the correct derivation requires identification in the Archive file.

The DECLARE\_SERIAL macro creates a static SerialRegister that registers the class and enables the SerialFactory to generate a Clone of the class. Save and Load functions are generated.

Examples for creating simple custom Serializable classes are in **ParticleEffect.h** and **ParticleEffect.cpp**. Pay close attention to how each class requires zero constructor arguments. This allows the static mSerialRegister to create the Clone. The base class Clone’s Load function uses the derived Load function.

In addition to classes being Serializable, function pointers are also Serializable. SerialFunctionFactory registers function pointers called from any object. This is useful in that many of the Widget classes contain function pointers to execute advanced event handling. Due to the method of function pointer storage, all functions in any class can be used so long as it has the format void(), e.g. void Up();

## Widgets

Widget classes form the basis for all GUI functionality within the application. Attach a WidgetComponent to a GameObject to create a root Widget (mRoot) object. Each Widget contains base variables to adjust its size, rotation, scale, color, visibility, and hierarchy (children).

It is recommended that an additional GameWorld be created with a WidgetComponent attached to a GameObject to handle the GUI as most GUI should be rendered on top of rendered world. Doing so eliminates the need for DrawableComponent sorting, thus speeding up rendering.

## Renderer

The Renderer uses either **OpenGL** or **DirectX 11**. Create only one Renderer, and do not use it in multiple threads.

### Shaders

Since the Renderer has two distinct SDKs that it can work with depending on how the program is set up, a custom shader format is required for the translations. Both **GLSL** and **HLSL** should be included in a XSH file depending on the type of Renderer desired. Create both OpenGL and DirectX shaders for full dual-renderer support.

Creating a custom shader requires specific #define macros to compile correctly.