SENTINEL

MANUAL

Table of Contents

[Naming Conventions 3](#_Toc380239652)

[Model Exporter 3](#_Toc380239653)

[Controls 4](#_Toc380239654)

[Level Editor 4](#_Toc380239655)

[Sentinel Game 4](#_Toc380239656)

[Create Custom Program 5](#_Toc380239657)

[DLL Import / Export 5](#_Toc380239658)

[Serializable 5](#_Toc380239659)

[Widgets 6](#_Toc380239660)

[Renderer 6](#_Toc380239661)

[Textures 6](#_Toc380239662)

[Shaders 6](#_Toc380239663)

[Table 1: Shader Macro Definitions 8](#_Toc380076736)

[Table 2: Shader Vertex Attribute Definitions 9](#_Toc380076737)

[Table 3: Shader Uniform Definitions 11](#_Toc380076738)

# 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 class 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.

Click the left mouse button to select objects.

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 export functions or classes. Otherwise, the program is set to import.

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. See **Serializable.h** for the entire list of macro definitions available.

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();

All serialized classes and functions use the HashString function found within **Util.h** to generate a hash code. The **djb2** algorithm used can cause collisions with minimal chance, but provides an efficient method of handling hash code generation for strings.

## Widgets

Widget classes form the basis for all GUI functionality within the application. Attach a WidgetComponent to a GameObject to create a WidgetWorld object. The MVC (Model-View-Controller) paradigm used requires each WidgetObject to contain a ModelWidget, ViewWidget, and ControllerWidget or some derivation.

## Renderer

The Renderer uses either **OpenGL** or **DirectX 11**. Create only one Renderer, and do not use it in multiple threads. There has been no indication that multiple rendering threads will benefit the application, but rather make it much more unmanageable. Additionally, the Renderer does not fully support multiple rendering threads.

### Textures

The Renderer imports all image formats supported by **stbi-1.33** (JPEG, BMP, PSD, GIF, HDR, PIC) and DDS files for **DirectX 11** only. All textures loaded from files automatically become 32-bit color, i.e. IMAGE\_FORMAT\_RGBA.

### 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. Single implementations, e.g. **GLSL** only, are accepted.

All shaders placed within the **Sentinel\_Editor\Assets\Shaders** folder are automatically loaded and compiled. Compilation errors are output to the compiler in the Output window, and an error message ends the program. Before compilation, the Renderer::CreateShaderFromFile function inserts an appropriate #define macro specifying the type, i.e. #define VERSION\_DX or #define VERSION\_GL, which, as a consequence, outputs the incorrect line numbers on errors or warnings.

Creating a custom shader requires specific #define and #ifdef macros along with attribute and uniform naming conventions to compile correctly. Follow all #ifdef statements with #endif. The shader attributes created within an **HLSL** designated shader requires no specific name, but it must specify the listed semantic. Material, which contains the Shader within each Mesh, updates all shader uniforms automatically before rendering.

Table 1: Shader Macro Definitions

|  |  |
| --- | --- |
| **Macro** | **Description** |
| #ifdef VERSION\_GL | Required before declaring a shader program section to be **GLSL** compatible. |
| #ifdef VERTEX\_SHADER | Designates the following code segment to be part of the Vertex Shader (**GLSL** only). |
| #ifdef GEOMETRY\_SHADER | Designates the following code segment to be part of the Geometry Shader (**GLSL** only). |
| #ifdef FRAGMENT\_SHADER | Designates the following code segment to be part of the Fragment Shader (**GLSL** only). |
| #ifdef VERSION\_DX | Required before declaring a shader program section to be **HLSL** compatible. |
| VS\_Main | Required for **HLSL** to compile the Vertex shader. |
| GS\_Main | Required for **HLSL** to compile the Geometry shader. |
| PS\_Main | Required for **HLSL** to compile the Pixel shader. |

Table 2: Shader Vertex Attribute Definitions

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| **GLSL**  vec3 Position;  **HLSL**  float3 Position  : POSITION; | Vertex position.  Note that the position is normally multiplied by the \_WVP uniform:  vec4 pos = \_WVP \* vec4( Position, 1 ); |
| **GLSL**  vec2 TexCoord#;  **HLSL**  float2 TexCoord#  : TEXCOORD#; | Texture coordinates where (0, 0) is the upper left, and (1, 1) is the bottom right.  TexCoord0 = TEXTURE\_DIFFUSE (0)  TexCoord1 = TEXTURE\_NORMAL (1)  TexCoord2 = TEXTURE\_PARALLAX (2)  Although each texture location has a formal index, they are not bound to those designations. |
| **GLSL**  vec4 QuadCoord#;  **HLSL**  float4 QuadCoord#  : QUADCOORD#; | Texture coordinates where (upper left, bottom right), e.g. (0, 0, 1, 1) represents the entire texture.  See TexCoord# for texture number designations.  Useful for Sprite rendering where a single point generates a Quad. |
| **GLSL**  vec3 Normal;  **HLSL**  float3 Normal  : NORMAL; | Vertex normal.  Note that the normal should be multiplied by the \_WORLD matrix uniform:  vec3 N = \_World \* vec4( Normal, 0 ).xyz; |
| **GLSL**  vec4 Color;  **HLSL**  float4 Color  : COLOR; | Vertex color.  Note that the color is input into the vertex as an unsigned int. |

|  |  |
| --- | --- |
| **GLSL**  vec4 Tangent;  **HLSL**  float4 Tangent  : TANGENT; | Vertex tangent for normal matrix contruction:  vec3 normal = (\_World \* vec4( Normal.xyz, 0 )).xyz;  vec3 tangent = (\_World \* vec4( Tangent.xyz, 0 )).xyz;  vec3 binormal = cross( normal, tangent.xyz ) \* Tangent.w;  mat3 matTBN = mat3( tangent.x, binormal.x, normal.x,  tangent.y, binormal.y, normal.y,  tangent.z, binormal.z, normal.z ); |
| **GLSL**  int BoneCount;  **HLSL**  int BoneCount  : BONE\_COUNT; | Number of bones affecting the vertex, max of 4. |
| **GLSL**  int4 BoneIndex;  **HLSL**  int4 BoneIndex  : BONE\_INDEX; | Index of the matrices that affect the vertex. |
| **GLSL**  vec4 Weight;  **HLSL**  float4 Weight  : WEIGHT; | Percentage of matrix at BoneIndex that affects the vertex, 0.0 – 1.0 |
| **GLSL**  mat4 Matrix;  **HLSL**  matrix Matrix  : MATRIX; | Matrix within a vertex.  For example, used for Sprite generation of the Quad. |

Table 3: Shader Uniform Definitions

|  |  |
| --- | --- |
| **Uniform** | **Description** |
| **GLSL**  mat4 \_WVP;  **HLSL**  matrix \_WVP; | World-View-Projection matrix.  Updated by Mesh through TransformComponent and CameraComponent. |
| **GLSL**  mat4 \_World;  **HLSL**  matrix \_World; | World matrix.  Updated by Mesh through TransformComponent. |
| **GLSL**  mat4 \_InvWorld;  **HLSL**  matrix \_InvWorld; | Inverse of World matrix.  Updated by Mesh through TransformComponent. |
| **GLSL**  mat4 \_View;  **HLSL**  matrix \_View; | View matrix.  Updated by Mesh through current CameraComponent set within GameWorld. |
| **GLSL**  mat4 \_InvView;  **HLSL**  matrix \_InvView; | Inverse of view matrix.  Updated by Mesh through current CameraComponent set within GameWorld. |
| **GLSL**  mat4 \_Proj;  **HLSL**  matrix \_Proj; | Projection matrix.  Updated by Mesh through current CameraComponent set within GameWorld. |
| **GLSL**  mat4 \_InvProj;  **HLSL**  matrix \_InvProj; | Inverse of projection matrix.  Updated by Mesh through current CameraComponent set within GameWorld. |

|  |  |
| --- | --- |
| **GLSL**  vec4 \_Ambient;  **HLSL**  float4 \_Ambient; | Ambient material color.  Updated by Mesh through Material. |
| **GLSL**  vec4 \_Diffuse;  **HLSL**  float4 \_Diffuse; | Diffuse material color.  Updated by Mesh through Material. |
| **GLSL**  vec4 \_Specular;  **HLSL**  float4 \_Specular; | Specular material color.  Updated by Mesh through Material. |
| **GLSL**  float \_SpecComp;  **HLSL**  float \_SpecComp; | Specular component of specular material color.  Updated by Mesh through Material. |
| **GLSL**  vec3 \_LightPos;  **HLSL**  float3 \_LightPos; | Light position.  Updated by Mesh through PointLightComponent within GameWorld. |
| **GLSL**  vec3 \_LightDir;  **HLSL**  float3 \_LightDir; | Light direction.  Updated by Mesh through DirectionalLightComponent within GameWorld. |
| **GLSL**  vec3 \_LightColor;  **HLSL**  float3 \_LightColor; | Light color.  Updated by Mesh through LightComponent within GameWorld. |
| **GLSL**  vec4 \_LightAttn;  **HLSL**  float4 \_LightAttn; | Light attenuation. Last value (w) represents the radius.  Updated by Mesh through LightComponent within GameWorld. |
| **GLSL**  mat4 \_LightCubeMatrix[6];  **HLSL**  matrix \_LightCubeMatrix[6]; | Camera matrices representing each of the 6 directions for a point light source.  Updated by Mesh through PointLightComponent. |
| **GLSL**  mat4 \_LightMatrix;  **HLSL**  matrix \_LightMatrix; | Camera matrix representing the direction of the directional light source.  Updated by Mesh through DirectionalLightComponent. |
| **GLSL**  vec3 \_CameraPos;  **HLSL**  float3 \_CameraPos; | Camera position.  Updated by Mesh through current CameraComponent set within GameWorld. |
| **GLSL**  mat4 \_Bones[100];  **HLSL**  matrix \_Bones[100]; | Bone matrices for the current Model. Maximum amount is set at 100.  Updated within Model. |
| **GLSL**  float \_DeltaTime;  **HLSL**  float \_DeltaTime; | Delta time between frames.  Updated within Mesh. |