CMP203 Coursework Report

Justin Syfrig

1904770@uad.ac.uk

Git Username: nefastes

Introduction

This projects was inspired by the “serious room” from the game *The Stanley Parable*. The idea is to have the most “serious” table inside the room, as an inside joke of the game after the player attempts to cheat. In order to show more aspects of 3D graphics, I also added a mirror, some procedurally generated shapes and some models on the table. All 3D graphics techniques will be discussed inside the *Scene explanation* section. The application has been written in C++ using OpenGL.

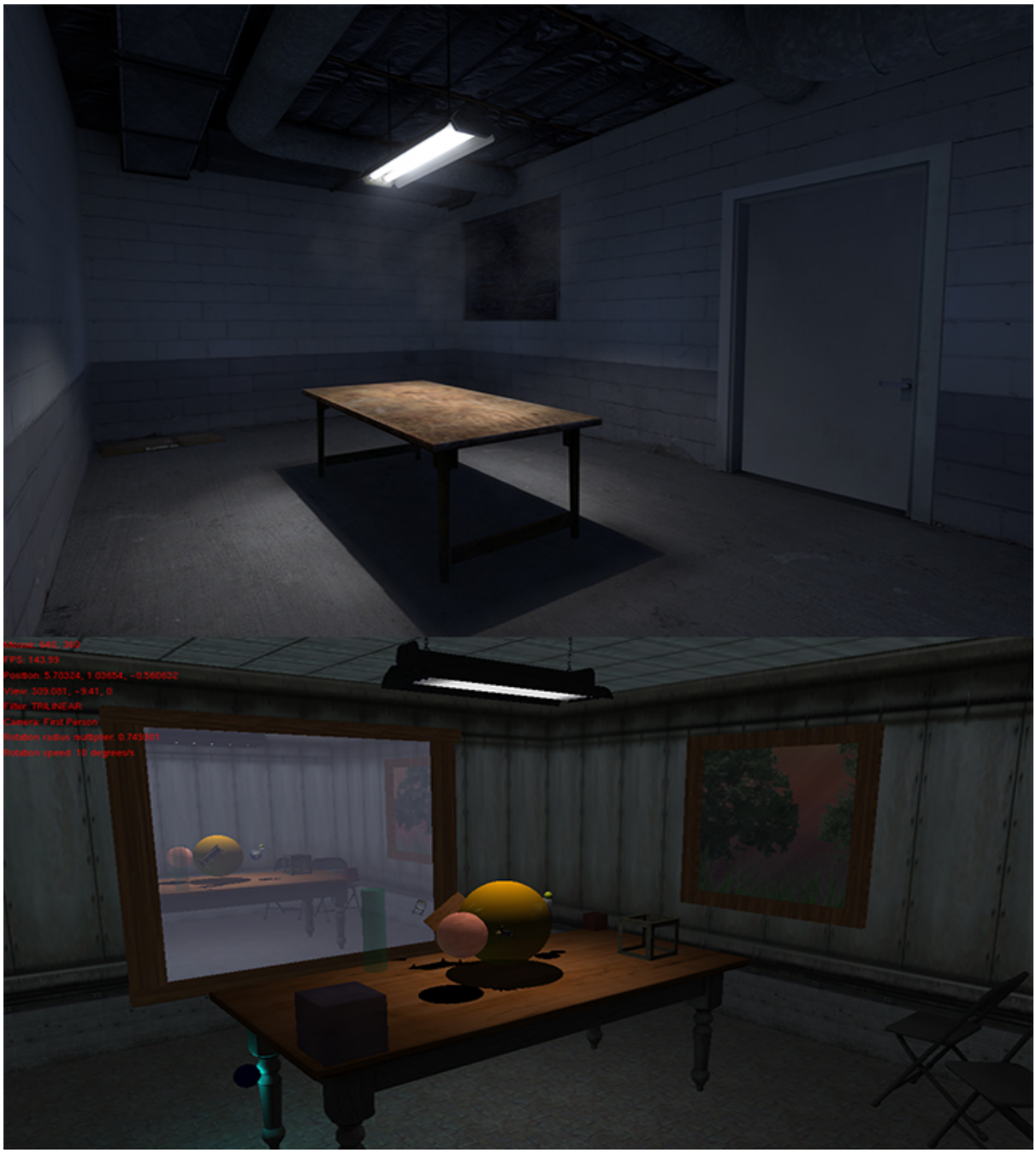

Figure 1: Comparison of the serious room in The Stanley Parable and my application (below)

Controls

This section I will cover the user controls over the application. Each **control key** is highlighted in bold, followed by a brief description of what it does.

F1 : Toggle wireframe mode

F2 : Toggle full-bright mode (deactivate/activate lighting, reflections and shadows)

F3 : Cycle through the texture filtering methods

F4 : Cycle through the camera types

F5 : Toggle “Steve mode”

W,**A**,**S**,**D** : While in first or third person camera, these keys can be used to move around

1,**2** : While in first or third person camera, these keys can be used to move down or up respectively

Q,**E** : While in first or third person camera, these keys can be used to rotate the camera horizontally

R,**F** : While in first or third person camera, these keys can be used to rotate the camera vertically

Mouse : The mouse can also be used to rotate the camera horizontally or vertically

T,**Y** : These keys can be used to decrease or increase the radius of the rotating objects

G,**H** : These keys can be used to decrease or increase the rotation speed of the rotating objects

Scene explanation

In this section I will explain how the scene has been built by going more in-depth with some key 3D graphics techniques. I divided these into smaller categories as a bulleted list.

- Camera

  The camera work is done using GLUT, an OpenGL toolkit that allows us to send coordinates for a position and two directions into a specific function “gluLookAt”. By doing so, we can move the view quite easily, as this function will calculate the projection transform with the given coordinates for us.

```
// Set the camera
gluLookAt(
    camera.getPosition(true).x, camera.getPosition(true).y, camera.getPosition(true).z,
    camera.getLookAt().x, camera.getLookAt().y, camera.getLookAt().z,
    camera.getUp().x, camera.getUp().y, camera.getUp().z
);
```

Figure 2: Example of the gluLookAt function call

In order to facilitate the calculation of these coordinates, I made a camera class. It will handle our inputs and return the new coordinates of the position and angles of the camera, which are represented by two vectors: the forward and the up vectors. The forward vector is returned by the *camera.getLookAt* getter, and the up vector by the *camera.getUp* getter. Both vectors are obtained with the parametric equation of a sphere. It is using the current orientation of our camera, modified by our inputs, as shown in the following figure:

```
//Update the forward vector
forward.x = sinY * cosP;
forward.y = sinP;
forward.z = cosP * -cosY;

//Update the up vector
up.x = -cosY * sinR - sinY * sinP * cosR;
up.y = cosP * cosR;
up.z = -sinY * sinR - sinP * cosR * -cosY;
```

Figure 3: Calculating the forward and up vectors using the parametric equation of a sphere

However, prior to this, we need to initialise a frustum, a geometry that defines a volume between two planes, the near and the far plane, so that we only render what is inside of this volume. This will drastically reduce the amount of calculations needed to render the scene, as we concentrate the rendering on that volume only and not on the entire scene. Luckily, the GLUT toolkit provides a couple of functions to help us with that as well.

Moving the camera is then a simple matter of adding the forward vector to our current position, and the same would apply for moving vertically using the up vector. Moving sideways is done with a side vector, which is a simple cross product of the forward and up vectors. Changing the orientation with the mouse involves measuring the displacement of the mouse to the centre of the window each frame, and putting it back on the centre afterwards, ready for the next frame. The distance it travelled horizontally will influence the camera's yaw angle, and the amount it travelled vertically will influence the pitch angle.

We now have a nicely working first person camera. Changing it to a third person camera can be done quite easily. In fact, by subtracting the forward vector from our current position, we now find ourselves looking about our current position. So we just need to trick the *gluLookAt* function by giving it this "false" position (because it is not the real position of the camera) and telling him that our "look-at-point" is the camera position. We can also decide to multiply the subtraction by a constant, so that our third person view is further to the position. The tracking camera was achieved by reusing the circle calculations of the moving blue light, which is rotating about the centre of the scene.

```
Vector3 Camera::getPosition(bool gluLookAt)
{
    //The currentCameraType variable allows us to get a "false" position when 3rd person is enabled
    if (currentCameraType == CameraTypes::THIRDPERSON &&  gluLookAt) return position - forward * thirdPersonDistanceMultiplier;
    else return position;
}

Vector3 Camera::getLookAt()
{
    if (currentCameraType == CameraTypes::THIRDPERSON) return position;
    else return position + forward;
}
```

Figure 4: Returning a false position and look-at-point when third person is enabled

• Generating objects

Generating shapes or models can be done quite easily using data structures called vertex arrays. There are three simple arrays (or vectors) required for a shape to be drawn: the vertices array, the normals array and the texture coordinates array. Each one of these will hold all the required data to render the shape. It can be as simple as a cube or as complex as a model. There are several techniques to render with these vertex arrays, I will briefly describe the two main methods I personally used. The first one involves drawing the shape based on the vertex order, which, as its name implies, draws the shapes considering that our arrays are in the correct order and will cycle trough the data from the beginning to the end of the arrays. The second method requires another array holding indices. In order to draw the shape, OpenGL will cycle through this array, retrieving an index in order to find the corresponding vertex, texture coordinate and normal in their respecting arrays. I only used this method to generate cylinders, everything else is using the first method.

1. BasicShape class

Instead of declaring similar properties of different basic shapes multiple times, I made a BasicShape class as a base class for all of my shapes. It contains all the main informations a shape needs, such as its position in the space, its colour, transparency, texture, etc. Additionally, this class will render a simple cube, using two virtual functions: *shapeSpecificDrawingMode* and *generateShape*. These functions will be

replaced by any other class inheriting from this BasicShape class in order to draw their specific shapes. The cube is initialised with vertex arrays when generating the shape and is rendered using quads.

2. Disc

My disc class inherits from the BasicShape class, so that it automatically contains a lot of useful data without the need of declaring them again. Additionally, two more variables are needed, a radius and a resolution, so that we can generate the shape using the equation of a circle. Unlike the cube, we will use triangles to approximate our disc. The resolution will therefore define how many of these triangles we would like to render. The more triangles we have, the closer to a real circle it will look, but it will require more and more computations, leading to performance issues.

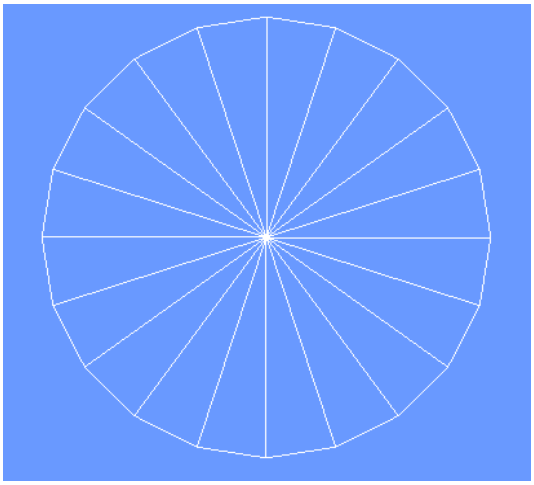

Figure 5: Wireframe representation of a disc

As stated above, to calculate the vertices of each triangle, we will start from the centre (the shape's position), and use the equation of a circle to calculate the remaining two vertices to form a complete triangle. This is done by incrementing the angle on the circle for each vertex. An important step is that once we have gone round the circle, we need to make sure our last vertices match the first ones, otherwise it might overlap or even show a little bit of space between the triangles. This is due to the floating point numbers limitation. In short, they have a limited precision, and some of it is lost on each iteration. This loss of precision accumulates to the point that our last vertices do not automatically match the first ones as they should. This is also the case for spheres and cylinders.

Texturing the disc is quite straight forward, we just need to match the vertex positions on the circle to the same values of UVs, using the formulas:

$$u = cos(theta) / diameter + 0.5$$

$$v = sin(theta) / diameter + 0.5$$

Note that we add 0.5 on both situations, because the centre of our disc in the texture file is on the UV coordinates (0.5, 0.5).

The normals are all perpendicular to the shape, for all vertices.

3. Cylinder

Generating a cylinder follows a simple idea, but it turns out to be much more difficult to implement. The idea is that the cylinder contains two or many discs, linked together and built on top of each other. I achieved this by generating each disc one after another, incrementing the Y position (the height) for each disc. I then also generated an array of indices, since generating the discs in this way did not provide vertices in the correct order. The indices will state that we need to use two vertices from the upper disc and two vertices from the disc below in order to from a quad.

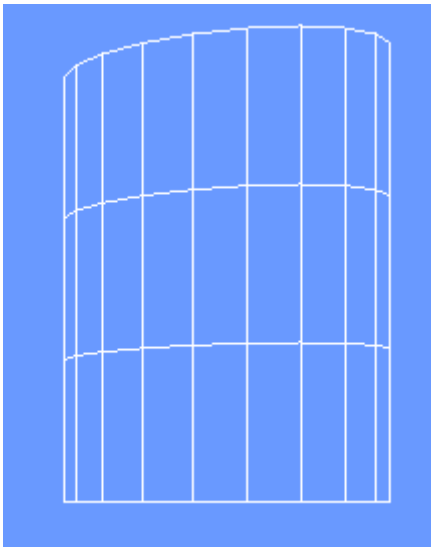

Figure 6: Wireframe representation of a cylinder

I also introduced another variable called *stackResolution*, which defines how many stacks of discs we would like to have in our cylinder. For now, increasing this value will also increase the height of the cylinder, but it will be better to have the height of the cylinder separated for more complex applications.

Texturing the cylinder is done by wrapping the texture around the shape. We will therefore clearly see where an anomaly if the texture in not tile-able (wasn't made to repeat), so we need to be careful about that.

Each normal has the same coordinate as its vertex, considering it is a unit cylinder. Otherwise we simply divide each coordinate by the radius.

4. Sphere

Generating a sphere is quite easy. It requires the same informations as for a disc, so my sphere class inherits from the disc class. We then use the parametric equation of a sphere to calculate all the vertices, in order, with the help of two loops that will track the longitude and latitude angles of our sphere. In essence, this is like building a 2D grid as for a globe made of latitudes and longitudes. I provided a simple pseudocode below, where instead of rendering the face we store it inside our vertex arrays.

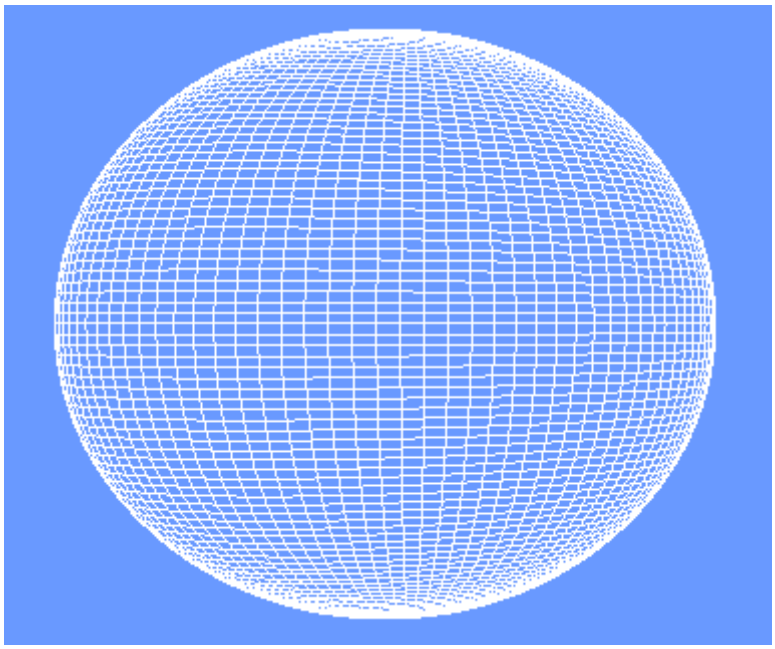

Figure 7: Wireframe representation of a sphere

```
THETA = 2 * PI / NSEGMENTS
GAMMA = PI / NSEGMENTS
LOOP LONGITUTDE:
    LOOP LATITUDE:
        CALCULATE X,Y,Z FOR 4 VERTICES (A QUAD)
            -> [LAT][LONG], [LAT+1][LONG], [LAT+1][LONG+1], [LAT][LONG+1]
        RENDER FACE
        INCREMENT LATITUDE
    END
    INCREMENT LONGITUDE
END
```

Figure 8: Pseudocode of the sphere generation

We must decide how many divisions of longitudes and latitudes we would like the shape to have, so we will be using the *resolution* variable for that.

Texturing a sphere and calculating its normals follow the same principles as the cylinder.

5. Skybox

The illusion of a sky can be done quite easily using a simple box or sphere, inside of which we find our camera. The idea is to have this skybox moving with the camera, by making its centre to have the same position as the camera's position.

In order for the illusion to work, we need to render this box first, before rendering any other object. We also want to disable any depth sorting to make sure every bits of the box is rendered, and re-enable it afterwards. We can then draw our objects, giving the illusion that they exist within this sky environment.

6. Light class

Lights can be of four different types: ambient, diffuse, point or spotlights. I therefore made a simple light class, which handles the creation and the rendering of each light objects that I initialise. I also make use of materials to have better lighting effects, even though it was not required for this coursework.

In my scene I used one spotlight, one point light and an ambient light. The spotlight is the main source of lighting in the scene, and it is combined with the ambient light, as it goes in pair with another light source (otherwise we would observe strange lighting behaviours). I made it randomly flicker to show how the spotlight influences the scene. The point light was mainly added to highlight a correct lighting on the models in the scene, so I also made it rotate about the centre of the scene and made it blue to achieve this purpose.

This is where all the normals for all of our shapes will be required, as the angle made between a normal and the direction from the light source to the associated vertex determines illumination.

7. Model loading

Models come in many different formats. For this coursework, we were provided a model loader, which loads a model in a OBJ format, considering the model was only made of triangles. I modified it a little bit to also render models that consist of quads, though it will not work if it contains a mix of triangles and quads. Once the model data is loaded, we need to process it to store it inside our vertex arrays, in order. Luckily, the OBJ format lists all the faces with their vertices, normals and texture coordinates, so we can just scan from there.

For the main lamp model, I had to separate the neons from the structure, in order to apply a material that makes it appear as a light source. Note that the material changes if the spotlight is off.

8. Texture filtering

If you look closely on some of my shapes, you will find strange black lines appearing on specific regions. This is due to the texturing filtering method, or more specifically how it is sampling the texture. I would suggest to change the filtering method using the corresponding key and see the black lines disappear with some of them.

Basically, it happens because the method may sample outside the texture on the edges, resulting on sampling black values and thus black lines.

Figure 9: Example of the black line appearing on a sphere with the Trillinear filtering method

9. Depth sorting on transparent shapes

Rendering transparent shapes is quite tricky. In fact, the depth-sorting algorithm provided by OpenGL is insufficient in this case. The rendering order of each shape matters: we need to render the furthest transparent shape first and the closest last. Otherwise, we will observe that transparent shapes behind another are partially or totally hidden, which is not the desired effect.

Figure 10: Example with and without sorting the transparent shapes

To fix this issue, we can put them all into a vector and sort them each frame based on the distance between their position and the camera position. The furthest transparent object will then be the first item on the vector, and the closest item will be the last, thus solving our issue.

Thankfully, since all our shapes now inherited from the BasicShape class, and therefore they all have a position information stored, we can make a vector of BasicShapes to contain all of our transparent shapes. All that's left is to sort it, and the C++ std::sort standard algorithm let us do that quite nicely, by letting us define how to sort our objects. A simple lambda function will do the trick, as it allows to also capture the camera object.

- Stencil buffer

In my scene I implemented a mirror. As little as it may seems, this is actually a quite good puzzle to solve in order to obtain a correct reflection. The way it works is by using the stencil buffer. It is a buffer that will hold certain values depending on what we tell him to draw. For instance, I initialise my stencil buffer with zeros, so we can think of it as an array full of zeros. To draw the limits of my mirror, I draw my mirror quad into the stencil buffer with ones.

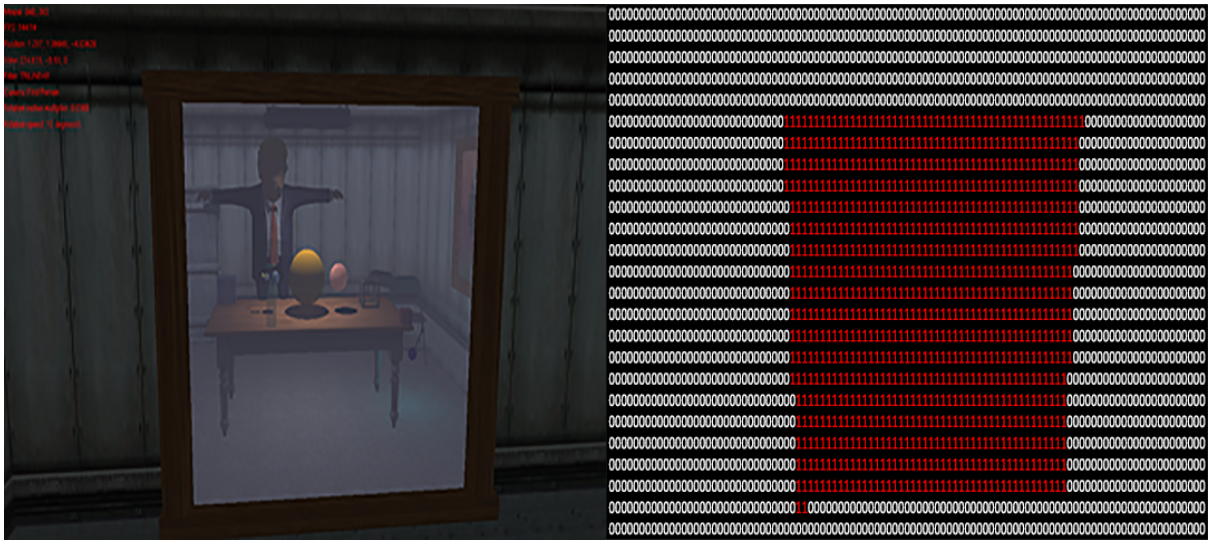

Figure 11: Representation of the stencil buffer values set by the mirror quad

Now that my stencil buffer is initialised, I can translate my entire room, reverse scale it so that it appears mirrored, make sure to draw the back face instead of the front face (because of the scaling) and draw my entire scene using a stencil test. The stencil test verifies that what we attempt to draw is within the region defined by our mirror quad, hence whether the pixel location has a value of 1 for the same stencil location. If the stencil buffer has a value of 0, the pixel is discarded. And we obtain a very nice looking mirror! The drawback, however, is that the scene is now being rendered twice, which can be quite computationally expensive. Note that the transparent shapes inside the mirror do not follow any sorting, as they will always be in the same order (we cannot go around them in the reflection world like in the real world). I therefore hard coded their rendering order when the reflection is being drawn.

- Shadows

Creating shadows can be done using several techniques, the one is used is called “Planar shadows”. They are created applying a specific transform which flattens a shape’s geometry into a defined plane. The drawback of this technique is that it will be difficult to render the shadow on curved surfaces, as we can see on the curved edges of the table in my application.

As stated above, we need to define a plane to which the shadow should appear. It can be a wall, a floor, or any flat surface. I chose to make an invisible quad of roughly the same size as the table’s surface, just a little bit above it, so that there would not be any Z-fighting (the renderer cannot decide if it needs to render to shadow or the table since they have the same position in space). With the shadow class provided for the coursework, it is then just a matter of creating the correct transform matrix in regards of the spotlight, and rendering all the objects we would like to cast shadows. I also made sure that the shadow’s colour is influenced by the transparency of the object casting it, whereas a transparent object will not cast a shadow as dark as a non-transparent object.

I also added a simple stencil test, that limits the shadows to be displayed within the invisible quad so that it does not go in the air, outside the table surface. I big challenge however, that was quite hard to figure out, was how to draw the shadows inside the mirror, since it was already using the stencil buffer. The way I solved it was by having a separate stencil function and initialisation when drawing into the reflected world. In the real world, I can just reset the stencil buffer completely, since it will never be mixed up with the mirror, and write ones into it with the invisible shadow surface quad, test for ones and bingo. In the reflected world however, we can’t reset the stencil buffer, because otherwise the shadows and the transparent shapes would be permanently displayed behind the mirror and never be hidden, even if the mirror is not visible. Instead, we can increment the non-zero values inside the stencil buffer by one for every pixel of the invisible shadow quad. We can then test for any non-zero values to draw everything correctly, but the shadows can only be drawn if the stencil value is 2.

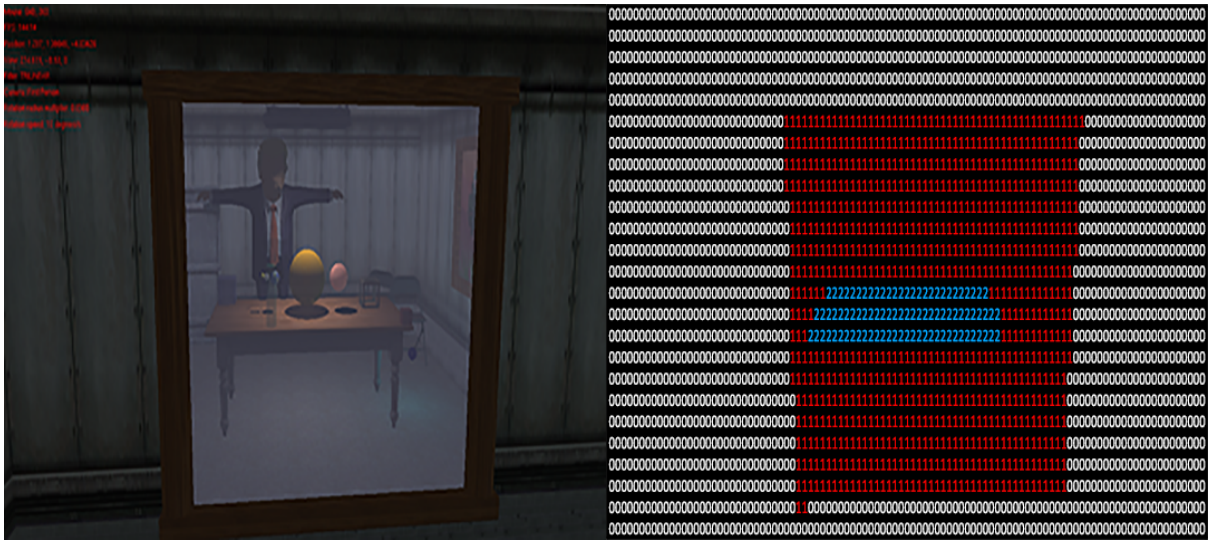

Figure 12: Representation of the invisible shadow quad incrementing the stencil values in the mirror

And voila, we obtain good looking shadows!

Conclusion

In this project I have demonstrated my knowledge of OpenGL, from generating simple shapes to loading complex models, as well as their correct texturing and lighting. The only thing I would have liked to add is a shadow volume for the table. It would have been a very great effect and add a little more realism to my scene. Definitely a great thing to do in the future! Another great feature to explore would be a map loader, as for now making the room walls and décor hard coded seems quite inefficient and difficult to modify or build. So having the ability to load a map from a level design application, such as Quake3's radiant tool, would be amazing! Overall, I am very satisfied with my project and hope to learn more in the coming semester!

References

- Images / Textures:

  NH'AlterEgo. (2018) *Skybox2 texture*.

  Unknown author. (unknown date) *Earth diffuse texture*. Available at: http://www.softwaresamurai.org/wp-content/uploads/2017/12/earth-diffuse.jpg (Accessed: 16 December 2020)

  Unknown author. (unknown date) *Surface of the moon texture*. Available at: https://st.depositphotos.com/1017411/1565/i/950/depositphotos_15659203-stock-photo-seamless-texture-surface-of-the.jpg (Accessed: 16 December 2020)

  Unknown author. (unknown date) *Seamless transparent glass texture*. Available at: https://3.bp.blogspot.com/-QUJUQeD-vx8/WHYDXkA9NZI/AAAAAAAABKg/uuRfmQ7uWdwXsE87_qzuGZmpET6EJxfHQCLcB/s1600/Transparent%2Bglass%2Bseamless%2Btexture%2B1.png (Accessed: 16 December 2020)

  Unknown author. (unknown date) *Mars texture*. Available at: https://opengameart.org/sites/default/files/Mars%20CH16.png (Accessed: 16 December 2020)

  Unknown author. (unknown date) *Lime2*. Available at: https://download.tuxfamily.org/blenderarchi/images/Lime2.png (Accessed: 16 December 2020)

  Unknown author. (unknown date) *Tall grass texture*. Available at: https://www.freeiconspng.com/uploads/tall-grass-png-images-pictures-12.png (Accessed: 16 December 2020)

  Valve, Halflife. (1998) *-0OUT_WALL3*. Available in the original game files.

  Valve, Halflife. (1998) *-0TNNL_FLR1B*. Available in the original game files.

  Valve, Halflife. (1998) *-1LAB1_W4GAD*. Available in the original game files.

  Valve, Halflife. (1998) *-1LAB3_FLR1B*. Available in the original game files.

  Valve, Halflife. (1998) *-2FREEZER_PAN2*. Available in the original game files.

  Valve, Halflife. (1998) *GENERIC_113C*. Available in the original game files.

  Valve, Halflife. (1998) *GLASS_BRIGHT*. Available in the original game files.

  Unknown author. (2017) *This high-definition photo of Steve Harvey*. Available at: https://imgur.com/oJHHtwy (Accessed: 17 December 2020)

- Models:

Printable_Models. (2017) *13494_Folding_Chairs_v1_L3*. Available at: https://free3d.com/3d-model/folding-chairs-v1--612720.html (Accessed: 13 December 2020)

Printable_Models. (2018) *10233_Kitchen_Table_v2_L3*. Available at: https://free3d.com/3d-model/-kitchen-table-v2--196000.html (Accessed: 15 December 2020)

Trippsmith05. (2018) *HangingLight*. Available at: https://free3d.com/3d-model/hanginglight-759165.html (Accessed: 15 December 2020)

George Washington. (2017) *Untitled Donald Trump*. Available at: https://clara.io/view/234eb72e-0f07-4818-8e22-d17533db641c (Accessed: 16 December 2020)