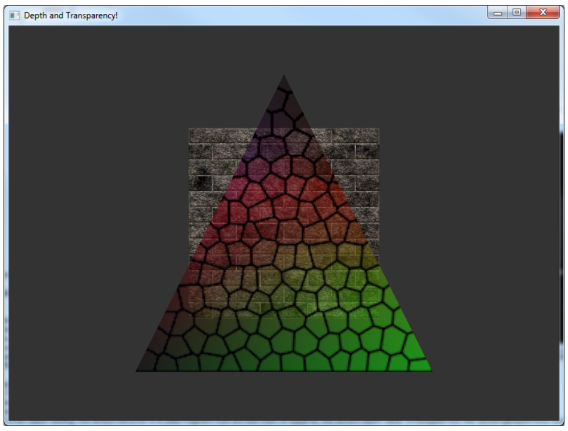
Tutorial 4: Depth and Transparency



Summary

In this tutorial, you are going to learn about how OpenGL determines which objects are in front of others in a scene - it's not as easy at it seems! Alpha blending is also introduced, which allows objects in our OpenGL scene have a varying amount of transparency.

New Concepts

Alpha, alpha blending, depth-buffer, z-fighting, depth testing

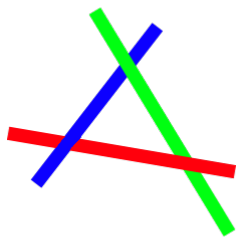
Introduction

As the scenes you attempt to create using OpenGL get more complex, you'll probably want some objects to appear in front of others. You'll also probably want some objects to be partially transparent, such as windows, or water. To do these things, you'll need to learn about handling the *depth* of a pixel on screen, and also handling its *alpha* value. Occasionally, the interaction between these values causes problems, so you'll also need to learn a few tricks to get the scene behaving as you want.

Depth Buffer

When drawing a 3D scene, objects may overlap on screen. Take for example, an enemy hiding behind a crate in front of you in a 1st person shooter. If the crate is drawn after the enemy, everything looks fine - the crate will obscure the enemy. But what about if the crate is drawn first? The back buffer you're drawing into only contains colour values, so there is nothing to determine whether the crate is in front or behind of the existing pixels, so the enemy will just be drawn over the top - now our enemy appears to be hiding *in front* of the crate!

Now, we *could* solve this by simply sorting the scene objects by their distance from the camera, the so called *Painter's Algorithm*, and this might work well in a lot of instances. But consider this: what about the scenario where object A overlaps object B and is overlapped by object C, but object B overlaps part of object C. What is the correct order to draw these objects? No matter which order you draw the objects in, part of one object will be incorrectly drawn in front of another.



There's no correct ordering for these objects, they all partially overlap each other

This can be solved by using a *depth buffer*, which unlike colour buffers, stores a value derived from the NDC space z-axis value of every pixel in the back buffer - that's why the depth bufer is sometimes called a *z-buffer*. As the the middle of NDC space to the edge has a range of 1.0, the depth buffer stores values between 0.0 and 1.0, with 0.0 being as close to the viewpoint, and 1.0 being as far away as possible while still being within the far z distance of the projection matrix.

With a depth buffer available, OpenGL can perform a *depth test* when trying to draw to the back buffer - by working out the eye-space z coordinate of the currently drawn screen fragment, it can be determined whether the fragment is in front of, or behind, any existing pixel data in the back buffer. If this value is less than what is currently in the depth buffer, then the fragment must be in front of whatever object was drawn to the back buffer earlier, so the new fragment *passes* the depth-test, and is drawn to the back and depth buffers. If it's more than the existing depth buffer value, it *fails* the depth test, and is discarded. Modern graphics hardware can sometimes perform this depth test *before* running the active fragment shader, so depth-testing can have a performance benefit, too. The direction of the depth test is generally configurable - you may want only fragments that are further away pass the depth test, for example.

The value stored in the depth buffer is interpolated from the z-axis values of the vertices being rendered. It's also important to note that this is the z value calculated *after* the perspective divide by the w component of the homogeneous coordinate. A side effect of this is that the values written to the depth buffer are not *linear*. Due to this nonlinearity, there is a varying amount of precision available in the depth buffer, depending on distance. The further away a fragment is, the less precision there is to store its depth. There are also only a finite number of bits available to store the depth buffer values, usually either 16 or 24 bits - with modern graphics hardware often supporting up to 32 bits. These two factors can cause an issue known as *z-fighting*. Due to the limited precision imposed by the number of depth bits used, it can occasionally happen that two surfaces that are very close to each other in distance end up with the 'same' rounded-off depth value. If the camera or surfaces move slightly, the rounding changes slightly, and the two surfaces seem to flicker, as their depth values are constantly rounded to be slightly lower than the other.

Transparency

In the the previous tutorials, you were introduced to colours that had *four* components - red, green, blue, and *alpha*. This alpha component represents how opaque the colour is - running from 0 (fully see-through) to 255 (fully opaque). Like with the colour interpolation you saw in tutorial 1, alpha is interpolated on a per-fragment basis, too. When coloured geometry is drawn to the back buffer, the alpha value of the currently processed fragment is used to determine what the final outputted fragment colour will be. If it's 255 (or 1.0 when using floating point values), then the outputted fragment colour is easy - it's whatever colour was output by the fragment shader, as the colour is fully *opaque*. If it's *less* than 255 (but greater than 0!) then the result will be *blended* with the current fragment colour to determine the final colour that will be written to the back buffer. This blending process is automatic (you don't have to perform it manually in a fragment shader), and also includes some controllable parameters. The equation used to determine the final blended colour is as follows:

$$\textit{Final Colour} = \textit{(Source Colour} \cdot \textit{Source Factor)} + \textit{(Destination Colour} \cdot \textit{Destination Factor)}$$

The source and destination colours are the current fragment output, and the current buffer value for that fragment position, respectively. The source and destination factors are controllable values - they could be 1, 0, the source or destination alpha value, or one of a number of API dependent variations.

Transparency and depth issues

Unfortunately, although the depth buffer allows *opaque* objects to be drawn in arbitrary order correctly, it has issues with transparent objects. The first issue, is that alpha blended fragments, even if they have an alpha value of 0, result in a write to the depth-buffer. So even a completely zero-alpha object was rendered to the screen, it would affect the depth-buffer, with potentially adverse affects. Writing to the depth buffer could be temporarily disabled to avoid this, but then a partially-alpha blended object could be overdrawn by an object further away, due to the lack of depth information.

The way blending is calculated also raises issues. Let's go back to our hiding enemy example. Instead of hiding behind a crate, imagine the enemy in our 1st person shooter has decided to hide behind a stained-glass window, which in rendering terms we imagine is just a quad with a stained glass texture on it, complete with alpha values to make the glass transparent. If the window is drawn after the enemy, everything is fine - both the alpha blending and the depth buffer work as intended, and our enemy is visible behind the stained glass window. If we draw them in the opposite order, what happens? If the depth buffer is *enabled*, our enemy disappears - every fragment the enemy is made up of has a depth greater than the stained-glass window, and so is discarded. If it's *disabled*, the enemy gets drawn in front of the window, and the stained-glass effect is broken - the stained glass colours will have been blended against the scene as it was *before* the enemy was drawn, resulting in incorrect colours.

When using transparency in graphics rendering, *render order matters*! A popular solution to this problem is to keep track of which objects are transparent, and have *two* render loops. The first loop draws all of the *opaque* objects, and then the second loop draws the *transparent* objects - sorted from back to front so that blending works correctly. This isn't perfect, as intersecting transparent objects won't have correctly blended colours, but the only other solution would be to sort each transparent *triangle* by distance - and even then, any intersecting triangles may have to be cut up further to get a completely error-free final render. You'll see an example of how to sort potentially transparent objects in a later tutorial.

Example program

To demonstrate what has been discussed in this tutorial, we're going to write a short example program that uses both depth and transparency. It'll be simple, but should give you some practical experience with dealing with transparent objects, and the issues that can arise when using the depth buffer. There's no new shaders this time, we're going to use the ones we made in Tutorial 2 - neither depth testing or alpha blending require any extra work in the shader! We're also going to write a new Mesh class function, one that will create a quad - we'll be using it again in later tutorials!

Mesh header file

We need one new **public** function, a static function that will return a fully formed quad *Mesh*.

```
...
public:
   static Mesh*   GenerateQuad();
...
```

mesh.h

Mesh Class file

The function itself is similar to *GenerateTriangle* in tutorial 1. Note that this time we're using triangle *strips* as the primitive type.

```
Mesh* Mesh::GenerateQuad() {
   Mesh* m                = new Mesh();
   m->numVertices         = 4;
   m->type                = GL_TRIANGLE_STRIP;

   m->vertices            = new Vector3[m->numVertices];
   m->textureCoords       = new Vector2[m->numVertices];
   m->colours             = new Vector4[m->numVertices];

   m->vertices[0]         =  Vector3(-1.0f,-1.0f, 0.0f);
   m->vertices[1]         =  Vector3(-1.0f, 1.0f, 0.0f);
   m->vertices[2]         =  Vector3( 1.0f,-1.0f, 0.0f);
   m->vertices[3]         =  Vector3( 1.0f, 1.0f, 0.0f);

   m->textureCoords[0]    = Vector2(0.0f, 1.0f);
   m->textureCoords[1]    = Vector2(0.0f, 0.0f);
   m->textureCoords[2]    = Vector2(1.0f, 1.0f);
   m->textureCoords[3]    = Vector2(1.0f, 0.0f);

   for(int i = 0; i < 4; ++i) {
      m->colours[i]       = Vector4(1.0f, 1.0f,1.0f,1.0f);
   }

   m->BufferData();
   return m;
}
```

mesh.cpp

Renderer header file

Our *Renderer* class header file is pretty mundane - new things in this tutorial are 4 new **public** functions, to control the depth and transparency state. We have two textures and *Meshes*, and two vectors to keep the world-space position of two objects. After that, we have our matrices, and some variables to store the state of our depth and transparency.

```cpp
#pragma once

#include "./nclgl/OGLRenderer.h"

class Renderer : public OGLRenderer {
public:
   Renderer(Window &parent);
   virtual ~Renderer(void);

   virtual void RenderScene();

   void  ToggleObject();
   void  ToggleDepth();
   void  ToggleAlphaBlend();
   void  ToggleBlendMode();
   void  MoveObject(float by);

protected:
   GLuint   textures[2];
   Mesh*    meshes[2];
   Vector3  positions[2];

   Matrix4   textureMatrix;
   Matrix4   modelMatrix;
   Matrix4   projMatrix;
   Matrix4   viewMatrix;

   bool  modifyObject;
   bool  usingDepth;
   bool  usingAlpha;
   int   blendMode;
};
```

renderer.h

Renderer Class file

The **constructor** for our *Renderer* class this tutorial is pretty straight forward. We instantiate two *Meshes* (one triangle and one quad), two textures, and two positions. Then we create our shader executable as normal - we're using the shaders we created in the texturing tutorial again, so the shader initialisation is identical to Tutorial 2! We must also set default values for the various toggle variables, and create a projection matrix, in this case the same perspective matrix we used in tutorial 2.

```
#include "Renderer.h"
Renderer::Renderer(Window &parent) : OGLRenderer(parent) {
   meshes[0] = Mesh::GenerateQuad();
   meshes[1] = Mesh::GenerateTriangle();

   meshes[0]->SetTexture(SOIL_load_OGL_texture(
      TEXTUREDIR"brick.tga",
      SOIL_LOAD_AUTO,SOIL_CREATE_NEW_ID, 0));

   meshes[1]->SetTexture(SOIL_load_OGL_texture(
      TEXTUREDIR"stainedglass.tga",
      SOIL_LOAD_AUTO,SOIL_CREATE_NEW_ID, 0));

   if(!textures[0] || !textures[1]) {
      return;
   }

   positions[0] = Vector3(0,0,-5);  //5 units away from the viewpoint
   positions[1] = Vector3(0,0,-5);

   currentShader = new Shader(SHADERDIR"TexturedVertex.glsl",
                    SHADERDIR"TexturedFragment.glsl");

   if(!currentShader->LinkProgram()) {
      return;
   }
   usingDepth     = false;
   usingAlpha     = false;
   blendMode      = 0;
   modifyObject   = true;

   projMatrix = Matrix4::Perspective(1.0f,100.0f,
             (float)width/(float)height,45.0f);

   init = true;
}
```

renderer.cpp

The **destructor** for our *Renderer* just **deletes** everything we created in the **constructor**, as usual. Remember, the *Mesh* class **destructors** will handle the deletion of the textures!

```
Renderer::~Renderer(void)  {
   delete meshes[0];
   delete meshes[1];
}
```

renderer.cpp

Our *RenderScene* function in this tutorial is pretty similar to previous tutorials - the new features introduced in this Tutorial are switched on and off elsewhere - remember OpenGL is a state machine, so it doesn't matter where functionality is set. We set our shader and its matrices, and bind our shader's texture sampler to texture unit 0. Then, we draw each mesh in turn, at its position, using the appropriate texture, before clearing up and swapping our buffers. Note how now our **glClear** function has two values **OR**ed together. As we now have a depth buffer as well as a colour buffer, we must clear that every frame too, using the **GL_DEPTH_BUFFER_BIT** symbolic constant. This will set every value in the depth buffer to a value of 1.0 - the very furthest edge of our Normalised Device Coordinate space.

```
void Renderer::RenderScene()  {
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glUseProgram(currentShader->GetProgram());

   glUniformMatrix4fv(glGetUniformLocation(currentShader->GetProgram(),
       "textureMatrix")   ,1,false, (float*)&textureMatrix);
   glUniformMatrix4fv(glGetUniformLocation(currentShader->GetProgram(),
       "viewMatrix")   ,1,false, (float*)&viewMatrix);
   glUniformMatrix4fv(glGetUniformLocation(currentShader->GetProgram(),
      "projMatrix")   ,1,false, (float*)&projMatrix);

   glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
      "diffuseTex"), 0);
   glActiveTexture(GL_TEXTURE0);
   for(unsigned int i = 0; i < 2; ++i) {
      glUniformMatrix4fv(glGetUniformLocation(
         currentShader->GetProgram(), "modelMatrix")  ,1,false,
         (float*)&Matrix4::Translation(positions[i]));
      glBindTexture(GL_TEXTURE_2D, textures[i]);
      meshes[i]->Draw();
   }
   glUseProgram(0);
   SwapBuffers();
}
```

renderer.cpp

To swap between the two objects in the scene, we have a very simple *ToggleObject* function, and to move the currently selected object, we have a *MoveObject* function. We use the **NOT** boolean operator to flip a **bool**, which we cast to an **int** to use as an index into our positions array, and modify the relevent position by a **float** value sent from our main loop.

```
void  Renderer::ToggleObject() {
   modifyObject = !modifyObject;
}
void  Renderer::MoveObject(float by)   {
   positions[(int)modifyObject].z += by;
}
```

renderer.cpp

ToggleDepth and *ToggleAlphaBlend* should be fairly explanatory, and work in a similar manner to the toggle functions in the last tutorial - we use the **NOT** boolean operator to flip our **bool** values, then use the ternary operator to swap between **glEnable** and **glDisable** for the OpenGL functionality we wish to toggle. *ToggleBlendMode* works pretty much the same way - except instead of a *boolean operator*, we use the **modulo** operator to cycle through 4 example types of alpha blending. The **glBlendFunc** OpenGL function sets the source and destination blending *factors* to be used to blend transparent fragments, as you saw earlier in the tutorial.

```
void  Renderer::ToggleDepth() {
   usingDepth = !usingDepth;
   usingDepth ? glEnable(GL_DEPTH_TEST) : glDisable(GL_DEPTH_TEST);
}

void  Renderer::ToggleAlphaBlend()  {
   usingAlpha = !usingAlpha;
   usingAlpha ? glEnable(GL_BLEND) : glDisable(GL_BLEND);
}

void  Renderer::ToggleBlendMode()   {
   blendMode = (blendMode+1)%4;

   switch(blendMode){
      case(0):glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);break;
      case(1):glBlendFunc(GL_SRC_COLOR, GL_ONE_MINUS_SRC_COLOR);break;
      case(2):glBlendFunc(GL_ONE, GL_ZERO);break;
      case(3):glBlendFunc(GL_SRC_ALPHA,GL_ONE);break;
   };
}
```

renderer.cpp

Main file

There's nothing too surprising in the Main file! Just some keyboard checks to toggle our depth and alpha functionality, and to move our triangles backwards and forwards.

```
#pragma comment(lib, "nclgl.lib")
#include "./nclgl/window.h"
#include "Renderer.h"

int main()  {
   Window w("Transparency and Depth!", 800 , 600, false);
   if(!w.HasInitialised()) {
      return -1;
   }
   Renderer renderer(w);
   if(!renderer.HasInitialised()) {
      return -1;
   }

   while(w.UpdateWindow() &&
   !Window::GetKeyboard()->KeyDown(KEYBOARD_ESCAPE)){
      if(Window::GetKeyboard()->KeyTriggered(KEYBOARD_1)) {
         renderer.ToggleEntity();
      }
      if(Window::GetKeyboard()->KeyTriggered(KEYBOARD_2)) {
         renderer.ToggleDepth();
      }
      if(Window::GetKeyboard()->KeyTriggered(KEYBOARD_3)) {
         renderer.ToggleAlphaBlend();
      }
      if(Window::GetKeyboard()->KeyTriggered(KEYBOARD_4)) {
         renderer.ToggleBlendMode();
      }
```

```

        if(Window::GetKeyboard()->KeyDown(KEYBOARD_UP)) {
            renderer.MoveObject(1.0f);
        }
        if(Window::GetKeyboard()->KeyDown(KEYBOARD_DOWN)) {
            renderer.MoveObject(-1.0f);
        }

        renderer.RenderScene();
    }
    return 0;
}
```

Tutorial4.cpp

Running the Program

Upon successful compilation and running of this program, you should be able to use the up and down arrow keys to move the meshes back and forth, and toggle transparency and depth buffer usage. When running the program, by default you'll be controlling the triangle. Note that no matter how long you press the up arrow for, it never seems to move behind the quad. That's because there's no depth testing going on by default, so the object rendererd second, in this case the triangle, will *always* be drawn on top of whatever had already been drawn. Now try pressing the *2* key - if you moved the triangle far away enough, it'll dissapear! *2* enables depth testing, so no matter which order our triangles are drawn in, the closer object will always appear to be in front of the further away one. If you press the back key long enough, the triangle will pop back into view, as it'll eventually be in front of the quad. Now, with the triangle in front of the quad, press *3*, to enable alpha blending. You should see something like the picture at the start of this tutorial. The coloured glass segments of the triangle texture have an alpha value of 128, so they are partially transparent. You can then toggle between different alpha blending factors using the *4* key.

Tutorial Summary

You should be able to understand how alpha blending works, and the different functions that can be applied to transparency. You should also have a basic understanding of the depth buffer, how to enable it, what it contains, and what pitfalls you might encounter when using depth alongside transparency. Later in this tutorial series we'll build upon this and take a look at some *scene management*, in order to better manage these depth problems, and render the scenes we create more efficiently. Next tutorial, though, we're going to take a look at another type of buffer, the *Stencil Buffer*.

Further Work

1) Testing against the depth buffer and writing to it can be controlled seperately. Investigate the **glDepthMask** function.

2) When depth testing, the default behavior is to pass fragment that are less than, or equal to, the current depth value. Investigate how the **glDepthFunc** function can change this behaviour.

3) Previous versions of OpenGL had an alpha testing feature, that would discard a fragment entirely if it had an alpha value less than a certain value - handy for skipping depth-writes for fragments with an alpha of 0! Unfortunately, this has been removed from the OpenGL 3 core profile. How could this functionality be replicated in a fragment shader? *Hint: you can use **if** statements in GLSL fragment shaders, and there is a fragment shader function called **discard**...*