Cosy Campfire

Group Members:

Ryan Holloway 300570174 ryanholloway123@gmail.com
Jamie Gambles 300475743 jamiegambles01@gmail.com
Aidan Edward 300577115 edwardaida@myvuw.ac.nz
Sam Dunnachie 300533979 dunnacsamu@myvuw.ac.nz

Abstract:

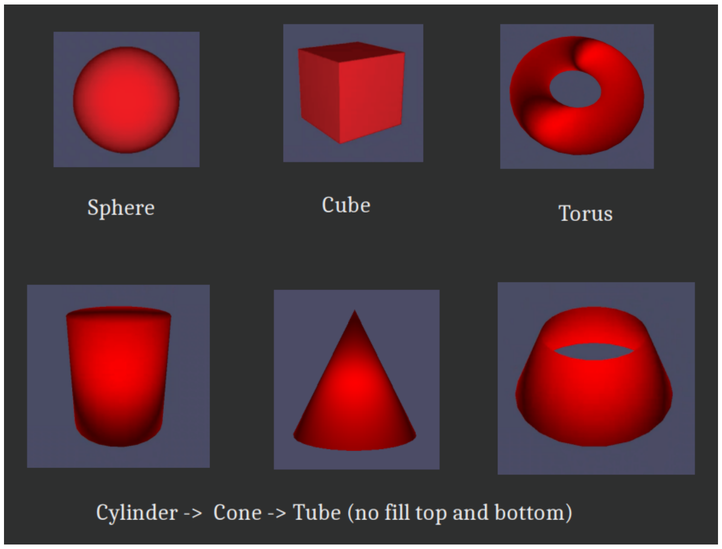
Our goal for this project was to create a ‘Cosy Campfire’ scene. For this, we needed to create a fire, an environment, objects to place around the fire in the environment, and any extra details we could think of to help the aesthetic, such as fireflies. My role was to provide the modeling for the scene. In our project planning discussions, we considered the possibility of many different objects that we could place in the scene. Coming from an artist background, I wanted the ability to create more than just one object, so I was investigating what solutions might make it possible to create these various objects, (trees, logs, tables, chairs, fire guard encircling the fire, etc). After researching various solutions, I came to the conclusion that the best way to create these various objects would be to create a system of parametric modeling tools capable of driving parameter values from the user’s input in the GUI.

Group Responsibilities:

Aidan Edward: Fire Simulation
Ryan Holloway: Parametric Modeling and GUI
Jamie Gambles: Texturing (parallax occlusion mapping)
Sam Dunnachie: Firefly Simulation (boids), additionally attempted soft shadows

Parametric Modeling:

Based on the way modern DCC modeling software typically works, there are several primitive shapes that are used as a starting point to generate all models; e.g. cube, sphere, cylinder, cone, torus. In order to create a parametric modeling system capable of generating numerous objects, I knew I would first need to create functions that generate these primitive shapes; all with the ability to scale, change position, and rotate along each axis, x, y, and z. This required first building the meshes of the primitive objects by defining their vertices and indices to draw the shapes. Second, I needed to find a solution to scale, rotate, and translate each of these objects. I defined a ‘centroid’ at the center of each primitive shape in order to apply translation to each object. Scale was handled through the generation of the mesh by controlling parameters that drove the height, width, depth, radius, etc. to all shapes where these parameters were relevant. Rotation was implemented through applying a rotation matrix after all the vertices and indices had been defined. Translating the objects back to the origin before applying the rotation matrix, then translating the objects back to their location after the rotation matrix enables the objects to rotate freely around the object’s origin, regardless of where it is located in the world’s space.



Once these primitive shape functions had been created, I could then move onto creating algorithms for functions to generate parent objects, such as the objects we hypothesized to place in our scene, e.g. chair, table, tree, logs, fire guard, etc. These parent object parameters are what drive the parameters to all the child (primitive) objects within the parent object's function. Each parent object presented its own challenges and creativity for solving the geometric mathematical algorithm for generating a parametrically designed object that could dynamically be altered by the user's input from within the GUI, which alters the parameters that generate each particular object.

Challenges:

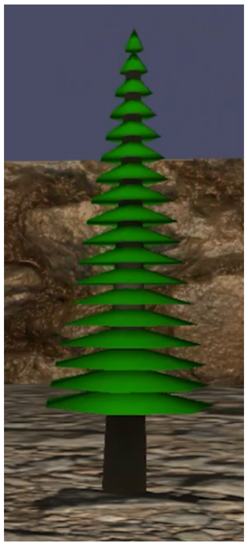
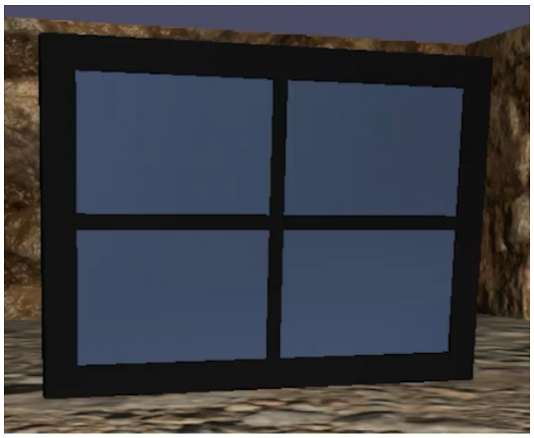
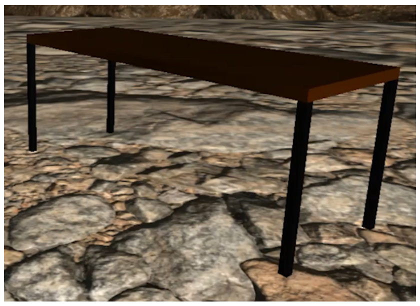
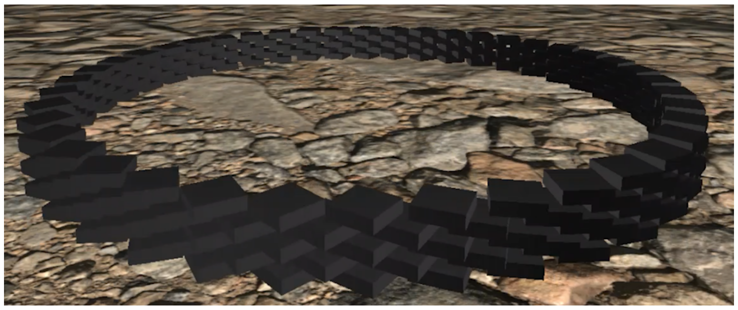
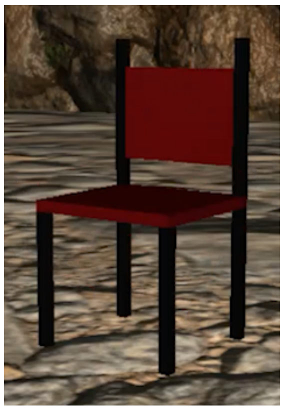
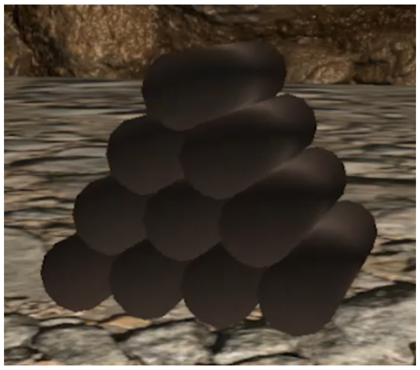
A challenge for me was to find a solution to render a dynamic number of objects simultaneously that could be continually updated from the GUI. The solution to this problem required generating a basic_model object pointer, allocated on the heap, for each primitive object used within every parent object's function. Every call on a primitive object is used to generate the gl_mesh of each basic_model object. Then the color and shader are explicitly defined, allowing for specific materials and colors of individual pieces of the parent objects to be hard coded within each parent object's function definition. Each of these primitive object basic_model pointers were stored in a unique vector for each parent object. Having each parent object contained in their own vector made it possible to call one parent object at a time in the rendering process, so the user can choose which objects to place in the scene or not. To render each parent object, I simply iterate through every basic_model (primitive) object vector and call draw on that current object.

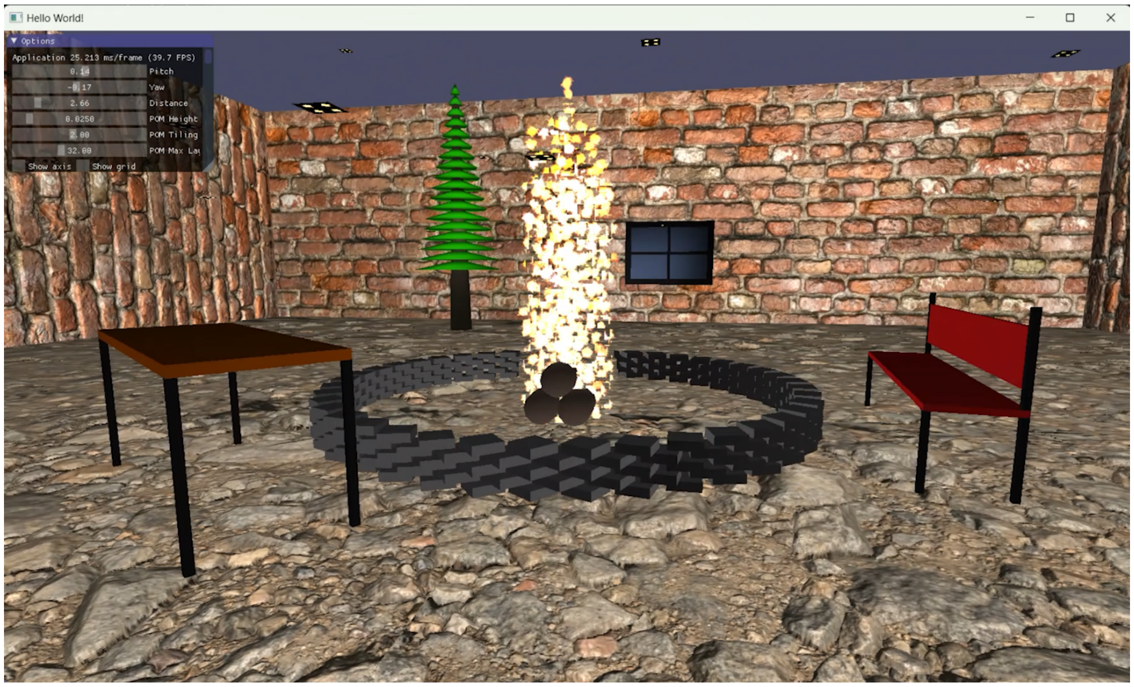
Another challenge was deleting the objects properly so the buffer would update as intended. I handled this in the IMGUI section of the code, so when any changes are made to the object from the GUI that respective parent object's vector of basic_model pointers is deleted and cleared before calling the updated version of the parent object with its new parameter values. This makes it so the user can interactively alter various attributes of the parent object by simply using the GUI sliders and seeing the changes happen immediately within the viewport.

Once the parent objects were rendering properly, I had an issue when trying to rotate a parent object. Instead of the entire parent object rotating, each individual child object would rotate around its own respective centroid, which made sense as the rotation was cascading down from the parent object to each respective child object's rotation. The solution I found to this problem was to create another set of rotation parameters ('rotation_2') for each primitive object. This rotation omitted the process of translating the objects back to the origin before applying the rotation matrix, then translating back to their previous location. Omitting the translation before and after the rotation enabled the objects to rotate as one entire parent object, with the caveat that the rotation of the object was centered around the world space location at (0, 0, 0).

Results:

I successfully created the following parent object generator tools; chair, table, tree, log pile, fire guard, window. It took me a while to create the entire parametric system of primitive objects that are used to generate larger parent objects, but now that the system is built there is an infinite number of possible objects that could be created.



```
Application 25.213 ms/frame (39.7 FPS)
0.14      Pitch
-0.17     Yaw
2.66      Distance
0.0250    POM Height
2.00      POM Tiling
32.00     POM Max Lay
Show axis   Show grid
```

Limitations:
One of the main limitations to my parametric modeling system is that each primitive object maintains symmetry over each respective axis of transformation. This could be improved by implementing a sheer transformation to the primitive objects. Additionally, various deformation techniques could potentially be used to create more unique shapes.

Integration:
We used Git to integrate our code which seemed to work well. My code did not interfere with my teammates code since all my various parameters, methods/functions had unique names. Once Jamie had set up an environment using his textures, I was able to use my parametric modeling tools to create objects and place them in the scene to create our goal of creating a 'Cosy Campfire' aesthetic we were hoping to create. Once I designed the objects and placed them in the scene accordingly, I documented the parameter values in the GUI's sliders, then hard coded those values to be the initial values for each parent object. In this way, the objects are in their intended placement and design for the scene when the program is run. Additionally, I altered the range of the sliders' values to be a reasonable range for each respective object's parameters, according to the scale of the scene.

https://www.youtube.com/watch?v=DR8vIFeESFE

CGRA 350 project video