# Rigid Body Simulation with OpenGL and BulletPhysics

ICG 2020 Final Project Report, B06902109 林仁傑

## 1. Introduction

In this project, I implemented a Rigid Body Simulation System with OpenGL and BulletPhysics. The scene I created is multiple balls bouncing in a box, though other scenes can be created with my system.

## 2. Environment

A. GLFW 3.3.2 for Microsoft Windows API Wrapping [1]

B. GLAD OpenGL Loader [2]

C. GLM 0.9.9.8 OpenGL Mathematics Library [3]

D. Bullet3 Physics SDK [4]

E. OpenGL 3.3

F. “happLY” Stanford PLY Format File Loader [5]

G. Blender 2.8.2a for creating meshes

H. Visual Studio 2019 (Build Release only; Libraries only set up for Release)

I. Windows 10 x64

# 3. Details

First, we use GLAD to auto detect our system and setup OpenGL. Then, we initialize GLFW to create a window in Windows system. Next, we load the models created with Blender in Stanford PLY format. For each mesh, we assign a VAO – Vertex Array Object to it to uniquely identify this mesh. We then bind a VBO – Vertex Buffer Object, which points to a vertex buffer, and an EBO – Element Buffer Object, which points to an index buffer, to each VAO. Thus we only need to bind the corresponding VAO when we want to render a mesh. Then, we create Model Objects which represents instances of meshes, i.e. we could have multiple instances of same mesh with different positions, rotations, and so on. Each Model Object points to a VAO which then points to the mesh it uses. Then, for each Model Object we wish to have physics on, we create a btRigidBody object which represents the collision object in the Bullet Physics system. We do not want to directly use the mesh to calculate collision because it is too expensive. Finally, for each Model Object, we generate the Model Matrix, and send it to the shader along with the Perspective Matrix and View Matrix generated from the singleton CameraObject which represents the camera. Then, we call the Bullet Physics system to update for a constant timestep, so that we can update the Model Object positions in OpenGL system according to the Bullet Physics Collision Object.

# 4. Difficulties

Most difficulties come from migrating from WebGL to OpenGL. Certain functions must be explicitly called in OpenGL, in contrast with WebGL. For example, "glEnable(GL_DEPTH_TEST)" must be called in OpenGL to enable Z-Buffering, while WebGL has such function enabled by default. I spend a lot of time debugging this, because I initially thought it is Blender which has exported the wrong normal vectors. Another difficulty is that I declared 2 std::vectors in the ModelBuffer Class to store the vertices and indices, then I only need to pass std::vector::data() to glBufferData, also avoids using C-style malloc() and free(), which could result in many Segmentation Faults. However, if I push back the ModelBuffer class into a std::vector, OpenGL will not render anything. With extensive debugging, I finally found out that the default behavior for std::vector:: push_back() is use the copy constructor to initialize the element in the vector, which means the vertex vector and index vector inside the ModelBuffer Class will be copied to a new location, but we have already bound the old location to the GPU, which will be destructed at the end of the std::vector:: push_back() evaluation. Thus, we need to force std::vector:: push_back() to use move constructor to force the ModelBuffer object in the vector to use the same memory location as the one we passed in.

# 5. Coding

Comments have been added to the source files. Please download them from

[8]

# 6. References


[1]

https://www.glfw.org/

[2]

https://glad.dav1d.de/

[3]

https://glm.g-truc.net/0.9.9/index.html

[4]

https://github.com/bulletphysics/bullet3

[5]

https://github.com/nmwsharp/happly

[6]

https://learnopengl.com/ for WebGL to OpenGL Transition

[7]

https://github.com/bulletphysics/bullet3/blob/master/examples/HelloWorld/HelloWorld.cpp/ for Bullet3 Physics Tutorial

[8]