Computer Graphics
Aum Patel

Contents

Chapter 1

Question 1 - Geometry and vertex attributes

Y-axis up, X axis left to right and Z axis towards you is a Right handed coordinate system. I have proven this by using my right-hand with the X-axis being on the thumb, the Y-axis being on the first finger and the Z-axis being on my middle finger and rotating it to meet the 3 requirements. You cannot rotate the left hand to do this.

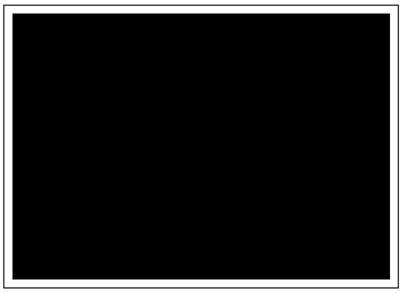

Figure 1.1: Image of hand with axis labeled.

I decided to draw out the model I was going to make in Blender first and then place the vertices on top of it.

The colors of the axis are consistent in all images: Red : X , Green : Y , Blue : Z

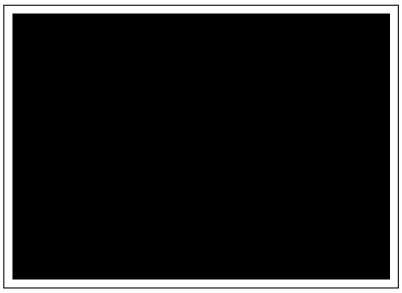

Figure 1.2: House in blender with drawn axis.

I chose the bottom vertex on the rear left of the house as the origin as I would only have to work with positive numbers for the rest of the vertices, and they would also be round numbers (except for the roof) as I am going to make the house with a square base of 1x1.

Below is a wireframe view of the same house with the coordinates labeled.

[redacted]

Figure 1.3: House in blender with wireframe and coordinates.

Coordinates for the vertices:

(a) (0, 0, 0)

(b) (0, 0, 1)

(c) (1, 0, 0)

(d) (1, 0, 1)

(e) (0, 1, 0)

(f) (0, 1, 1)

(g) (1, 1, 0)

(h) (1, 1, 1)

(i) (0.5, 2, 0)

(j) (0.5, 2, 1)

16 triangles will be used to make up the house, and there will be normals that are shared between them.

I calculated the vector normals by using the following equation:

If we have 3 vectors that make up a a triangle in an anti-clockwise manner: $V0, V1, V2$, to calculate the normal facing outwards we do:

$A = V1 - V0 \mid B = V2 - V0$

$Normal = A \times B$

Doing this with the first triangle on the table (triangle on the left face of the house) give you:

$A = (0, 0, 1) - (0, 0, 0)$
$A = (0, 0, 1)$

$B = (0, 1, 1) - (0, 0, 0)$
$B = (0, 1, 1)$

$Normal = (0, 0, 1) \times (0, 1, 1)$
$Normal = (-1, 0, 0)$

I confirmed this calculation to be correct by looking at the shape itself in 3D space, and $(-1, 0, 0)$ is the normal that would be correct.

I had no need to normalise the vectors as they output from the cross product was a sensible number.

Same calculation is done for the rest of the sides, triangles facing the same direction will have the same surface normals.

Triangles	Normals
(a, b, f) (a, f, e)	(-1, 0, 0)
(b, h, f) (b, d, h) (f, h, j)	(0, 0, 1)
(d, g, h) (d, c, g)	(1, 0, 0)
(c, e, g) (c, a, e) (g, e, i)	(0, 0, -1)
(a, c, b) (b, c, d)	(0, -1, 0)
(e, j, i) (e, f, j)	(-1, 0.5, 0)
(h, i, j) (h, g, i)	(1, 0.5, 0)

To write the .obj file, I have to change the alphabetical indices that I have been using into numerical. I have rounded the numbers to 6 decimal places. I have the same number of vertex normals as vertices as I decided to make the shape smooth shaded.

NOTE THE VERTEX NORMALS HAVE NOT BEEN NORMALISED.

v 0 0 0

v 0 0 1

v 1 0 0

v 1 0 1

v 0 1 0

v 0 1 1

v 1 1 0

v 1 1 1

v 0.5 2 0

v 0.5 2 1

vn -0.333333 -0.333333 -0.333333

vn -0.333333 -0.333333 0.333333

vn 0.333333 -0.333333 -0.333333

vn 0.333333 -0.333333 0.333333

vn -0.666667 0.166667 -0.333333

vn -0.666667 0.166667 0.333333

vn 0.0 0.166667 -0.333333

vn 0.0 0.166667 0.333333

vn 0.0 0.333333 -0.333333

vn 0.0 0.333333 0.333333

usemtl matWall

f 1//1 2//2 6//6

f 1//1 6//6 5//5

f 2//2 8//8 6//6

f 2//2 4//4 8//8

f 6//6 8//8 10//10

f 4//4 7//7 8//8

f 4//4 3//3 7//7

f 3//3 5//5 7//7

f 3//3 1//1 5//5

f 7//7 5//5 9//9

f 1//1 3//3 2//2

f 2//2 3//3 4//4

usemtl matRoof

f 5//5 10//10 9//9

f 5//5 6//6 10//10

f 8//8 9//9 10//10

f 8//8 7//7 9//9

Chapter 2

Question 2 - Vertex Buffer Object(VBO) design and transformations

```
You now have to design a VBO to contain the vertex attributes for your house. Choose a suitable la
```

```
Your house is to be translated so it is centred at a new position on the ground plane, rotated by
```

My single VBO will look like this:

```
x y z x y z x y z r g b r g b r g b
```

which is the standard format, making the attributes tightly packed.

I will add the vertices for the triangles into an array of floats of size 3 * numberOfTriangles, three coordinates per triangle in the counter-clockwise order they should be drawn to render outside, the first two triangles will look like this followed by the rest of the vertices: triangleVertices[30] = {0, 0, 0,

0, 0, 1

0, 1, 1,

0, 0, 0,

0, 1, 1,

0, 1, 1,

...}

As the RGB values are in the VBO, I will initialise them as an array of size 30 as well. triangle-Colors[30] = {1, 1, 1,

1, 1, 1,

1, 1, 1,

1, 1, 1,

1, 1, 1,

1, 1, 1,

...}

Two uint (GLuint) variables will need to be created, one is VAO (will be called **vao**) and the other is VBO (will be called **vbo**).

We then call the following functions:

```
// Will have to create two arrays of floats that contain the vertices for all
    ↪  the triangles and the colours for all the triangles both size 10
const GLfloat triangleVertices[30] = {...‘will have all the vertices here as
    ↪ floats‘...}
const GLfloat triangleColours[30] = {...‘will have all the colour data here
    ↪ as floats‘...}

// Will generate a vertex array object (allocate space) and assign it to vao.
    ↪  We pass through a reference so that it will change the variable.
glGenVertexArrays(1, &vao);

// Bind the VAO created to the current object
glBindVertexArray(vao);

// We then allocate space and generate a vbo, again passing through a
    ↪ reference
glGenBuffers(1, &vbo)

// then bind the vbo to the target which is GL_ARRAY_BUFFER
glBindBuffer(GL_ARRAY_BUFFER, vbo)

// We pass in the data separately by sub-dividing the buffer, as we would
    ↪ like to allocate vertices and color to the same buffer. This could be
    ↪ done so that you create two separate buffers and VBOs like seen on
    ↪ https://www.khronos.org/opengl/wiki/Tutorial2:_VAOs,_VBOs,
    ↪ _Vertex_and_Fragment_Shaders_(C_/_SDL)#Compilation
// Here, 0 is the starting index and 30 is the sizeof(triangleVertices)
glBufferSubData(GL_ARRAY_BUFFER, 0, 30, triangleVertices); // vertices

// Here, 30 is the starting index and 30 is the sizeof(triangleColours)
glBufferSubData(GL_ARRAY_BUFFER, 30, 30, triangleColours); // colours

// we will enable the vertex attribute arrays one after the other creating
    ↪ the correct
glEnableVertexAttribArray(0);

glEnableVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);


glEnableVertexAttribArray(1);
// here the last parameter is a pointer to 30 as that is where the colour
    ↪ data starts in the buffer
glEnableVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (const GLvoid*) 30);
```

Now to transform it and then render the object we will use a glutil::MatrixStack to transform the object.

```

glUtil::MatrixStack matrixStack; // creating the stack
matrixStack.SetIdentity(); // setting it to an identity matrix

matrixStack.Push();
    // 14 points on X, 6 Points on Z, so it is still on the ground plane while
        ↪  being on a new positions
    matrixStack.Translate(14, 0, 6);
    // rotating 38 degrees on the y axis
    matrixStack.Rotate(glm::vec3(0, 1, 0), 38);
    // scaling the house by 3.7f uniform points
```

```
    matrixStack.Scale(3.7f);


    // now to render the triangles we pass in the vao that we created and then
        ↪  draw it on screen.
    glBindVertexArray(vao);
    glDrawElements(GL_TRIANGLES, , GL_UNSIGNED_INT, 0)
matrixStack.Pop();
```

Chapter 3

Question 3 - Camera Positioning

zNear and **zFar** define the distance of the near and far clipping planes. Anything that is in front of the camera but its distance is less than **zNear**, it will not be displayed; vice versa with **zFar**, anything in front of the camera but farther away than **zFar** will not be shown.

fovy is the field of view of the camera, giving a bigger number will increase the field of view and display more on screen, decreasing it will compress the image and only show a little portion of it. It is the angle θ of the separation of the planes on the Y axis. See image I created below that shows it better.

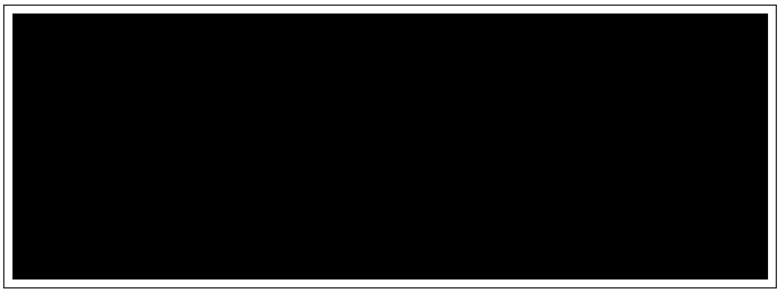

Figure 3.1: Diagram showing what the FOV number/angle means. In this image the two lines are 60° apart

The **aspect** is the ratio between the width (x) and the height (y) of the view, it can also be described as the field of view in the X axis.

Calling *gluPerspective*() with these parameters (which are all of type GLdouble) will set up a perspective projection matrix.

I will use the original position for house as seen in question 1 and not the position transformed to in question 2.

I will have the camera's perspective projection matrix be created with the following parameters:

```
fovy = 60.0 // an appropriate angle that is similar to the human eye
aspect = 1 // making it square
zNear = 1 // I do not want objects in front of the camera to clip to early
zFar = 50 // plenty of room for the far end of the clipping plane.
```

Now to position the camera and provide the forward and the up vector. I will position it so that the camera is pointing straight into it and looking at the front of the house.

```
// The house is sitting on the XZ plane and has a total height of 2, so I
    ↪ placed the camera on the center with 1 on the Y axis. As the house
    ↪ goes from 0 to 1 on the X, I decided to place the camera in the center
    ↪ , this being 0.5. As the house already extrudes by 1 on the Z axis, I
```

```
    moved the camera back by 2 extra points to make sure that the house will fit into the FOV, making it 3 on the Z.
cameraPosition = glm::vec3(0.5f, 1, 3)

// the forward is -1 on the Z as it is facing the front of the house that has  a normal of (0,0,1)
cameraForward = glm::vec3(0,0,-1)

// the up is going to be positive on the Y as the camera is not angled.
cameraUp = glm::vec3(0,1,0)
```

There are two methods of rotating the camera around a certain object while always keeping it in the center of view. One is by creating a Catmull Rom Spline that is a circle around the house.

The other is using sin and cos (using a trig unit circle) over delta time to get the correct positions of X and Z. I will be going with the later method as it seems the most straight forward.

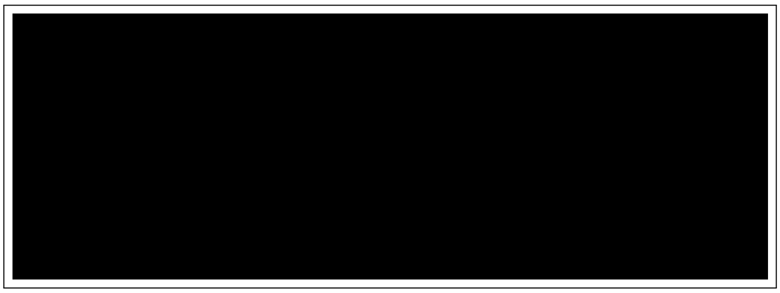

Figure 3.2: Diagram explaining using the trig unit circle finding out the x and z positions

We can use the glm::lookAt() method to get a viewing matrix that will change what is being seen in the world. The following code will be executed every frame, and am assuming delta time *dt* is being passed through. The radius will be 10 in this case, and will be offset by the house's position.

```
// set the radius
radius = 10

// the height from the house that the camera will be placed at
auto yPos = 1 + house.position.y // offset by the house's Y

// house.position is a vec3 with the house's position
// get the X position
auto xPos = (cos(dt) * radius) + house.position.x; // offset by house's X

// get the Z position
auto zPos = (sin(dt) * radius) + house.position.z; // offset by house's Z

cameraPosition = glm::vec3(xPos, yPos, zPos); // will be the vEye in the lookAt

cameraUp = glm::vec3(0,1,0); // will be the camera up variable

// updating the view matrix with the new camera data
// lookAt(eyePosition, viewPoint, cameraUpVector)
viewMatrix = glm::lookAt(cameraPosition, house.position, cameraUp);
```

Chapter 4

Question 4 - Shader Implementation