SCIT

School of Computing & Information Technology

CSCI336 – Interactive Computer Graphics

Viewing and Projection

In this tutorial, we will look at viewing and projection transformations in OpenGL using the GLM library and loading models using the Open Asset Import library.

Index Buffer Object

In previous tutorials, we have always defined separate vertices for each triangle. However, it is possible to render primitives using share vertices using index buffers. For example:

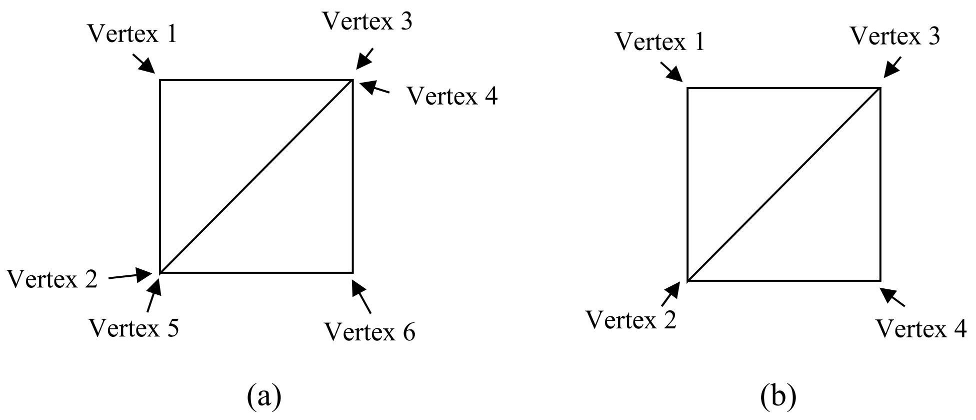


Figure 1

In figure 1(a), the square is made up of two triangles that do not share any vertices. Therefore, the total number of vertices that must be specified is 6. Whereas for the square in figure 1(b), vertex 2 and 3 are shared between both triangles, thus reducing the total number of vertices to 4.

Open the Tutorial4a visual studio project and have a look at the code in Tutorial4a.cpp. Two sets of vertices are defined; g_vertices1[] has 6 vertices, as it is for a square made up of two triangles that do not share any vertices; whereas g_vertices2[] has only 4 vertices. This is followed by an index array:

```
GLuint g_indices[] = {
       0, 1, 2,      // triangle 1
       2, 1, 3,      // triangle 2
};
```

The index values refer to the vertices that form the two triangles based on the 4 vertices in g_vertices2[]; i.e. triangle 1 is made up of vertex 1, 2 and 3; and triangle 2 is made up of vertex 3, 2 and 4. Refer to the image in figure 1(b).

Next, an identifier for the index buffer object is declared:

```
GLuint g_IBO;
```

VBOs are created for both objects and their vertex data copied to the GPU:

```
	glGenBuffers(2, g_VBO);
	glBindBuffer(GL_ARRAY_BUFFER, g_VBO[0]);
	glBufferData(GL_ARRAY_BUFFER, sizeof(g_vertices1), g_vertices1, GL_STATIC_DRAW);
	glBindBuffer(GL_ARRAY_BUFFER, g_VBO[1]);
	glBufferData(GL_ARRAY_BUFFER, sizeof(g_vertices2), g_vertices2, GL_STATIC_DRAW);
```

For the index buffer, we create an index buffer object, bind it to type GL_ELEMENT_ARRAY_BUFFER and copy the index data to the GPU:

```
	glGenBuffers(1, &g_IBO);
	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, g_IBO);
	glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(g_indices), g_indices, GL_STATIC_DRAW);
```

Two VAOs are created and their states initialised. Note that the second VAO has the index buffer bound to it:

```
	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, g_IBO);
```

The first object is rendered using **glDrawArrays**:

```
	glBindVertexArray(g_VAO[0]);			// make VAO active
	glUniformMatrix4fv(g_modelMatrixIndex, 1, GL_FALSE, &g_modelMatrix[0][0][0]);
	glDrawArrays(GL_TRIANGLES, 0, 6);
```

Whereas the second object is rendered using **glDrawElements**:

```
	glBindVertexArray(g_VAO[1]);			// make VAO active
	glUniformMatrix4fv(g_modelMatrixIndex, 1, GL_FALSE, &g_modelMatrix[1][0][0]);
	glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

The second parameter of **glDrawElements** is the number of elements to be rendered. There are two triangles, which are specified by 6 indices. The third parameter is the type, and the last parameter is the element offset.

Compile and run the program.

You should see two squares. Notice that the two triangles for the square on the left have distinct colours. This is because the vertices of both triangles are not shared and can each have different attributes. On the other hand, while the square on the right was constructed using less vertices, the two triangles cannot have distinct colours, because each shared vertex only has one colour attribute.

Colour Cube

In the Tutorial4b visual studio project, we will create a colour cube using the index buffer approach. A cube consists of 8 vertices as shown in figure 2. Each vertex will have a distinct colour which will be interpolated across the triangles. The cube has 6 faces with each face consisting of two triangles, giving a total of 12 triangles.

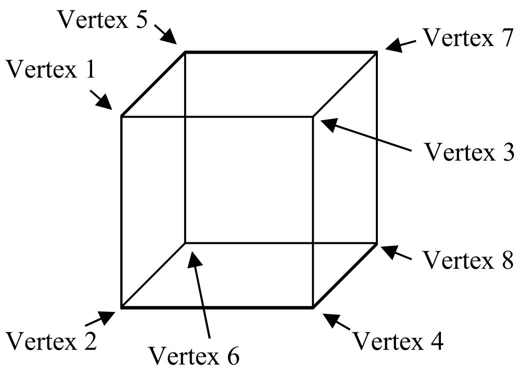


Figure 2

The following code defines the 8 vertices of the colour cube and indices of the 12 triangles:

```
Vertex g_vertices[] = {
       // vertex 1
       -0.5f, 0.5f, 0.5f,   // position
       1.0f, 0.0f, 1.0f,    // colour
       // vertex 2
       -0.5f, -0.5f, 0.5f,  // position
       1.0f, 0.0f, 0.0f,    // colour
       // vertex 3
       0.5f, 0.5f, 0.5f,    // position
       1.0f, 1.0f, 1.0f,    // colour
       // vertex 4
       0.5f, -0.5f, 0.5f,   // position
       1.0f, 1.0f, 0.0f,    // colour
       // vertex 5
       -0.5f, 0.5f, -0.5f,  // position
       0.0f, 0.0f, 1.0f,    // colour
       // vertex 6
       -0.5f, -0.5f, -0.5f,// position
       0.0f, 0.0f, 0.0f,    // colour
       // vertex 7
       0.5f, 0.5f, -0.5f,   // position
       0.0f, 1.0f, 1.0f,    // colour
       // vertex 8
       0.5f, -0.5f, -0.5f,  // position
       0.0f, 1.0f, 0.0f,    // colour
};

GLuint g_indices[] = {
       0, 1, 2,      // triangle 1
       2, 1, 3,      // triangle 2
       4, 5, 0,      // triangle 3
       0, 5, 1,      // ...
       2, 3, 6,
       6, 3, 7,
       4, 0, 6,
        6, 0, 2,
        1, 5, 3,
        3, 5, 7,
        5, 4, 7,
        7, 4, 6,        // triangle 12
};
```

Since this uses an index buffer to render the colour cube, we use **glDrawElements**. There are 6 faces which consist of 12 triangles, each triangle has 3 vertices. Hence, the number of elements is 12 x 3 = 36:

```
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);
```

Viewing and Projection Transformations

View and projection transformations are performed in the vertex shader. Therefore, the vertex shader is modified slightly to have a uniform variable to receive the result of the combined model, view and projection matrices. The following is in MVP_VS.vert:

```
uniform mat4 uModelViewProjectionMatrix;
```

The resulting position is transformed based on this matrix:

```
gl_Position = uModelViewProjectionMatrix * vec4(aPosition, 1.0);
```

The following is declared in Tutorial4b.cpp to store the uniform variable's location:

```
GLuint g_MVP_Index = 0;                 // location in shader
```

And its location in the shader is obtained as follows (in the init function):

```
g_MVP_Index = glGetUniformLocation(g_shaderProgramID, "uModelViewProjectionMatrix");
```

To store the viewing and projection matrices, the following are declared:

```
glm::mat4 g_viewMatrix;           // view matrix
glm::mat4 g_projectionMatrix;     // projection matrix
```

These matrices are initialised in the init function, using GLM functions:

```
g_viewMatrix = glm::lookAt(glm::vec3(0.0f, 0.0f, 4.0f), glm::vec3(0.0f, 0.0f, 3.0f),
       glm::vec3(0.0f, 1.0f, 0.0f));
```

This uses the **glm::lookAt** function to create a view matrix. The first vector sets the position of the camera to be 3 units away from the origin in the +ve z-direction. The second vector sets the camera look at a position that is 2 units away from the origin in the +ve z-direction. This means that the direction of the camera is in the -ve z-direction. The third vector sets the camera's up vector.

Next, to initialise the projection matrix, we need to find the aspect ratio. To do this we get the framebuffer's width and height, before computing the aspect ratio:

```
int width, height;
```

```
glfwGetFramebufferSize(window, &width, &height);
float aspectRatio = static_cast<float>(width) / height;
```

Once we have the aspect ratio, we can use the **glm::perspective** function to create a perspective projection matrix:

```
g_projectionMatrix = glm::perspective(glm::radians(45.0f), aspectRatio, 0.1f, 100.0f);
```

The first parameter of **glm::perspective** is the field of view, the second parameter is the aspect ratio, the third parameter is the distance to the near clipping plane, whereas the last parameter is the distance to the far clipping plane.

With the view and projection matrices initialised, we can render the object. Before we do this, we concatenate the model, view and projection matrices, then pass this to the shader:

```
glm::mat4 MVP = g_projectionMatrix * g_viewMatrix * g_modelMatrix;

glUniformMatrix4fv(g_MVP_Index, 1, GL_FALSE, &MVP[0][0]);

glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);
```

The code also has an update_scene function that allows us to move, rotate and scale the colour cube based on keyboard input, by updating the model matrix. Note that on some systems you may have to lower the sensitivity values, otherwise the object will move very fast. The section on frame rates below explains how to overcome this problem.

Depth Buffer Test

Compile and run the program. Notice that the colour cube doesn't look right. This is more obvious when you move and rotate the colour cube.

The reason for this is because visibility determination was not enabled. This means that polygons that are rendered later will simply overwrite previously rendered polygons, without checking which polygon should be in front and which should be behind. To overcome this, we need to enable the depth buffer test. This can be done in the `init` function:

```
    glEnable(GL_DEPTH_TEST);    // enable depth buffer test
```

Before rendering, the depth buffer bits must be cleared, along with the colour buffer bits (in the render_scene function):

```
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Compile and run the program. It should now render the colour cube correctly.

Viewport Transformation

Instead of rendering to the entire framebuffer (the default viewport transformation), we can change the viewport transform to set the portion of the frame that we want to render to. This is done using the **glViewport** function.

If the viewport transformation is not set, a default one will use. To see what effect of the default viewport transformation, comment out the following line in the render_scene function:

```
    //glViewport(0, 0, g_WindowWidth, g_WindowHeight);
```

Compile and run the program. Try resizing the window.

Now uncomment that line of code, run the program and try resizing the window. On some systems, the behaviour will be no different, on others you will notice a difference.

If we want to scale the rendering to fit the window, we need to update the width and height arguments that are passed to **glViewport**. To do this, uncomment the following line of code in the main function:

```
glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
```

This registers a function that will be called whenever the window is resized:

```
static void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
    g_WindowWidth = width;
    g_WindowHeight = height;
}
```

Notice that this function updates the window width and height variables.

Run the program now and try resizing the window. Notice any difference?

Let's say we only want to render to the centre of the frame, we can create a border of 100 pixels from the top, down, left and right boundaries of the frame where we do not want anything to be drawn. To do this, we can add the following (in the render_scene function):

```
glViewport(100, 100, g_WindowWidth - 200, g_WindowHeight - 200);
```

Compile and run the program. Move the colour cube to the sides, notice that it will not be rendered when it is close to the sides. Notice also that the scene appears smaller, since the scene has been projected to fit within a smaller rendering window.

If you don't want the scale the rendering to fit the window size, comment out the following line:

```
//glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
```

So that the window width and height variables are not updated when the window is resized. Alternatively, you can pass the values directly:

```
glViewport(100, 100, 512 - 200, 512 - 200);
```

Compile and run the program. Try resizing the window now.

Loading Models

A model's 3D mesh is simply a collection of information like vertex positions, normals, texture coordinates, etc. contained in a file, using a certain 3D model file format. In this section, we will look at a basic example for loading and displaying 3D models based on the Wavefront .obj file format. The .obj file format is one of the simplest 3D mesh file formats. You can open it using a word editor to view the contents. For example, the contents will look something like this:

```
v 0.437500 0.164063 0.765625
v -0.437500 0.164063 0.765625
v 0.500000 0.093750 0.687500
vn 0.189764 -0.003571 0.981811
vn 0.646809 -0.758202 0.082095
vn 0.999573 -0.014496 -0.024445
vt 0.315596 0.792535
vt 0.331462 0.787091
vt 0.331944 0.799704
```

where 'v' is for a vertex position, 'vn' is for a vertex normal, 'vt' is for a texture coordinate.

While we can write our own reader to import the contents, this is tedious. This section demonstrates how to use the Open Asset Import Library (Assimp) to load and display simple .obj files. Open the Tutorial4c Visual Studio project. Have a look in the GraphicsSDK\include folder, notice that there is an "assimp" folder. This folder contains all the assimp header files. At the start of the Tutorial4c.cpp file, you will notice that we include some of these headers files:

```
#include <assimp/cimport.h>
#include <assimp/scene.h>
#include <assimp/postprocess.h>
```

We also declare a struct to store mesh properties, i.e. mesh vertices and the number of vertices, and a variable for the mesh:

```
typedef struct Mesh
{
	Vertex* pMeshVertices;			// pointer to mesh vertices
	GLint numberOfVertices;			// number of vertices in the mesh
} Mesh;

Mesh g_mesh;					// mesh
```

Next, have a look at the load_mesh function which accepts a model file name as one of its arguments. The first line uses an assimp importer function to load a scene object from the file:

```
const aiScene* pScene = aiImportFile(fileName, aiProcess_Triangulate);
```

From the scene object, we can obtain a mesh. For simple models, we assume that it only contains a single mesh (this is not true for complicated models, which will require us to loop through all the meshes in a model):

```
const aiMesh* pMesh = pScene->mMeshes[0];
```

From this, we can obtain the number of mesh vertices:

```
mesh->numberOfVertices = pMesh->mNumVertices;
```

Then we check whether the mesh contains vertex coordinates. If it does, we allocate enough memory to store the vertex information and read the vertex information from the mesh into an array, which we will later pass to a VBO.

```
if (pMesh->HasPositions())
{
	mesh->pMeshVertices = new Vertex[pMesh->mNumVertices];

	for (int i = 0; i < pMesh->mNumVertices; i++)
	{
		const aiVector3D* pVertexPos = &(pMesh->mVertices[i]);

		mesh->pMeshVertices[i].position[0] = (GLfloat)pVertexPos->x;
		mesh->pMeshVertices[i].position[1] = (GLfloat)pVertexPos->y;
		mesh->pMeshVertices[i].position[2] = (GLfloat)pVertexPos->z;

		// since we have no lighting, give each vertex a random colour
		mesh->pMeshVertices[i].color[0] = static_cast<double>(rand()) / RAND_MAX;
		mesh->pMeshVertices[i].color[1] = static_cast<double>(rand()) / RAND_MAX;
		mesh->pMeshVertices[i].color[2] = static_cast<double>(rand()) / RAND_MAX;
	}
}
```

Finally, we release the asset import scene object:

```
aiReleaseImport(pScene);
```

To use this function, we simply call it with the model name and path (in the init function):

```
load_mesh("models/cube.obj", &g_mesh);
```

We then create the VBO and VAO and buffer the contents that we read from the mesh file:

```
glBufferData(GL_ARRAY_BUFFER, sizeof(Vertex)*g_mesh.numberOfVertices, g_mesh.pMeshVertices,
	GL_STATIC_DRAW);
```

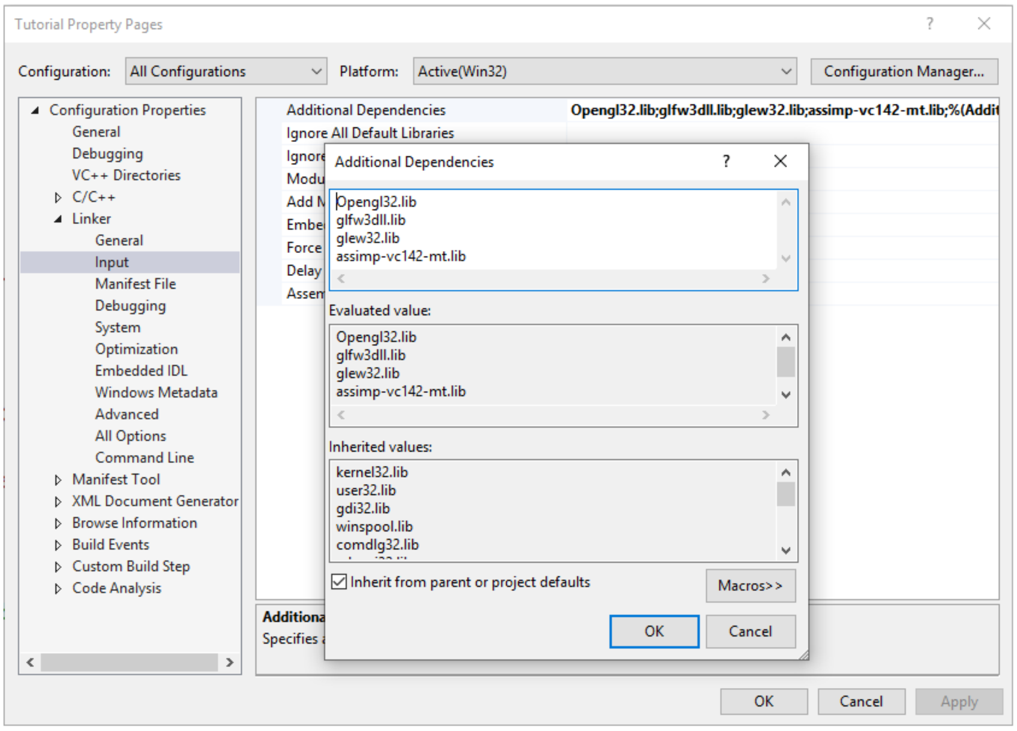
To render the mesh, we draw triangles based on the number of mesh vertices:

```
	glDrawArrays(GL_TRIANGLES, 0, g_mesh.numberOfVertices);
```

Don’t forget to deallocate memory before exiting the program:

```
	if (g_mesh.pMeshVertices)
		delete[] g_mesh.pMeshVertices;
```

Like the AntTweakBar library, you will need the assimp dynamic link library, i.e. assimp-vc142-mt.dll. This is located in the same directory as the .cpp files in Tutorial4c. However, if you want to distribute your program, the .dll file needs to be in the same directory as the application program’s .exe file. Also, note that there is a file called assimp-vc142-mt.lib in the GraphicsSDK\lib folder that you will have to link before you can compile the code. Refer to the “Setting up the Environment” document in Tutorial1 if you don’t know how to do this.


Tutorial Property Pages

Configuration: All Configurations   Platform: Active(Win32)

Additional Dependencies: Opengl32.lib;glfw3dll.lib;glew32.lib;assimp-vc142-mt.lib;%(Addit

Additional Dependencies

```
Opengl32.lib
glfw3dll.lib
glew32.lib
assimp-vc142-mt.lib
```

Evaluated value:

```
Opengl32.lib
glfw3dll.lib
glew32.lib
assimp-vc142-mt.lib
```

Inherited values:

```
kernel32.lib
user32.lib
gdi32.lib
winspool.lib
comdlg32.lib
```

☑ Inherit from parent or project defaults

Compile and run the program.

Things to do

By now, you should understand viewing and projection transformations using matrices. Try modifying the program to do the following:

- Place the camera at different positions and look at the scene from different viewing directions
- Try loading some of the other model files located in the models folder of Tutorial4c

References

Among others, much of the material in this tutorial was sourced from:

- https://www.khronos.org/opengl/wiki/
- http://www.opengl-tutorial.org/
- http://ogldev.atspace.co.uk/index.html