Student Name: Pesch Nathan
Student ID: 0556396

Virtual Reality
Project: OpenGL

VRIJE UNIVERSITEIT BRUSSEL

Contents

1 Overview

This project is an implementation of a basic rendering engine using C++ and OpenGL. To include OpenGL, the GLFW and Glad libraries were used. To support the loading of 3D models, the Assimp library is used. The included physics engine is Bullet Physics. All included libraries were built from source to static library files.
The program shows a level a level of the classic game Super Mario 64, namely Peaches' castle exterior. In this level, a simple physics simulation is performed and a display of the most notable features of the renderer are done. The user is given control over a free moving, first person camera, to be controlled with z, q, s, d, space, shift and ctrl.

2 README

The project can be built on Windows machines by using the included CMakeLists.txt. The binary will be inside /bin/. All textures, models and shader files are in the /res/ folder. All source cpp files are located in the src directory. each header definition file is located in the /include/user/ folder, to separate from the headers of the installed libraries. Relative file paths are used across the board.
All building was performed with the ucrt64 g++ compiler, therefore I do not know if another compiler such as MSVC will work. I included all the statically built libraries (glfw, assimp, Bullet3Common, BulletCollision, BulletDynamics, LinearMath and Assimp) in the /lib/directory. The CMakeLists.txt should automatically link these during building.

Command for building without cmake:
g++ -g -std=c++17 -IProject/include -LProject/lib Project/*.cpp Project/glad.c -glfw3 -lm -lassimp -lBulletDynamics -lBullet3Common -lBulletCollision -lLinearMath -lz -lgdi32 -lopengl32 -o Project/bin/MyProgram.exe

3 Implementation

3.1 Features

3.1.1 Specular textures

Every mesh can load either diffuse or specular textures, or both. For specular textures, the values are assumed to be in the red channel. The fragment shader will then calculate the reflect direction based on the view direction with the vector dot product. Based on this reflect direction, specular light is applied according to the sampled specular texture's red channel.

3.1.2 Light

Point Lights Point lights are implemented as having a physical mesh, but not using the default shaders. A light shader that only outputs the light color as the fragment color is used. To support multiple lights, **Uniform Buffer Objects** are used. The default fragment shader expects an array a uniform array of point light structs that hold the light position, color, and the linear, quadratic and constant factors of the point light equation to be used. Then iteratively for each light in this array, the addition to the current fragment is computed based on the point light fallof equation with the distance of the light to the current fragment. Using uniform buffer objects allows for easily

adding more point lights to the world and updating the properties of this light in real time. For the uniform buffer objects, based on the std140 layout specified by openGL, the offsets of the different values are calculated.

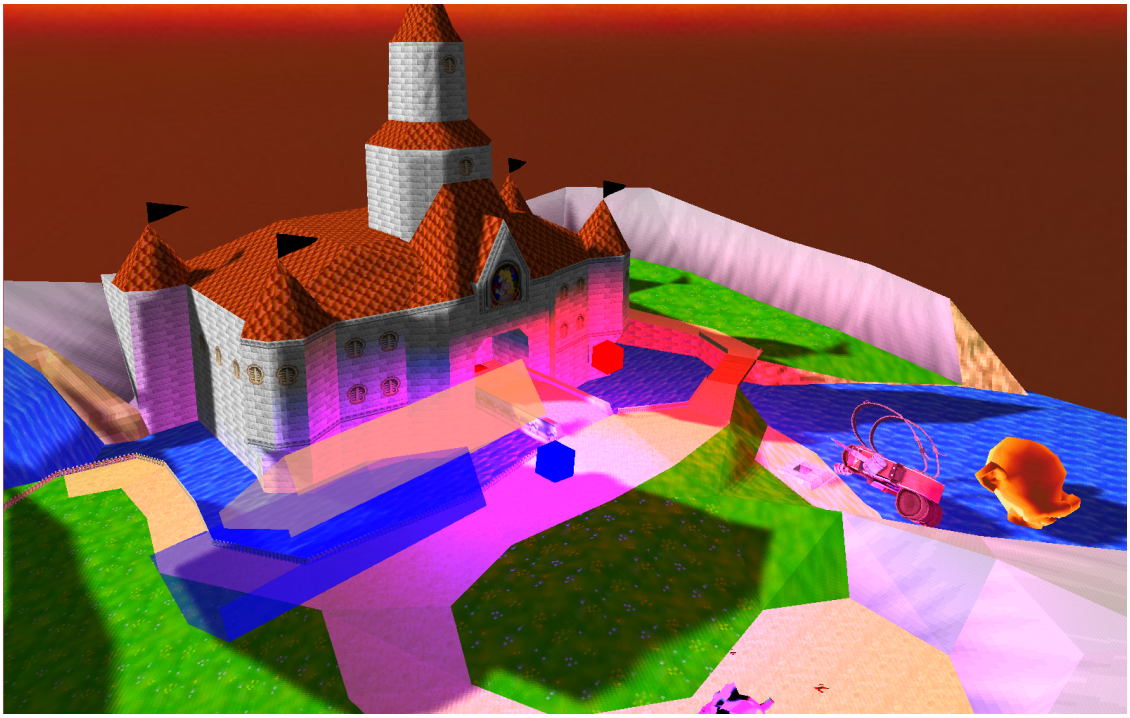

Figure 1: 2 differently colored, moving point lights in the level.

3.1.3 Cubemap

To display a cubemap as the skybox, The Skybox class loads the 6 faces of the cube as textures and assigns them to the faces. A Vertex Array Object and Vertex Buffer object are created with the simple vertices of a cube, and drawing is performed by first setting the depth function to GLEQUAL to make sure it is drawn behind everything in the world, and then removing the translation from the view matrix so that the skybox is always centered around the player. In the vertex shader for the skybox the depth is assigned to 1 so it is effectively displayed infinitely far away from the player.

Figure 2: Skybox texture visible in the level.

The skybox supports refraction on objects. Objects with the def_reflect.frag shader sample the cubemap through a cubesampler to calculate the reflect direction based on the view direction and the normal vector of the current fragment to then compute an envColor value which is output to the FragColor.

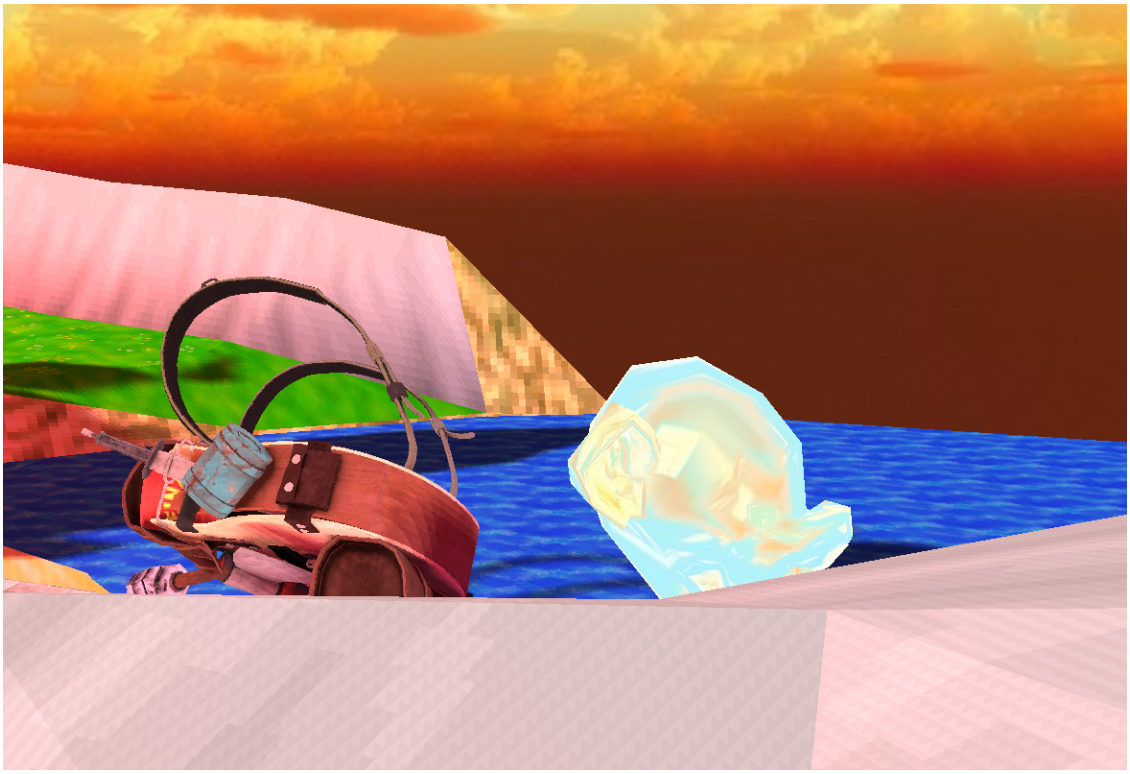

Figure 3: reflection of the skybox on the monkey model.

3.1.4 Transparency and Blending

To support basic transparency in textures, the default fragment shader discards a fragment if it's alpha value is below a threshold of 0.1. To support blending of semi transparent objects, all objects with semi transparent textures are added to a vector. This vector is sorted each iteration by distance to the camera. Objects that are furthest away are drawn first. All semi transparent objects are drawn after the other opaque objects, making sure to enable GL_BLEND to enable the defined blending function, which is FinalColor=(SourceColor x Alpha)+(DestinationColor x (1-Alpha))

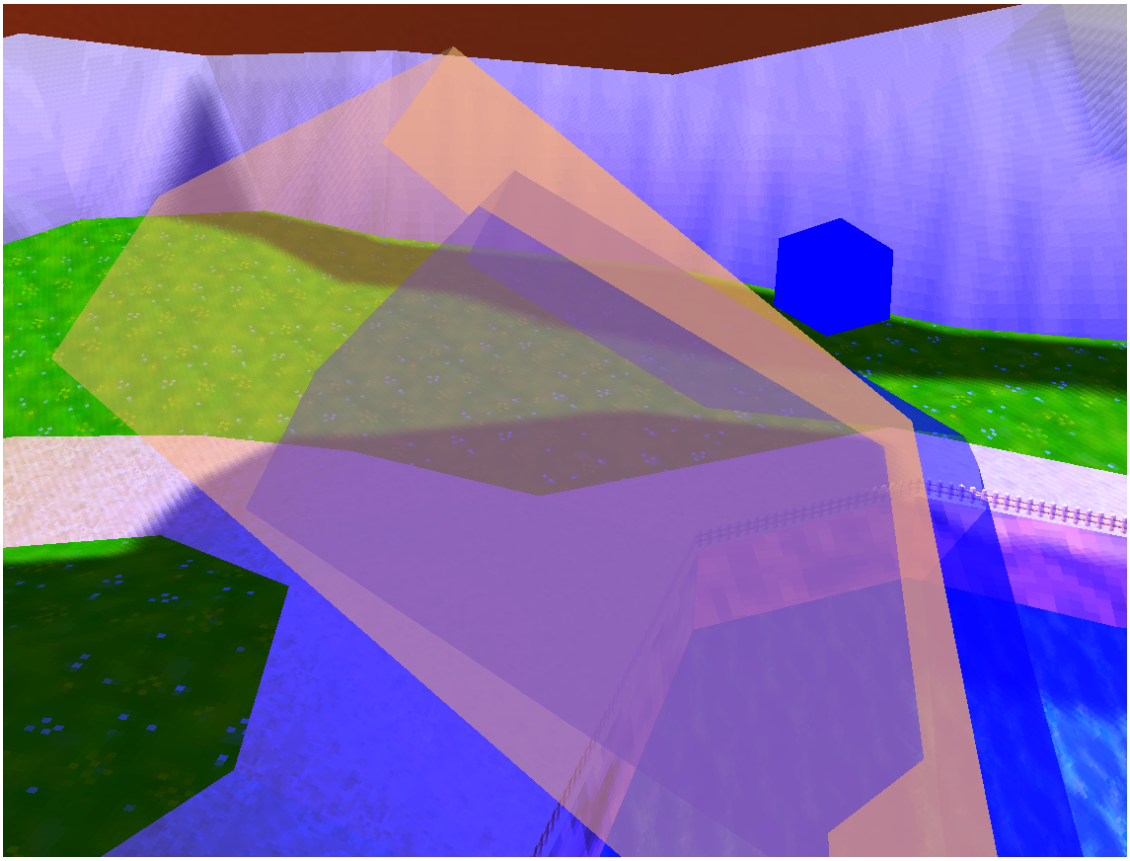

Figure 4: Two semi-transparent cubes blending with each other.

3.1.5 Shadows

Shadows are implemented by constructing a framebuffer object on which the depth of the scene is rendered from the perspective of the directional light (the shadow caster), which gets loaded into a shadowmap texture. Since for directional lights all light rays are considered parallel, an orthogonal projection from the light's position in the world, looking at the origin is constructed. in the drawing phase, first, the shadow frame buffer is bound and the depth info for each object is rendered into the shadow map texture with the shadow map shader, taking into account this computed orthogonal projection. Then, the buffer is unbound and the uniforms for the default shader are updated. Then all objects are drawn as normal. In the default fragment shader, when computing the directional light, The shadow is calculated by converting the fragment light position coordinates to coordinates in clip space, then the depth value of the rendered shadow map is compared to the current depth of the fragment. if it is smaller, the fragment is in shadow. Then a bias is computed to prevent shadow acne. Based on the angle of the surface normal versus the light direction, a larger bias is applied for steeper angles. Then by using a 2D filter kernel with percentage-closer the shadow are smoothed out to prevent harsh edges.

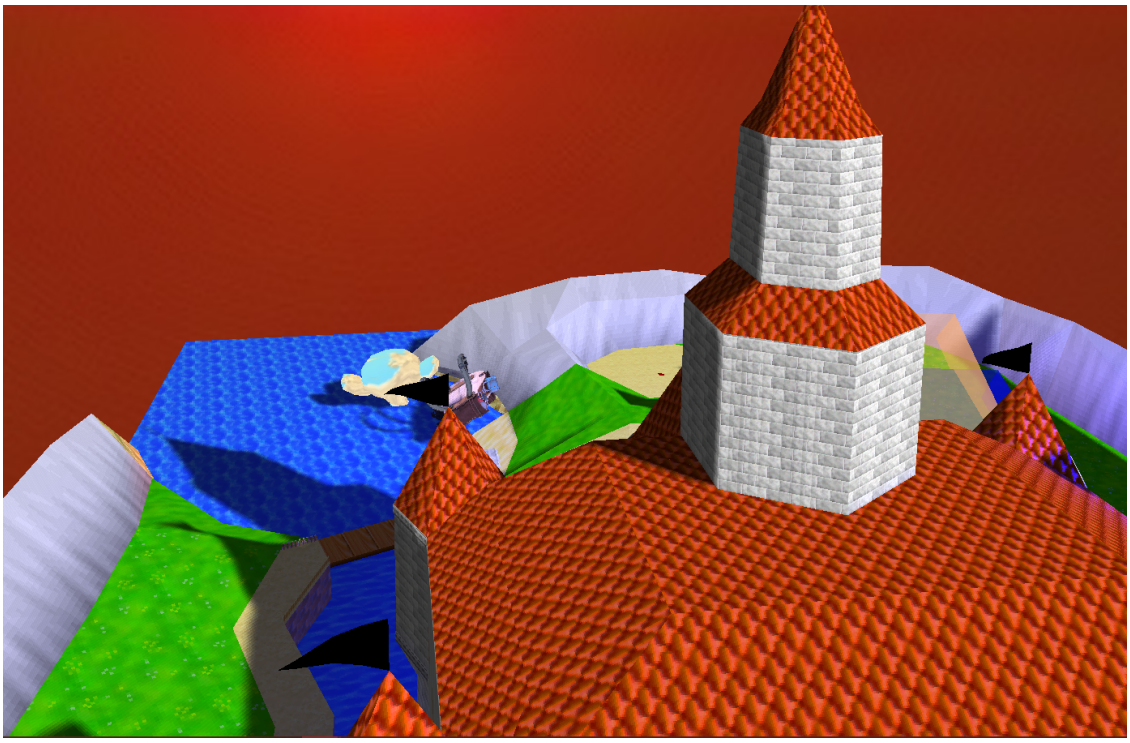

Figure 5: The castle casting its shadow over the landscape.

3.1.6 Physics

The PhysicsNode3D class represents an object with a model and physics. After loading the model, the constructor will generate a collision shape (BVH triangle for static meshes with 0 mass, Convex Hull for objects with mass) based on the loaded model. A user made collisionshape can also be provided. From this collisionshape, a rigidbody is created and the intertia is calculated. These objects need to be added to the dynamicsworld to participate in the simulation. Each iteration of the program loop, these objects call .Update(), which updates the position and rotation of the object according to the world transform in the physics simulation of Bullet Physics. With these updated values, the model matrices are updated, which allows their new state to be drawn. With these features in place, objects with mass fall to the static ground with no pass, rolling and tumbling according to their inertia, creating a realistic physics response.

3.1.7 Model Loading

To support model loading, Assimp is used. The project supports a multitude of filetypes, including fbx, obj, gltf, glb, dae (collada), ... The model loader class processes each mesh in the model, Node by node and mesh by mesh. For each mesh, the vertex, normal and texture data is read from the assimp data objects.
Textures are loaded based on their type: diffuse textures are loaded into the diffuse slots, specular textures into the specular slot. The texture data is loaded separately if the textures are embedded. External textures are assumed to be in the ../res/textures folder. They are then loaded like other textures in the texture class. Embedded textures can be uncompressed or compressed. Uncompressed textures can be directly loaded with stbi like other textures. Compressed textures need to be cast into unsigned char* type to allow stbi to load it, regardless of whether it is in jpg or png

format. The loaded data is the also fed into an instance of the texture class and assigned to the mesh.

3.2 Architecture

The main architecture choices for this project are the separation into Node3D and PhysicsNode3D classes. Node3D is a parent class that has no physics interaction, but does have support for a model. The point lights in the projects are derived from Node3D. PhysicsNode3D is also derived from Node3D, but adds the Bullet Physics functionality. All these classes derive from the Node class. This node class was created to conceptually support trees of nodes being each other parent's and children, allowing the transform changes to be propagated, children to be added relative to others and such.