```cpp
struct VertexData {
    glm::vec3 position;
    glm::vec2 UV;
    glm::vec3 normal;
};

struct ObjData;
using namespace std;

struct ObjData {
    tinyobj::attrib_t attrib;
    std::vector<tinyobj::shape_t> shapes;
    GLulong numFaces;
    GLuint vaoId;
    std::string baseDir;
    std::vector<tinyobj::material_t> materials;
    std::map<std::string, GLuint> textures;
    std::vector<GLuint> indices;
    std::vector<VertexData> vertexList;
    glm::mat4 transform;
    bool loaded = false;
    bool loadedToMem = false;
};
```

```cpp
std::vector<std::pair<ObjData*, std::string>> scene1;
std::vector<std::pair<ObjData*, std::string>> scene2;
std::vector<std::pair<ObjData*, std::string>> scene3;
std::vector<std::pair<ObjData*, std::string>> scene4;
std::vector<std::pair<ObjData*, std::string>> scene5;
```

```cpp
ObjData sword1, crates1, backpacks1;
scene1.push_back( Val: std::make_pair(&sword1, "Sword/moonlight.obj"));
scene1.push_back( Val: std::make_pair(&crates1, "crate/Crate1.obj"));
scene1.push_back( Val: std::make_pair(&backpacks1, "backpack/backpack.obj"));

ObjData backpacks2, crates2, skulls2;
scene2.push_back( Val: std::make_pair(&skulls2, "Skull/Skull.obj"));
scene2.push_back( Val: std::make_pair(&backpacks2, "backpack/backpack.obj"));
scene2.push_back( Val: std::make_pair(&crates2, "crate/Crate1.obj"));

ObjData earths3, skulls3, teapots4;
scene3.push_back( Val: std::make_pair(&skulls3, "Skull/Skull.obj"));
scene3.push_back( Val: std::make_pair(&earths3, "earth/Earth.obj"));
scene3.push_back( Val: std::make_pair(&teapots4, "teacup.obj"));

ObjData pedestals4, skulls4, bunnys4;
scene4.push_back( Val: std::make_pair(&pedestals4, "pedestal/10421_square_pedastal_iterations-2.obj"));
scene4.push_back( Val: std::make_pair(&skulls4, "Skull/Skull.obj"));
scene4.push_back( Val: std::make_pair(&bunnys4, "bunny.obj")
);

ObjData earths5, ganeshas5, swords5;
scene5.push_back( Val: std::make_pair(&earths5, "earth/Earth.obj"));
scene5.push_back( Val: std::make_pair(&swords5, "Sword/moonlight.obj"));
scene5.push_back( Val: std::make_pair(&ganeshas5, "Ganesha/Ganesha.obj"));
```

A scene is represented as a vector of pair structs ObjData which stores the data needed to load an .obj file and its directory in the files which is a string. Every scene has 3 .obj files in it.

```
//start the threads
objLoader scene1Loader( id: 1, [&] scene1, running);
scene1Loader.start();

objLoader scene2Loader(id: 2, [&] scene2, running);
scene2Loader.start();

objLoader scene3Loader(id: 3, [&] scene3, running);
scene3Loader.start();

objLoader scene4Loader(id: 4, [&] scene4, running);
scene4Loader.start();

objLoader scene5Loader(id: 5, [&] scene5, running);
scene5Loader.start();
```

```
Semaphores* running = new Semaphores(available: 2);
```

Each objLoader is an IETThread which will run independently once start() is called. The parameters are the ID, the scene, and a semaphore used to limit the number of threads currently running.

```
void objLoader::run()
{
    load();

    while(true)
    {
        if (LoadBtn == true)
        {
            load();
            LoadBtn = false;

        }
        else if (UnloadBtn == true)
        {
            unload();
            UnloadBtn = false;
        }

    }

}
```

```
void IETThread::start()
{
    std::thread(&IETThread::run, this).detach();
}
```

objLoader.start() executes the run function of the thread. This is because objLoader is a class that inherits from the IETThread class. It loads the scene first by calling the load() function because all scenes need to be loaded on startup. It then goes on a loop checking to see if the imgui button for that specific thread is pressed.

```
void objLoader::load()
{
    if (finishLoad == true)
    {
        return;
    }

    for (int i = 0; i < scene.size(); i++) {
        this->sleep( ms: 5000);

        mutexSem->acquire();
        LoadObjFile_(scene[i].first, &scene[i].second);
        progressPercentage = (float)(i + 1) / (float)scene.size() * 100.0f;
        mutexSem->release();


    }
    std::cout << id << std::endl;

    finishLoad = true;

}
```

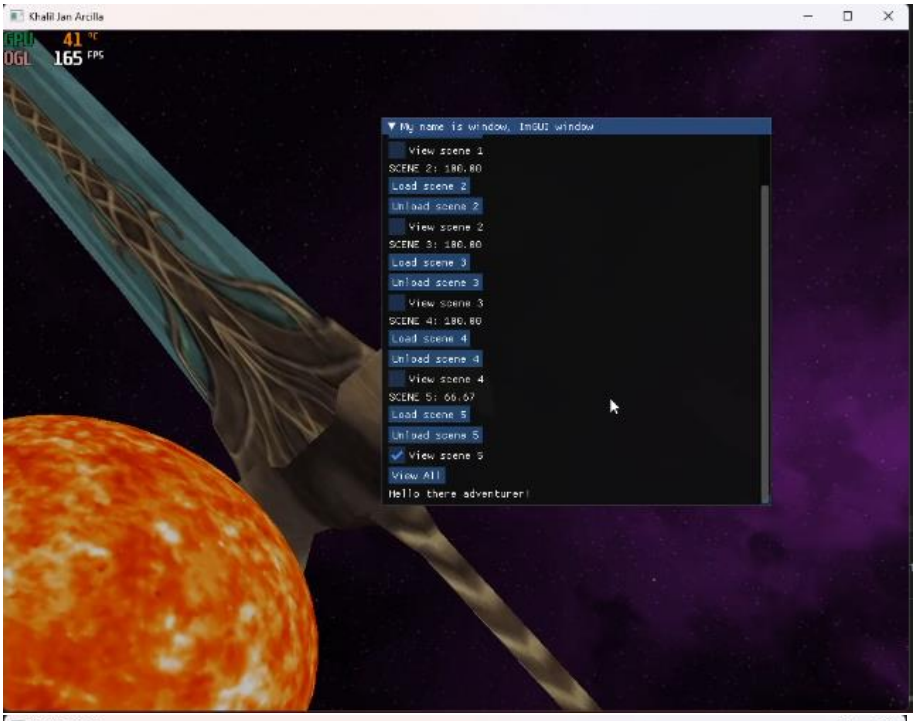
This is how .objs are loaded. The thread is slept for 5000ms and it acquires a semaphore which only has 2 available keys to maintain performance. The LoadObjFile_() function is the one that is filling up the data in the struct ObjData. This why scene[i].first and scene[i].second were passed, the pointer to the ObjData and its location respectively.

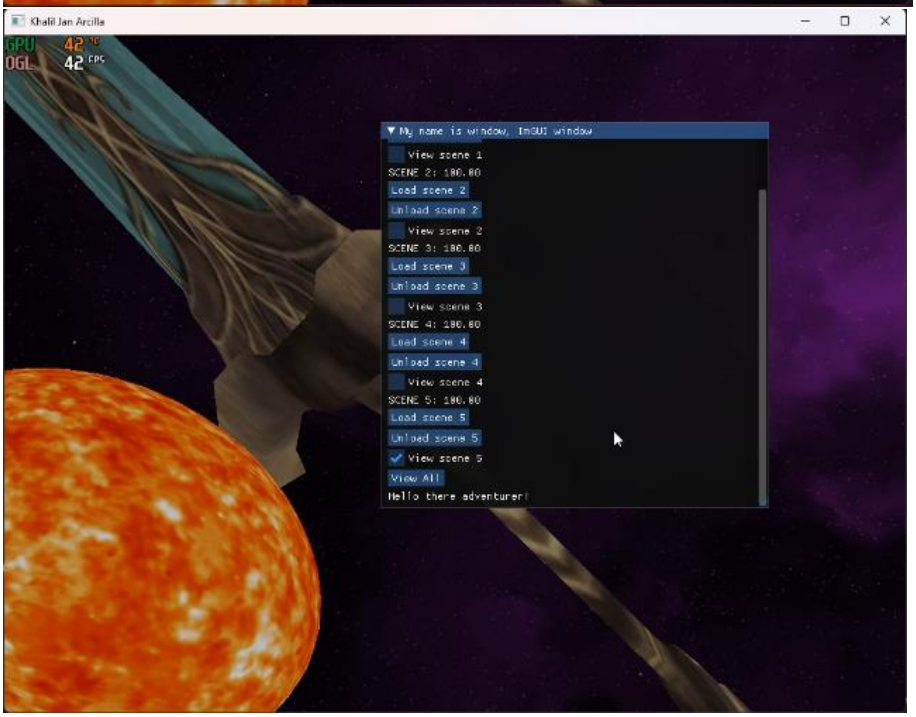
```
for (int i = 0; i < scene1.size(); i++) {
    if (scene1[i].first->loaded && scene1[i].first->loadedToMem == false)
    {
        LoadObjToMemory_(scene1[i].first, scaleFactor: 5.0f, bunnyOffset, texture, texture2: multiTexture1, i);
        scene1[i].first->loadedToMem = true;
    }

}
for (int i = 0; i < scene2.size(); i++) { ... }
for (int i = 0; i < scene3.size(); i++) { ... }
for (int i = 0; i < scene4.size(); i++) { ... }
for (int i = 0; i < scene5.size(); i++) { ... }
```

In the main thread, that is where it is loaded to memory since that is where openGL context is initialized. This is where things like textures and buffers are initialized. Meaning only the main thread is communicating with the GPU and storing to memory.

In this implementation, each thread is only used as a slave thread to load and unload data concurrently; it does not stand alone. For obj files that are small, this implementation is enough, but as seen with the ganesha.obj in scene5, it causes a performance drop when it is loaded into memory because of its size. An alternative to this is to initialize an openGL context for each separate thread and let it have its own resources so that every obj is loaded into memory in separate threads.


Khalil Jan Arcilla
GPU 41 °C
OGL 165 FPS
My name is window, ImGUI window
View scene 1
SCENE 2: 180.80
Load scene 2
Unload scene 2
View scene 2
SCENE 3: 180.80
Load scene 3
Unload scene 3
View scene 3
SCENE 4: 180.80
Load scene 4
Unload scene 4
View scene 4
SCENE 5: 66.67
Load scene 5
Unload scene 5
View scene 5
View All
Hello there adventurer!


Khalil Jan Arcilla
GPU 42 °C
OGL 42 FPS
My name is window, ImGUI window
View scene 1
SCENE 2: 180.80
Load scene 2
Unload scene 2
View scene 2
SCENE 3: 180.80
Load scene 3
Unload scene 3
View scene 3
SCENE 4: 180.80
Load scene 4
Unload scene 4
View scene 4
SCENE 5: 180.80
Load scene 5
Unload scene 5
View scene 5
View All
Hello there adventurer!

```
for (int i = 0; i < scene1.size(); i++) {
    if (glIsVertexArray(scene1[i].first->vaoId) == GL_TRUE && view1 == true)
    {
        glBindVertexArray(scene1[i].first->vaoId);
        glBindTexture(target: GL_TEXTURE_2D, texture: texture[i]);
        glUniformMatrix4fv(modelTransformLoc, 1, GL_FALSE, glm::value_ptr([&] scene1[i].first->transform));
        glDrawElements(mode: GL_TRIANGLES, scene1[i].first->numFaces, type: GL_UNSIGNED_INT, (void*)0);
    }

}
for (int i = 0; i < scene2.size(); i++) { ... }

for (int i = 0; i < scene3.size(); i++) { ... }

for (int i = 0; i < scene4.size(); i++) { ... }

for (int i = 0; i < scene5.size(); i++) { ... }
```

This is how each scene is rendered and drawn. It checks if it has a valid VAO and if the view toggle is checked.

```
if (ImGui::Button(label: "View All")) {
    this->view1 = true;
    this->view2 = true;
    this->view3 = true;
    this->view4 = true;
    this->view5 = true;
}
```

Button to view all scenes at once

```
void objLoader::unload()
{
    //loading->acquire();
    if (finishLoad == false)
    {

        return;
    }
    for (int i = 0; i < scene.size(); i++) {
        std::cout <<"delete: "<< scene[i].first->baseDir << std::endl;
        //glDeleteVertexArrays(1, &scene[i].first->vaoId);
        //unloadVAO();

        scene[i].first->attrib.vertices.clear();
        scene[i].first->attrib.normals.clear();
        scene[i].first->attrib.texcoords.clear();
        scene[i].first->shapes.clear();
        scene[i].first->materials.clear();
        scene[i].first->indices.clear();
        scene[i].first->vertexList.clear();

        for (auto& texture :pair<const string, unsigned>& : scene[i].first->textures) {
            glDeleteTextures(n: 1, &texture.second);
        }
        scene[i].first->textures.clear();
        //UNLOADING DATA
        scene[i].first->loaded = false;
        scene[i].first->loadedToMem = false;

    }
    this->progressPercentage = 0;
    this->finishLoad = false;
    //loading->release();

}
```

```
void objLoader::unloadVAO()
{
    if (finishLoad == false)
    {

        return;
    }
    for (int i = 0; i < scene.size(); i++) {
        std::cout << "delete VAO: " << scene[i].first->baseDir << std::endl;

        glDeleteVertexArrays(1, &scene[i].first->vaoId);

    }
}
```

```
ImGui::Text(fmt: "SCENE 1: %.2f", scene1Loader->progressPercentage);
if (ImGui::Button(label: "Load scene 1")) {
    scene1Loader->LoadBtn = true;
}
if (ImGui::Button(label: "Unload scene 1")) {
    scene1Loader->UnloadBtn = true;
    scene1Loader->unloadVAO();

}

ImGui::Checkbox(label: "View scene 1", &this->view1);
```

This is how a scene is unloaded. The unload function is called when the imgui button is pressed which sets the thread's UnloadBtn to true. The slave threads only clear out the data inside the ObjData. In the main thread, that is where the unloadVAO is called because that is where openGL is initialized. If unloadVAO is called by a thread other than the main thread, it won't be cleared from memory due to how openGL works.