CS4052 Graphics Assignment 5

Edward Bergman
#15335633
https://www.youtube.com/watch?v=CXykREcLhMw

Features Present:

- **Basic**:
    - 3D object/view
    - Limited user interaction (key press and camera movement)
    - Phong Illumination
    - Hierarchical Model (one-to-many)
- **Advanced:**
    - Dynamic Lighting
    - Animation (Not really advanced animation)

Quick Note:

We try to avoid showing too much code or an illustration of every point made as this would make the report longer than it already is. There are comments in each header file if further enquiry is needed.

User interaction (keyboard input):

We didn't like freeGlut's input functions as they can't deal with C++ classes so we created a wrapper **InputHandler.** Using this we could more easily handle user interaction for our purposes of Camera Control (**CameraController**) and Keyboard / Sphere Control (**KeyboardController**).

In our **CameraController** we map 'wasd' and 'eq' for moving the camera around the scene. (No rotation)

In our **KeyboardController**, we map 12 keys to a musical octave, one note each. We can also use '<' and '>' to increase/decrease the octave.

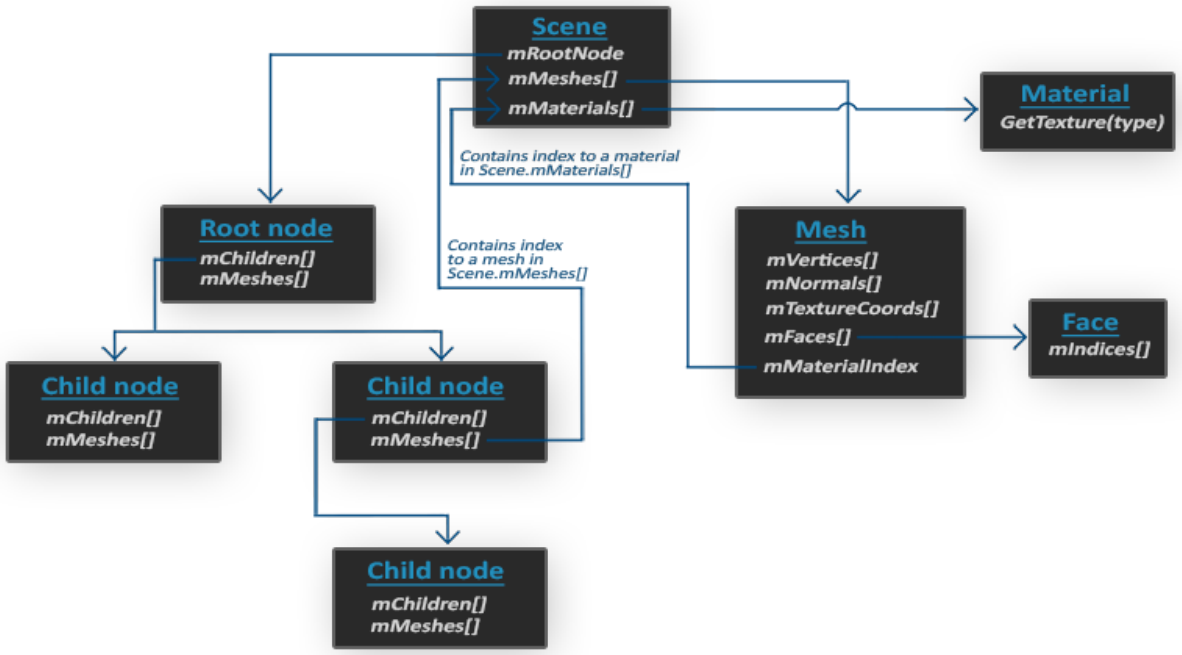
Each note corresponds to an animation on the **Keyboard** and the **Sphere** which we will see later.

Camera:

The **Camera** can be moved in all the cardinal directions + up/down. It is controlled by the **CameraController**. We have functionality for rotation and camera orientation in our **Camera** class but would make little sense in our scene.

Model Loading/Structure:

We wanted to simplify model loading so we used Assimp's Model structure and wrapped that up in our own type **Model::Object (Model.h)** which handles hierarchies and meshes.



We then use **ModelImporter** to ease importing of models. The end result was allowing us to import models as such:

```
std::unordered_map<std::string, std::string> modelNames {
 {"keyboard" , "keys.dae" },
 {"sphere" , "sphere.dae"}
};
```

```
ModelImporter::loadModels(modelNames);
keyboard = new Keyboard(ModelImporter::getModel("keyboard"));
sphere = new Sphere(ModelImporter::getModel("sphere"), lsSphere);
```

Dynamic Lighting and Shaders:

Each **Sphere Face** has a one-to-one correspondence to a note and we wanted the Face to light up to its appropriate colour for a short time as an animation, as well as not overwrite any existing lights. To allow this, we needed to be able to have multiple lights in the scene and be able to update them so they are not static. Most of this is done in the **Light** namespace for setting uniforms in the Shader based on a **Light** objects state. We can also turn lights on and off so our shader does/doesn't taking lights into account.

Our **fragment shader** has structs for the various Light types we can use (Directional, Spotlight and Pointlight). As we have multiple light sources, we create an array of structs to hold them. We also simplify calculating our entire lighting in the scene using GLSL

functions.

```
vec3 CalcDirLight(DirLight light, vec3 modelVertexNormal, vec3 viewDirection);
vec3 CalcSpotLight(SpotLight light, vec3 modelVertexNormal, vec3 modelVertexPosition, vec3 cameraDirection);
vec3 CalcPointLight(PointLight light, vec3 modelVertexNormal, vec3 modelVertexPosition, vec3 cameraDirection);
```

Note: we calculate all our lighting in Model space, this is because the tutorial we used did so and it makes more intuitive sense as to how lighting works.

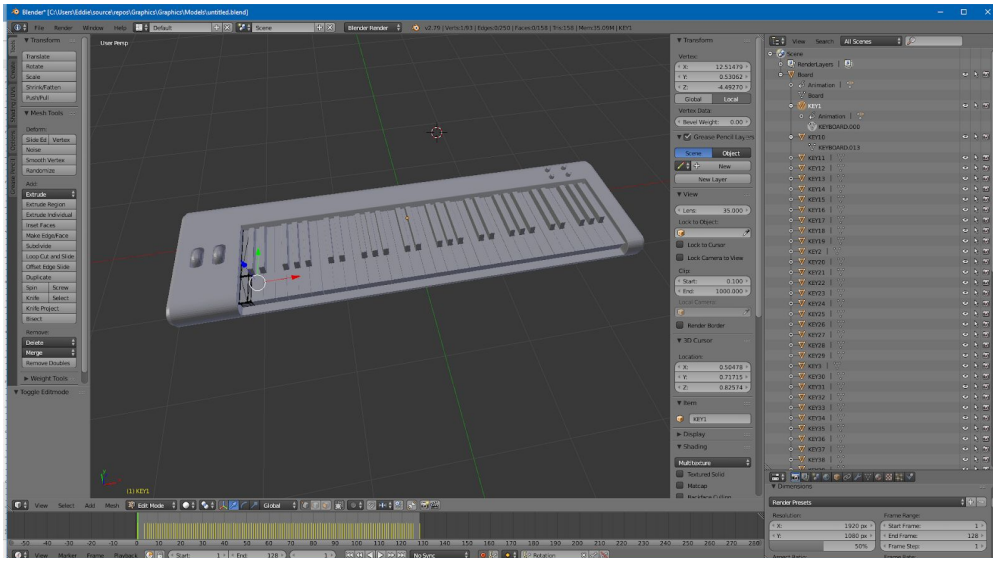
We combine the results together to get our final lighting of a fragment.

```
void main(){

  vec3 cameraDirection = normalize(cameraPosition - modelVertexPosition);
  vec3 result = vec3(0.0, 0.0, 0.0);

  for(int i = 0; i < DIRECTIONAL_LIGHT_LIMIT; i++) {
    result += CalcDirLight(directionalLights[i], modelVertexNormal, cameraDirection);
  }
  for(int i = 0; i < SPOT_LIGHT_LIMIT; i++) {
    result += CalcSpotLight(spotLights[i], modelVertexNormal, modelVertexPosition, cameraDirection);
  }
  for(int i = 0; i < POINT_LIGHT_LIMIT; i++) {
    result += CalcPointLight(pointLights[i], modelVertexNormal, modelVertexPosition, cameraDirection);
  }
  FragColor = vec4 (result, 1.0);
}
```

Note: We don't actually use any **Light::Directional** in our scene.

Blender Models and loading:

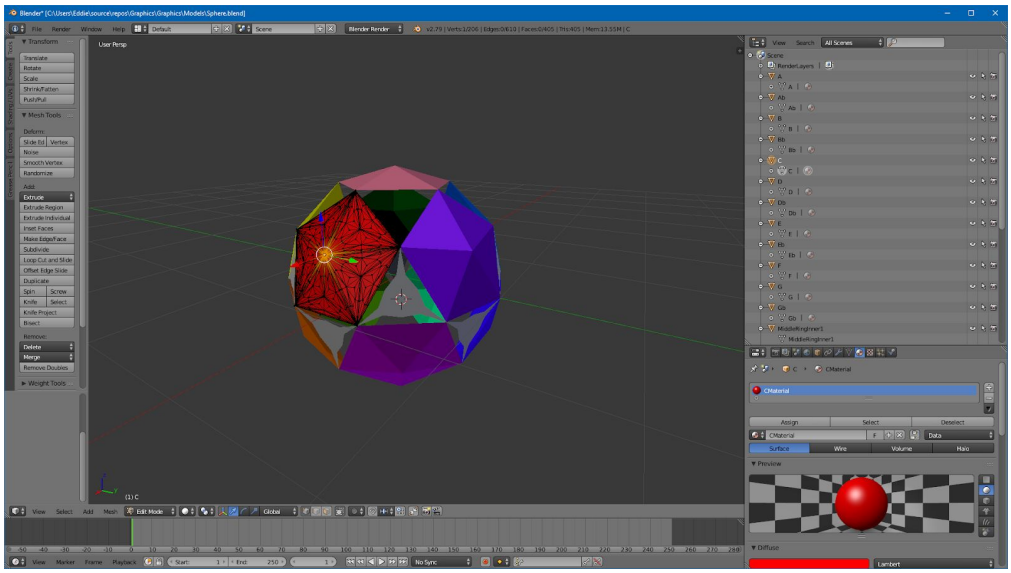
Our original **Keyboard** model was one entire mesh. This posed a problem as we wanted to animate the pressing of the keys. We created each key as its own mesh and added it as a child of the whole Keyboard.



We also had to investigate assimp flags to ensure it did not remove the node hierarchy that we created in blender.

```
//Flags:
//http://assimp.sourceforge.net/lib_html/postprocess_8h.html#a64795260b95f5a4b3f3dc1be4f52e410
const aiScene* scene = aiImportFile(location.c_str(),
  aiProcess_Triangulate
  | aiProcess_JoinIdenticalVertices
  //| aiProcess_PreTransformVertices
);
```

To create our **Sphere** we took an Icosahedron and separated each face into its object with its own mesh, all of them being a child of the Sphere object. Also to have specular lighting visible (all be it quite subtle), we increased the face count of each Face's mesh.



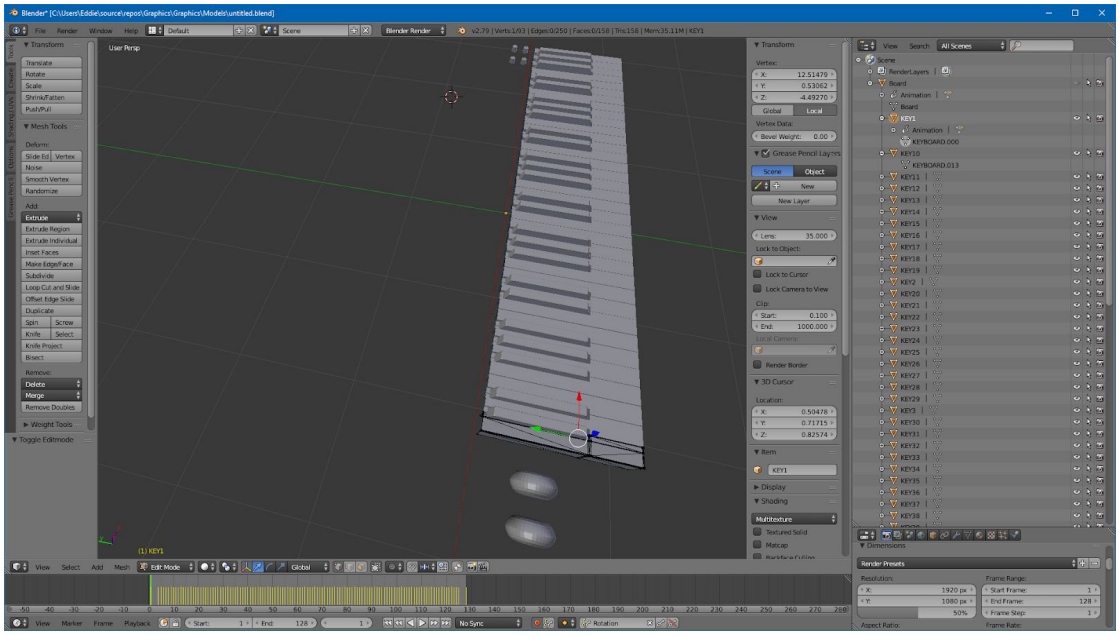
Animation:

Our animations are getting the Keys of the Keyboard to press in when the appropriate note is hit, having the Sphere pulsate with key presses and to also have the Sphere Faces light up and gradually fade out according to the notes pressed. To keep track of our animations, we used the concept of Keyframes.

Keys:

As our keys don't move independently from each other, we keep the origin position the same as the keyboard, meaning we do not need to apply a child transformation for each key.

To make the **Key** presses rotation simple, we lined up the keys and keyboard along the x-axis, allowing us to rotate them by simply rotating its parent matrix.



Keyframes: 128
Angle: 0 -> -4
We map each keyframe to an angle of depression, such that keyframe 0 = -4 degrees and keyframe 128 is 0 degrees. (we've returned to resting position)

Sphere Face Lighting:
When we hit a note, the corresponding face lights up. The lights state stay updated in the shader each frame. We slowly dim the light each keyframe as well until it fades out.

```
void SphereNoteFaceAnimation::animate(glm::mat4 mat)
{
    sphereFace.light.position = glm::vec3(mat * sphereFace.modelNode.transformMatrix * glm::vec4{ 0,0,0,1 });
    sphereFace.light.light.ambient *= 0.95;
    sphereFace.light.light.diffuse *= 0.95;
    sphereFace.light.light.specular *= 0.95;
    Light::Point::updateLightValues(sphereFace.pointLightId, sphereFace.light);
    Light::Point::updatePosition(sphereFace.pointLightId, sphereFace.light);
    keyframe++;
}
```

Sphere Movement:
The last bit of animation we do is having the sphere pulsate on key press and also have the **Ring** (the ring part that rotates out). They follow the same general principle of splitting their keyframe limit into two, half for outward movement away from home and the other half returning to home. They are also designed such that multiple can be active at the same time and still have the Sphere return to its home state.
Ring Rotation Animation

```
glm::mat4 RingAnimation::animate(glm::mat4 mat)
{
    glm::vec3 rots = (keyframe < (KEYFRAME_COUNT / 2.0)
        ? rotations * (float) keyframe
        : rotations * (float) (KEYFRAME_COUNT - keyframe)
    );
    float scaleValue = (keyframe < (KEYFRAME_COUNT / 2.0)
        ? 1 + scaling * (float)keyframe
        : 1 + scaling * (float)(KEYFRAME_COUNT - keyframe)
        );

    glm::mat4 m = glm::scale(mat, scaleValue * glm::vec3{ 1.0,1.0,1.0 });
    m = glm::rotate(m, rots.x, {1.0, 0.0, 0.0});
    m = glm::rotate(m, rots.y, { 0.0, 1.0, 0.0 });
    m = glm::rotate(m, rots.z, { 0.0, 0.0, 1.0 });
    keyframe++;
    return m;
}
```

Sphere pulsating animation

```
glm::mat4 SphereAnimation::animate(glm::mat4 mat)
{
    float scaleValue = (keyframe < (KEYFRAME_COUNT / 2.0)
        ? 1 + scaling * (float)keyframe*2
        : 1 + scaling * (float)(KEYFRAME_COUNT - keyframe)*2
        );
    keyframe++;
    return glm::scale(mat, scaleValue * glm::vec3{ 1.0,1.0,1.0 });
}
```

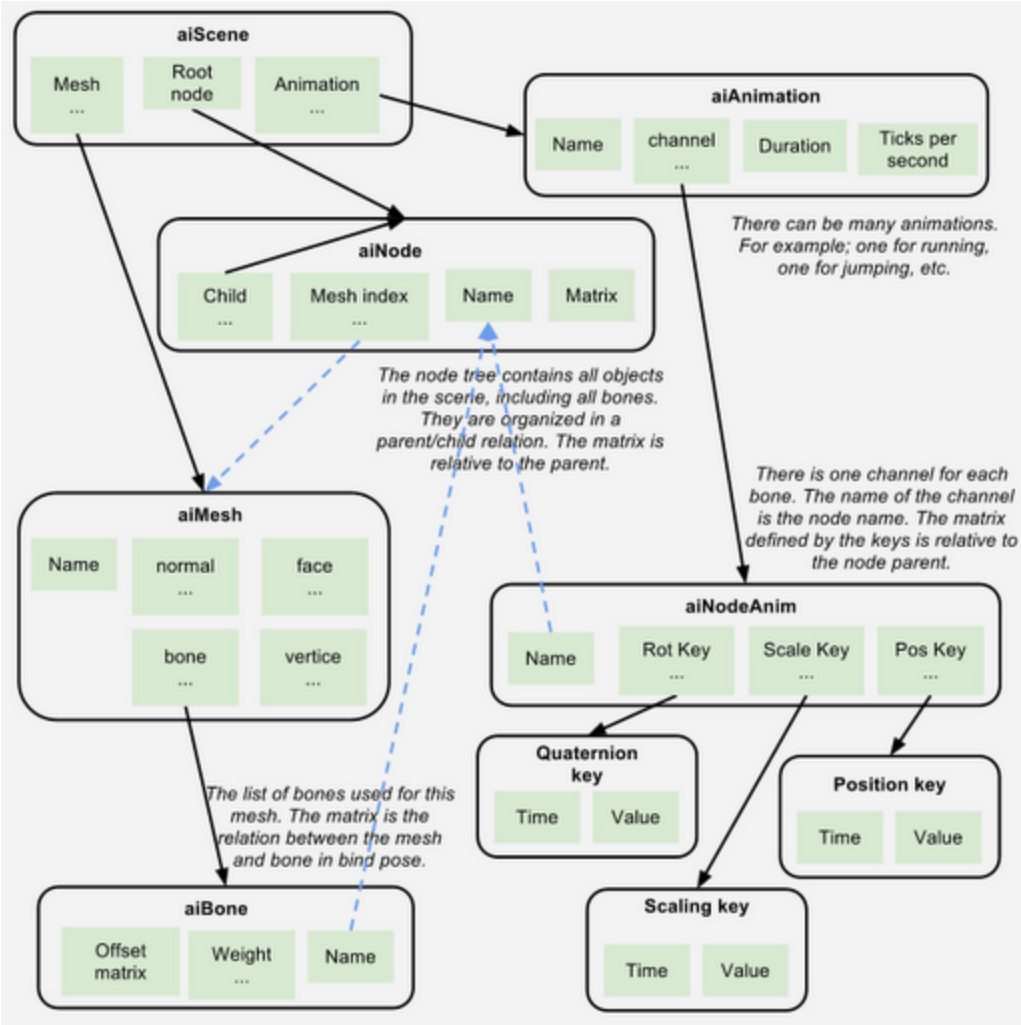
Things that didn't work out:

Textures:

We didn't have a purpose for textures initially so we made the program without any texturing. We thought this could be an easy addition into the overall program once it was complete (perhaps make the floor a large wooden texture). This turned out not to be the case as we needed to handle textured and none textured models differently, requiring alternative Shaders/changing Shader code to be conditional or having single pixel textures to emulate our current **Material** design of giving models their look and feel. We tried initially to integrate textures in but realised that due to time constraints we would be better served forgoing this feature. In conclusion, we should have implemented texturing early on in the design and not as a feature that could be easily added after.

Blender animation imports:

We initially tried to animate the keys in blender and then follow Assimps structure for how animations are stored.



We didn't require any bones but we figured basic translation/scaling/rotation animations would be enough for our animations. If you see the keyboard figure above, you can see keyframes for the animation in the bottom. We exported this model to a Collada file and attempted to load them into our animation structs. After a while we realised Assimp was refusing to detect any animations. We thought this might be an exporter issue but the animations were present in the exported file **keys.dae** so we feel there was an issue with Assimp or the exporter that blender used. In the future we might try using the latest version of Assimp available of their website and seeing if that would fix the issue. In conclusion, we decided to manually create animations in code.

Blender Material imports:

Likewise, **Material**'s didn't pan out as hoped for. We had hoped to use Blender for texturing and material information as that is a prominent feature of the tool. After the mess with animations we decided to manually code in Materials too. We used a static dictionary of all the materials we used.

```
struct Data{
    glm::vec3 diffuseColour;
    glm::vec3 specularColour;
    float shininess;

    Data(glm::vec3 diffuse, glm::vec3 specular, float shininess)
        :diffuseColour(diffuse), specularColour(specular), shininess(shininess) {}
};

static std::unordered_map<std::string, Data> MATERIALS {
    { "board"
        , { Colour::get("red"), Colour::get("white"), 1}
    },
    { "whiteKey"
        , { Colour::get("white"), Colour::get("white"), 1}
    },
    { "blackKey"
        , { Colour::get("black"), Colour::get("white"), 1}
    },
    { "defaultSphereFace"
```

Conclusion:

We learnt a lot creating this project in OpenGL and the course in general. How to manage data and syncing up state between the GPU and a program. How lighting and shading works and the computational challenges that come with realistic lighting modelling. How model hierarchies allow for structured control of models and allow for animations. We learnt to use Blender which we think is an amazing tool for creative purposes.

While the project did not pan out as nice as we had hoped, we feel it gave a good understanding of where the challenges lie with graphical programming.