Firefly Disco Final Report!

Team members

Sumedha Bhangale (bsumedha)
Dexter Langman (dlangman)
Wojciech Truty (wtruty)

Figure 1: The final scene.

Link to video: https://youtu.be/dcOdTya5L-A
Backup link to video: http://www.tekshome.com/cs148/Presentation.mp4

Brief overview

We put in a lot of time and effort, but It turned out beautifully! We managed to make a dynamic scene with many, moving fireflies. Objects in the scene move in response to music, and the brightness of the light sources also varies with the music. Compared to our original proposal, we didn't add too many static objects, as that would've been more repetition of principles we'd already demonstrated. We also don't have the fireflies directly interact with objects in the scene – collision detection would've taken more time.

Division of labor

- Sumedha Bhangale
  - Deferred shading algorithm for lighting.
  - Create static, textured table.
- Dexter Langman
  - Flocking algorithm on fireflies.
  - Visual representation of fireflies.
- Wojciech Truty
  - FFT Processing.
  - FFT data application to dynamic objects.
  - Background skybox setup.

Dexter Langman

Flocking algorithm

I used the flocking algorithm description on Wikipedia[1] as a basis. At a high level, each firefly tries to approach the center of the swarm, align with neighbors, and back off from neighbors if they get too close. I also make sure the direction doesn't change too rapidly or else they would look jittery.
My implementation went very smoothly hitting very few snags. All pieces are weighted, and the equation looks something like this:

Next direction = current direction + alignment + attraction to center + avoid collision + little randomness

All the parameters are tunable, so I can adjust the swarm's density, speed, number of fireflies, etc. For example, if I put more weight on the alignment factor, the whole swarm starts to look like it's moving in unison, wobbling around the center. As is, it should more like a cluster with occasional outliers.
Part of the algorithm is for each individual to flee from neighbors that get too close to try to avoid collision. My first implementation looked too rigid, with fireflies appearing to bounce sharply away from others. To make the flock look smoother and more natural, I made the repulsion factor exponential based on how close the cubes are to each other. While in the attached video you can see that this does allow cubes to graze and pass through each, when we increase the average distances in the final product, no two fireflies should ever get too close.

Firefly appearance

The fireflies are each represented by a triangle fan with a "billboarding" effect as described in lecture, always rotating to face the viewer. This was tricky for me to get right, despite the matrix transformations being fairly simple, but Wojciech helped point me in the right direction.

Each firefly has a variable alpha value, solid at the center and zero at the perimeter. You can see this in Figure 2: Fireflies with variable opacity, distance-based size scaling., especially with the red firefly in the foreground somewhat obscuring the white firefly in the background. Each firefly's alpha is also a function of the FFT's sampling, allowing them to flicker with the music.

[1] https://en.wikipedia.org/wiki/Flocking_(behavior)

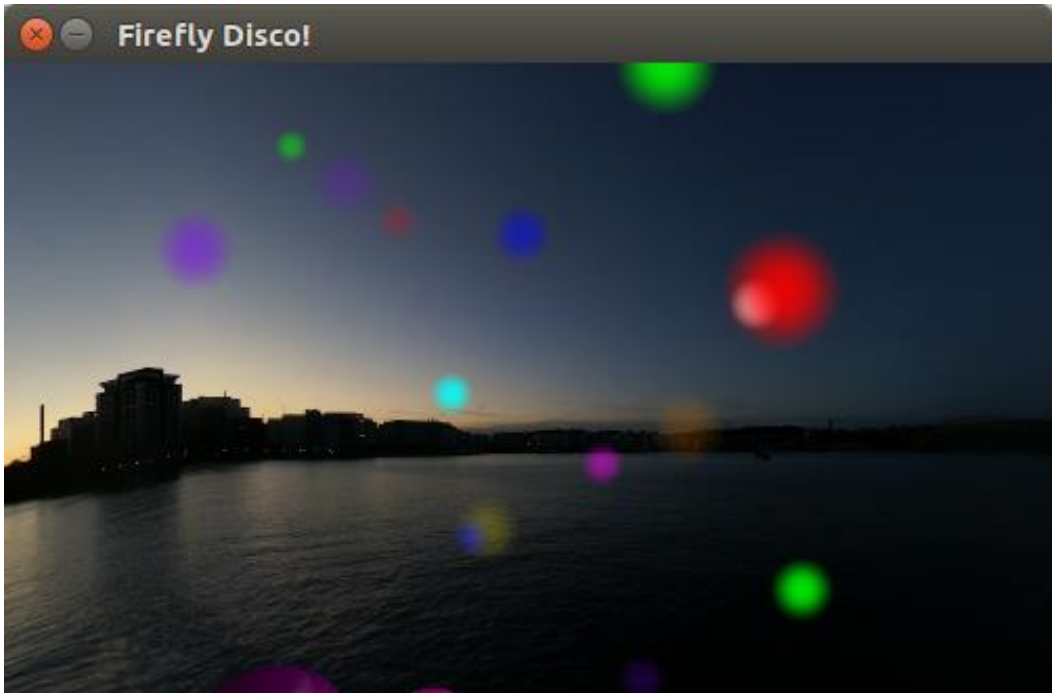

Figure 2: Fireflies with variable opacity, distance-based size scaling.

Finally, at Wojciech's suggestion, I added an extra little bit of artificial size scaling based on distance from the camera. This scaling makes the visual size difference, and thus, the apparent z-distance between the fireflies a little more obvious to help distinguish which ones are closer or further when the camera's very close. The effect needed to be very subtle, otherwise the fireflies become unnaturally larger as the camera approaches them. Overall, I think it turned out really nicely. Admittedly, it's almost unnoticeable unless the camera's moving, but that said, it would be jarring if it were noticeable.

Wojciech Truty

FFT-Driven Shape Generation

My task was to generate the shapes which will respond to the FFT stimulus. The objects are a flat plane, which we referred to as paper in our proposal, as well as several spheres.

Audio I/O and FFT

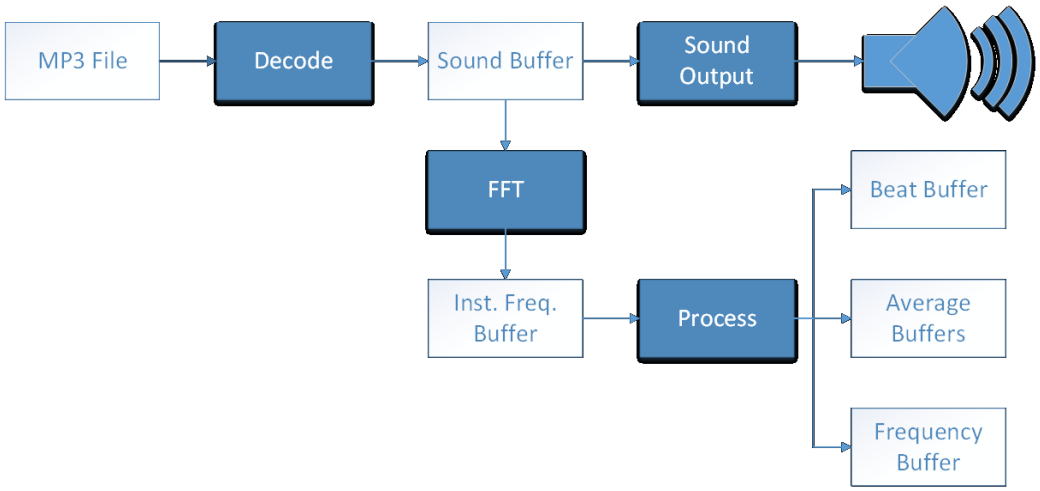

Figure 3: Data flow diagram of the audio processing step

Figure 3 shows the flow diagram of our audio processing implementation. Audio files are decoded using libAV into a raw sound buffer[2]. The buffer length is 26ms for an 44.1kHz mp3. The sound buffer is fed directly to libAO, so that it can be output to the speakers. In parallel, we take 512 samples from the sound buffer, pass it through a Hanning Window[3], and then feed it to the KissFFT library[4].

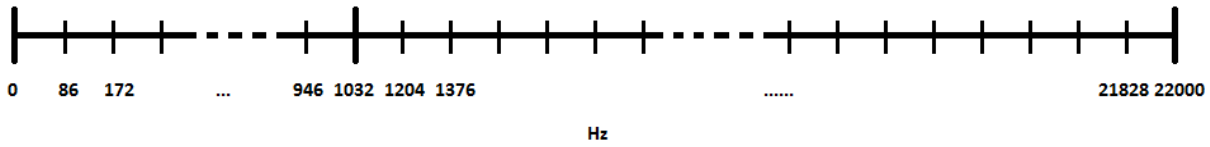

Figure 4: Compressing frequency bins above 1kHz to show better detail of lower frequencies

The output of KissFFT gives me 256 samples of audible frequency spectrum, evenly spread at resolution of 86Hz (given 44.1kHz input audio). For better visuals, it would have been nice to make the frequency scale a log scale, but I didn't want waste compute cycles on the math, so I just did a simple hack. The final output from my FFT program feeds first 12 frequency bins (0-1kHz) directly, and then compresses the remaining 21Khz by feeding two FFT bins into each output bin. We start with 256 bins, but end up with 134 bins. That way we magnify the first 1kHz for better visuals. This is demonstrated by Figure 4. We save the FFT into a 128 entry buffer, thus giving us around three seconds worth of historical FFTs.

I do simple beat detection[5] by looking at the 86kHz bin, and comparing the instantaneous magnitude to the average of the last 40 magnitudes. It is not perfect but works on some sound samples with decent results. I also compute average of instantaneous magnitudes at different parts of the spectrum to feed them as input into other visuals, for example as lighting intensity for the fireflies. To prevent the audio from stuttering, all of the audio processing runs in a separate thread than the visual processing and rendering.

Spectrum Visualization

I created a flat mesh composed of 128x128 vertices. I save the 134x128-entry FFT buffer into a 136x130 height map, and between each draw call I shift it back by one. In my shader I access the height map for each vertex to determine the "y" component. I also generate the normal for each vertex on the fly. As shown in Figure 5, I average out the normals of the 6 triangles adjacent to the vertex. Side vertices are handled easily, since the height map is bigger by one vertex on each edge and we set these extra vertices to a height of zero. The texture map is also configured to loop, so outside edges will always have a height of 0, and normal are computed correctly.

[2] https://socapex.wordpress.com/2015/04/11/libav-libao-and-qt5-audio-player-tutorial/
[3] http://stackoverflow.com/a/3555393
[4] https://sourceforge.net/projects/kissfft/
[5] https://en.wikipedia.org/wiki/Beat_detection

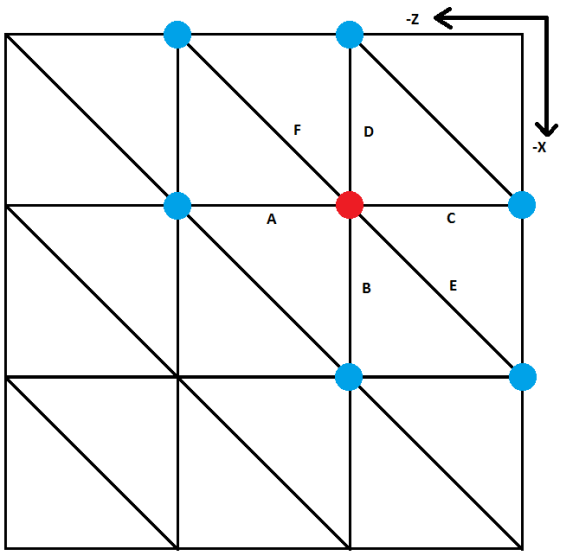

Figure 5: Flat paper mesh

I color the mesh by setting the point Y=1 to pure red with a small amount of blue, and Y=0 to mostly pure green with some blue component. That way we get a nice visual of the intensity of the sound. The blue component is needed because during the deferred lighting phase, if a pure blue firefly light is shining on the object, we want it to appear blue. Without the blue component, a purely blue firefly would make no contribution to the diffused lighting of the object (I found that out only after we added colored light sources).

One additional visual improvement that I needed to make was to smooth out some jagged pieces, so I used a Gaussian filter on my height map. Figure 6 shows the final state of the mesh in our project.

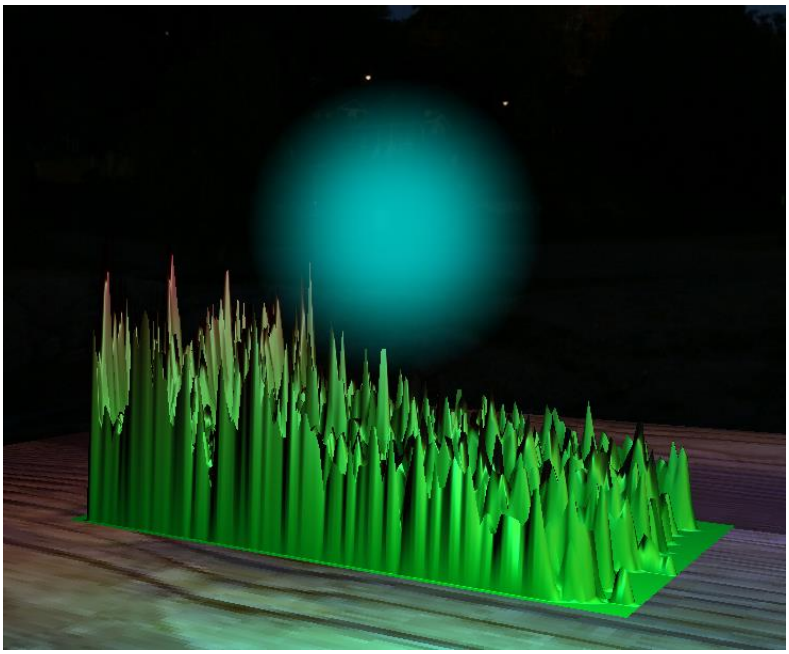

Figure 6: FFT visualization with light-blue firefly illuminating it

I also have two spheres which use different visuals to show aspects of the spectrum. One of the spheres (Figure 7) is fed by the beat detector, and on each beat it bounces. I use the underdamped

harmonic oscillation equation in my shader to generate the bounce[6]. When a beat is detected I reset the time variable to zero, and increment it until we stop bouncing. Also, when it crosses below the Y=0 point, I deform the sphere, as if it were a rubber ball.

One difficulty with the bounce was to make it slow enough to look natural, yet dampen fast enough so that we finish bouncing by the time the next beat arrives. Originally, I also used a cosine function for the oscillation function, which made the ball suddenly jump from resting position to maximum height on a bounce (when time reset to zero). This did not look natural. Instead I changed the cosine function to a sine function (since sin(0) = 0), so that the ball gradually makes it to the high point.

The second sphere (Figure 8) is deformed by a sinusoidal function whose magnitude is based on the average instantaneous magnitude at 11kHz-12kHz range. Its color also changes on each beat.

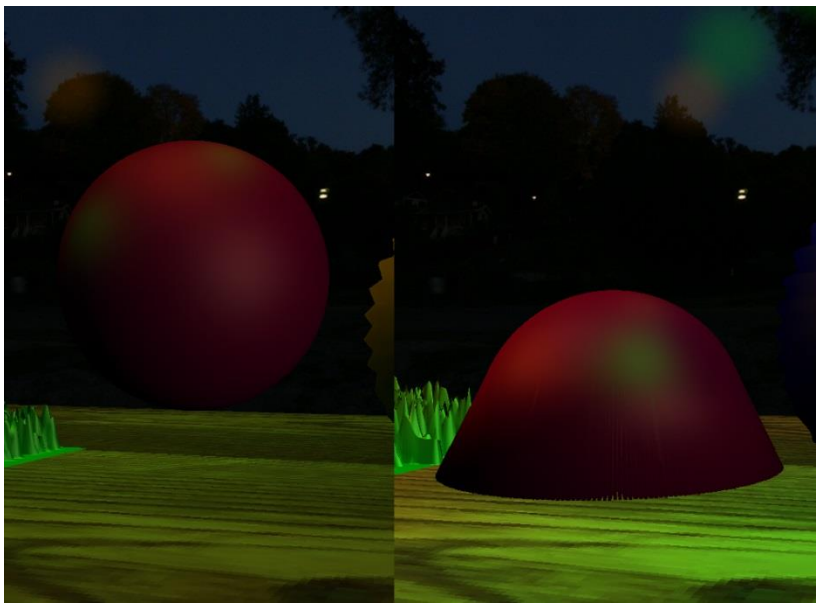

Figure 7: Sphere bouncing to the beat

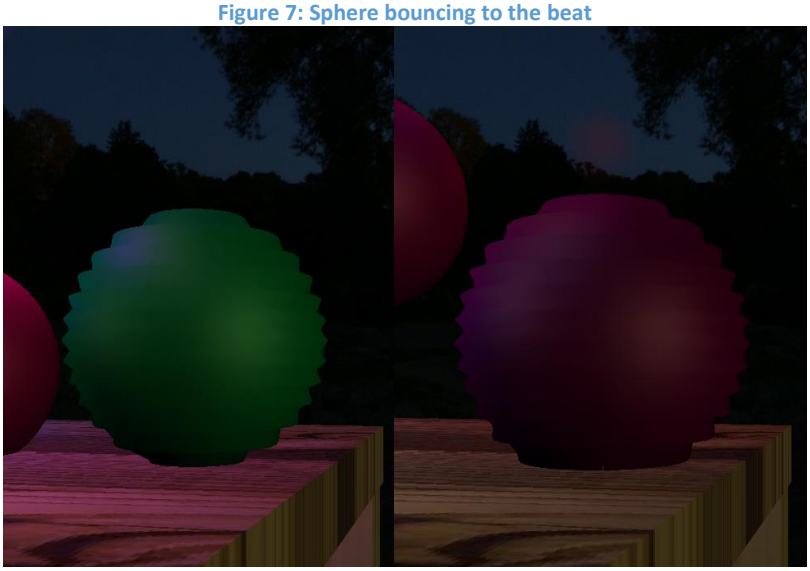

Figure 8: Deforming and color-changing sphere

[6] http://hyperphysics.phy-astr.gsu.edu/hbase/oscda.html

Background Visualization

To place our objects in an evening setting, I found a freely available[7] evening cube-map texture. I created a cube (skybox) which encompassed all of the objects, and applied the texture to it as a GL_TEXTURE_CUBE_MAP type. The main challenge for me was to separate the texture from deferred shading/lighting since we did not want the fireflies affecting the look of the skybox. With Sumedha's help, we forward-rendered the skybox. The texture contained a sunset scene, so we used the position of the sunset as an additional light source, to cast light on our objects.

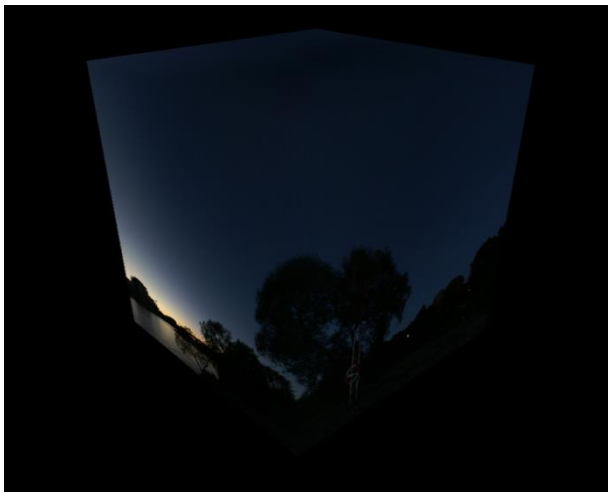

Figure 9: Our skybox

Sumedha Bhangale

Deferred shading

Our scene consisted of numerous light sources in the form of fireflies. Also there were a large number of vertices, hence a large number of fragments from the animated objects. We decided to do deferred shading on the table and objects to reduce the overall light computation for the scene on the GPU. The fireflies and the skybox are forward rendered.

Implementation details:

Deferred shading is split into two shading passes[8]:

- Geometry pass

  In this pass, I perform vertex shading in the regular manner, calculating the position, normal and texture co-ordinates of the vertex to pass on to the fragment shader. It is the fragment shader that is different in this pass. Instead of calculating the color that is to be rendered on the screen for the fragment, I store all the information from the vertex shader as well as the color/texture information for the fragment as 2D texture.

- Lighting pass

  In the lighting pass we render a 2D quadrilateral of the size of the window to fill the window completely. The 2D textures stored in geometry pass are applied to this quadrilateral. For the

[7] http://opengameart.org/content/night-skyboxes

[8] http://learnopengl.com/#!Advanced-Lighting/Deferred-Shading

lighting calculations, the position, color and light intensity values of the fireflies are given to the fragment shader. The color for each pixel is calculated as the sum of the color from each firefly using Phong shading.

In order to store the output of the geometry pass we use Framebuffer Object[9] provided by OpenGL Objects. This is separate from the default framebuffer of the GPU and can be used to store data in the intermediate stage which won't be rendered on the screen.

We store the geometry information from the geometry pass in 3 2D textures: position, normal and color + specular intensity. The third texture stores color in 'rgb' components and the specular intensity value in the 'a' component. These textures are added to the FBO as GL_COLOR_ATTACHMENTs. The glDrawBuffers function then instructs the GPU to store the output of the fragment in these buffers inside this FBO.

It is needed to store the depth data of the scene too. This is done by using the Renderbuffer object[10] for the OpenGL Objects. We attach this object to the FBO as GL_DEPTH_ATTACHMENT and store the depth data in it.

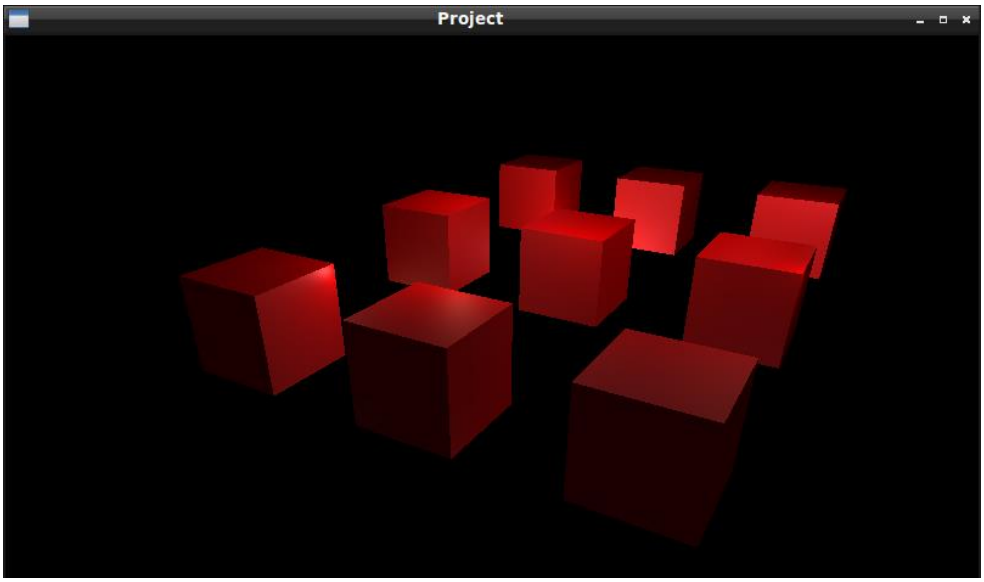

Figure 10: Milestone 1 - deferred shading with 32 light sources.

Challenges faced:

The main challenge we faced was while putting it all together. Since the fireflies and the skybox were forward rendered and the table with the objects was deferred shaded on a window filled 2D quadrilateral, the fireflies and skybox were obscured behind the 2D quadrilateral. In order to mingle the fireflies with the table and the objects we needed to do depth testing. I implemented bit blit[11] and then enabled depth testing to manage this. The depth information from the framebuffer object is read and drawn into the GPU's default framebuffer. Then I enable the depth test before forward rendering the fireflies and the skybox.

[9] https://www.opengl.org/wiki/Framebuffer_Object
[10] https://www.opengl.org/wiki/Renderbuffer_Object
[11] https://en.wikipedia.org/wiki/Bit_blit

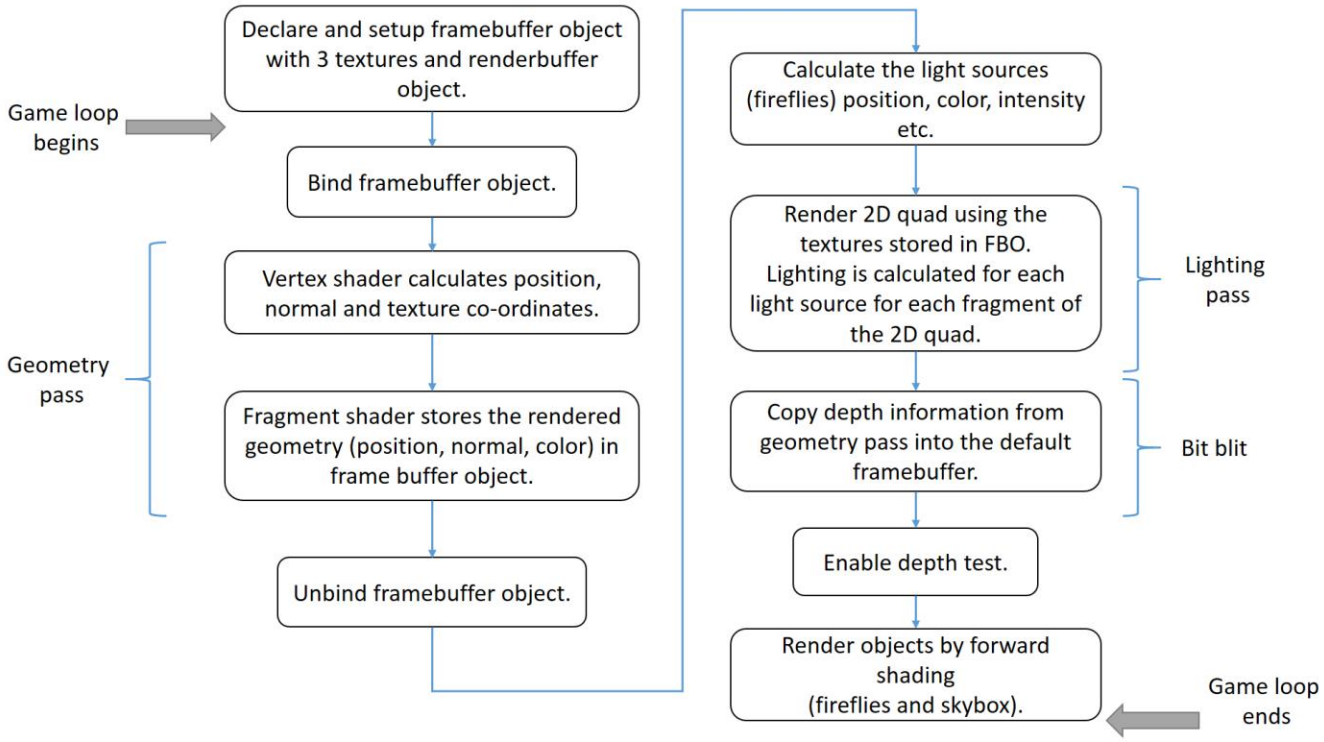

Figure 11: Deferred shading flowchart.

Table generation

I generated the table in blender in order to get a .obj file that could be easily imported in our project. The table consists of 5 cubes scales and translated appropriately. Finally, a 2D wooden texture is applied to it to.

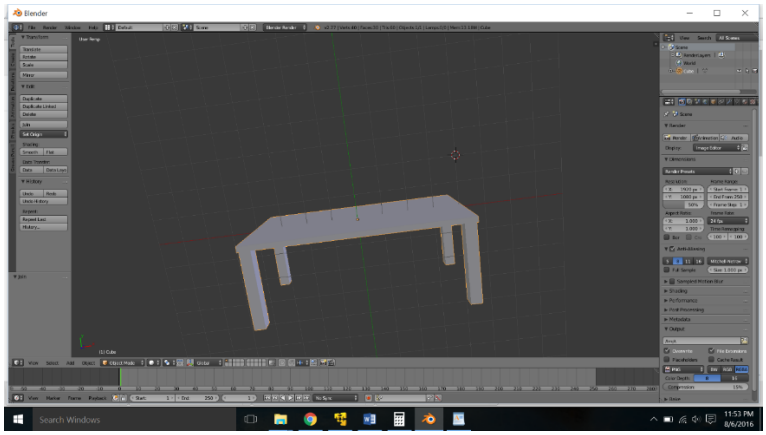

Figure 12: Table construction in blender.

I also generated the spheres in blender to get an easy to import .obj file in which the sphere is made of triangle strip.

There was a minor hiccup in the generation of .obj file. Blender produces the .obj file such that the depth information is stored along Y axis while OpenGL uses Z axis. So I had to set an option while exporting the .obj file to have Z axis as depth buffer.