Alpha Engine Tutorial

What is Alpha Engine?

Alpha Engine contains a library of code that facilitates creating an application on a desktop environment. It is a thin layer that wraps around Windows OS and OpenGL. As such, it provides functions for input, graphics rendering and window management. Note that it **does not** wrap around audio.

Project Creation

Open Visual Studio 2022 and create a new C++ Empty Project.

- [Create a new Project] > [Empty Project] > [Create]

Next, we will add the Alpha Engine library to a location within the project folder. This is so that our project can easily locate Alpha Engine's header and library files:

- Locate the project's Solution File (.sln) using Windows Explorer.
- Create a folder named **Extern** at that location.
- Copy the **AlphaEngine_V3.08** folder in the given **AlphaEngine_V3.08.zip** file into the **Extern** folder.

Then, we add a folder for our Assets.

- Locate the project's Solution File (.sln) using Windows Explorer.
- Create a folder named **Assets** at that location.
- Copy the contents in the given **Assets.zip** file into the **Assets** folder.

In this project, we will not be building in 32-bit systems, so to avoid confusing, we need to remove 32-bit system-related configurations from the project:

- Under [Build] > [Configuration Manager]
    - [Active Solution Platform] > [Edit…] > Click on [x86] > [Remove]
    - [Platform] > [Edit…] > Click on [Win32] > [Remove]

Project Configuration

Next, we configure the project properties under the [View] > [Properties Window] window. Open that window for the next steps.

Unless stated otherwise, make sure to set [Configurations] to "All Configurations".

Add the additional directories the compiler needs to look for when compiling and linking:

- Under [Configuration Properties] > [VC++ Directories]
    - Append "$(SolutionDir)Extern\AlphaEngine_V3.08\include" to [General] > [Include Directories]
    - Append "$(SolutionDir)Extern\AlphaEngine_V3.08\lib" to [General] > [Library Directories]

Configure the linker to link to the appropriate Alpha Engine depending on whether we are on Debug or Release configurations:

- Under [Configuration Properties] > [Linker] > [Input] > [Additional Dependencies]
    - With [Configuration] set at **Debug**, append "Alpha_EngineD.lib"
    - With [Configuration] set at **Release**, append "Alpha_Engine.lib"

Configure the character set the project is using.

- Under [Configuration Properties] > [Advanced] > [Character Set]
    - Set to **Use Multibyte Set**

Configure the subsystem the project is using.

- Under [Configuration Properties] > [Linker] > [System] > [Subsystem]
    - Set to **Windows (/SUBSYSTEM:WINDOWS)**

Compiler and Debugger Configuration

The next steps will configure the output of the compiler, set the working directory of the debugger, and write our first lines of code in order to compile and run an executable successfully.

Unless stated otherwise, make sure to set [Configurations] to "All Configurations".

Set the output directory of the compiler where the executable will be created. We will set this to a folder named **bin** at the directory the Solution file is in:

- Under [Configuration Properties] > [General] > [Output Directory]
  - Set to "$(SolutionDir)\bin\$(Configuration)-$(Platform)\"

Set the intermediate directory of the compiler. This is where all the 'rubbish' files that the compiler generates will go to. We will set this to a folder named **.tmp** at the directory the Solution file is in:

- Under [Configuration Properties] > [General] > [Intermediate Directory]
  - Set to "$(SolutionDir)\.tmp\$(Configuration)-$(Platform)\"

Set the working directory of the debugger to be in the same directory as the executables output by the compiler

- Under [Configuration Properties] > [Debugging]
  - Set to "$(SolutionDir)\bin\$(Configuration)-$(Platform)\"

Setup a post-build event, to copy the appropriate .**dll** and **assets** to where the executable is.

- Under [Configuration Properties] -> [Build Events] > [Post-Build Event] > [Command Line] > [Edit…]
  - With [Configuration] set at **Debug,** append the following lines:
    - xcopy "$(SolutionDir)Assets*" "$(OutDir)Assets\" /s /r /y /q
    - xcopy "$(SolutionDir)Extern\AlphaEngine_V3.08\lib\freetype.dll" "$(OutDir)" /s /r /y /q
    - xcopy "$(SolutionDir)Extern\AlphaEngine_V3.08\lib\Alpha_EngineD.dll" "$(OutDir)" /s /r /y /q
  - With [Configuration] set at **Release,** append the following lines:
    - xcopy "$(SolutionDir)Assets*" "$(OutDir)Assets\" /s /r /y /q
    - xcopy "$(SolutionDir)Extern\AlphaEngine_V3.08\lib\freetype.dll" "$(OutDir)" /s /r /y /q
    - xcopy "$(SolutionDir)Extern\AlphaEngine_V3.08\lib\Alpha_Engine.dll" "$(OutDir)" /s /r /y /q

Create a fresh .cpp with the entry point named **Main.cpp**

- [Project] > [Add New Item..] > [C++ File]

- Copy the code from the given **barebones.cpp** into the **Main.cpp** file that you just created.
- Build and run the project
- You should see a blank window with the title “My New Demo” pop up with a console window at the back. The problem should close upon pressing the escape key.

If you got here, congratulations! You have set up Alpha Engine!

Understanding how to render something on the screen

In Alpha Engine, rendering something on the screen goes through similar steps as rendering something using a graphics API like OpenGL, DirectX, Vulkan, etc; Alpha Engine simplifies the steps so that you don't have to manage the underlying details.

The steps are as follows:

1. Create a Mesh (sometimes called a Model) on initialization. A Mesh is just a collection of triangles, which are a collection of 3 vertices. You can of a Mesh as a blueprint, which we can create clones of to draw on the screen.
2. In the game loop:
    a. Inform the engine that about how you want to draw the Mesh (e.g. what position, what transform, what texture, etc)
    b. Draw the mesh

This means that with 1 mesh, you can draw several copies of it on the screen!

Typically, in a 2D application, you can get away with a single square mesh: you just need to tell the engine what texture to use before drawing the mesh.

Creating a Mesh

The code below will create a square mesh of 1 pixel width and height in Alpha Engine. Do this **BEFORE** the game loop (not during!).

```
// Pointer to Mesh
AEGfxVertexList * pMesh = 0;

// Informing the library that we're about to start adding triangles
AEGfxMeshStart();

// This shape has 2 triangles that makes up a square
AEGfxTriAdd(
      -0.5f, -0.5f, 0x00FF00FF, 0.0f, 0.0f,
       0.5f, -0.5f, 0x00FFFF00, 0.0f, 0.0f,
      -0.5f,  0.5f, 0x0000FFFF, 0.0f, 0.0f);

AEGfxTriAdd(
       0.5f, -0.5f, 0x00FFFFFF, 0.0f, 0.0f,
       0.5f,  0.5f, 0x00FFFFFF, 0.0f, 0.0f,
      -0.5f,  0.5f, 0x00FFFFFF, 0.0f, 0.0f);

// Saving the mesh (list of triangles) in pMesh
pMesh = AEGfxMeshEnd();
```

After you are done with the Mesh, remember to free it **AFTER** the game loop **BEFORE** the application exits:

```
AEGfxMeshFree(pMesh);
```

Loading a Texture

The code below will load a Texture located in our Assets folder and store it under pTex:

```
AEGfxTexture* pTex = AEGfxTextureLoad("Assets/PlanetTexture.png");
```

After you are done with the Texture, remember to free it **AFTER** the game loop **BEFORE** the application exits:

```
AEGfxTextureUnload(pTex);
```

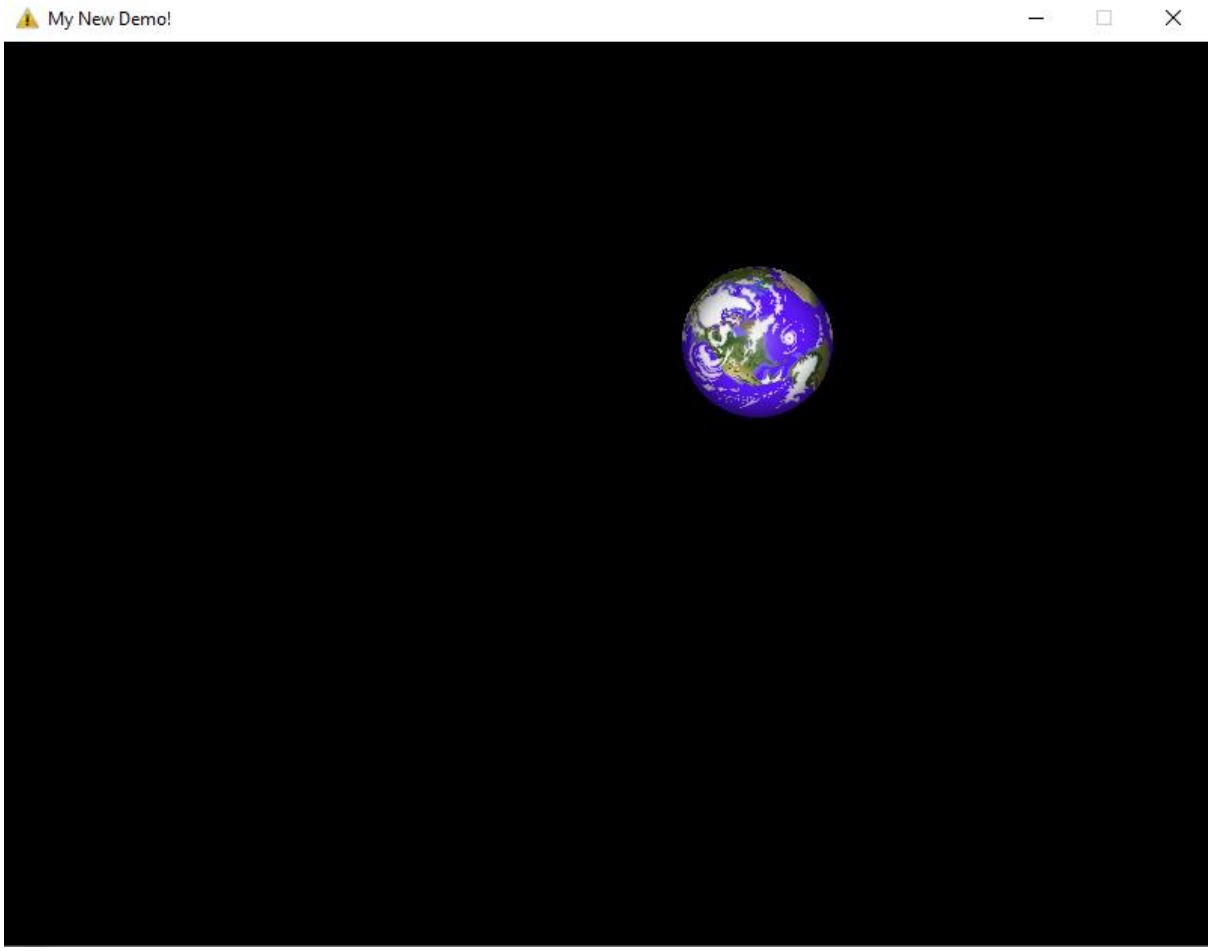
Rendering

Now that we created our mesh, we can then use the mesh to render during the game loop. How you want to do this depends on you but below is an example of drawing 2 'sprites' using the mesh we created.

Below the comments that say: "Your own rendering logic goes here", between `AESysFrameStart()` and `AESysFrameEnd()` write the following code:

```
// Set the background to black.
AEGfxSetBackgroundColor(0.0f, 0.0f, 0.0f);

// Tell the engine to get ready to draw something with texture.
AEGfxSetRenderMode(AE_GFX_RM_TEXTURE);

// Set the tint to white, so that the sprite can
// display the full range of colors (default is black).
AEGfxSetTintColor(1.0f, 1.0f, 1.0f, 1.0f);

// Set blend mode to AE_GFX_BM_BLEND
// This will allow transparency.
AEGfxSetBlendMode(AE_GFX_BM_BLEND);
AEGfxSetTransparency(1.0f);

// Set the texture to pTex
AEGfxTextureSet(pTex, 0, 0);

// Create a scale matrix that scales by 100 x and y
AEMtx33 scale = { 0 };
AEMtx33Scale(&scale, 100.f, 100.f);

// Create a rotation matrix that rotates by 45 degrees
AEMtx33 rotate = { 0 };
AEMtx33Rot(&rotate, PI/4);

// Create a translation matrix that translates by
// 100 in the x-axis and 100 in the y-axis
AEMtx33 translate = { 0 };
AEMtx33Trans(&translate, 100.f, 100.f);

// Concat the matrices (TRS)
AEMtx33 transform = { 0 };
AEMtx33Concat(&transform, &rotate, &scale);
AEMtx33Concat(&transform, &translate, &transform);

// Choose the transform to use
AEGfxSetTransform(transform.m);

// Actually drawing the mesh
AEGfxMeshDraw(pMesh, AE_GFX_MDM_TRIANGLES);
```

After compiling, you should see PlanetTexture.png drawn on the screen, scales to 100x100 pixels, rotated by 90 degrees and translated by 100 on the x-axis and 100 on the y-axis:



Go through line by line and make sure you understand what each statement does. You are highly encouraged to modify and play around each line.

More functions and details can be found in AEGraphics.h.

As an exercise, try to make the planet rotate clockwise around the center of the screen and rotate anti-clockwise around itself!

Input

Alpha Engine only supports mouse and keyboard input. The functions and defines can be found in AEInput.h.

Getting mouse position is straightforward with **AEInputGetCursorPosition**:

```
// get mouse's x and y positions relative to the window screen space
s32 x, y;
AEInputGetCursorPosition(&x, &y);
```

When it comes to key input (via mouse buttons or keyboard keys), pay close attention to 'how' a key is being pressed. The OS can only tell us whether a key is pressed or released. Alpha Engine takes a step further by recording whether a key is pressed/released the previous frame.

For common cases, the following functions might be good enough for your application:

- **AEInputCheckTriggered**: This checks if a key is just pressed
- **AEInputCheckReleased**: This checks if a key is just released

For example:

```
// Checks if escape key is recently pressed
if AEInputCheckTriggered(AEVK_ESCAPE) { ... }
```

But if you want something a little more, like 'holding down a key', you will have to do a bit more work on your end. There are many ways to do this, but one way involves checking the key's state on the previous frame and the current frame with **AEInputCheckPrev** and **AEInputCheckCurr** respectively.

Below is a table that shows how one would interpret the combination of previous and current key states:

Previous frame	Current frame	Comments about the key
0	0	Key is not pressed at all
1	0	Key Is recently released
0	1	Key is recently pressed
1	1	Key is held