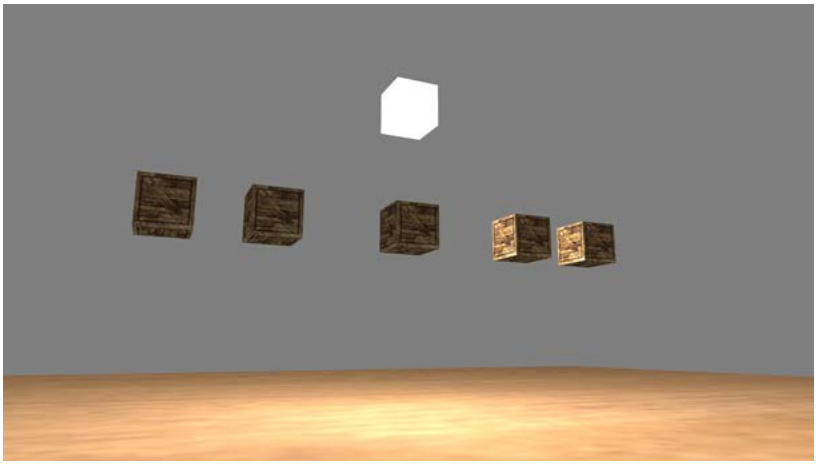

REAL-TIME RIGIDBODY PHYSICS ENGINE

A simple engine that simulates 3D box physics with kinematics and collisions

ABSTRACT

Simulating physics phenomena in video games is a complex and difficult process. It requires a solid understanding of the laws of motion and a proper implementation of them into code in order to produce the most heuristic technique possible. In this project, simulated physics are demonstrated under ideal conditions with additional handy tools and features that make the demonstration more appealing.

George Mavroeidis (40065356)

COMP 477 Fall 2020 – Dr. Tiberiu Popa

TABLE OF CONTENTS

1. MOTIVATION

For the past five years, I've been developing projects using graphics engines that contain rigid body physics. Blender, Unity, Unreal Engine 4 and others are examples of established real-time engines that have such a mature and powerful feature. With just a few clicks, it is possible to replicate realistic motions of physical objects with high accuracy and modifiability. In video game development, most of my projects involve some sort of mechanics principles and laws of physics that are already pre-computed with the engine, but I am able to experiment with their parameters. For example, a scene with a ball that moves in space and bounces off walls, may take a few minutes or even seconds to set up in Unity. However, the source code behind it deceives its simplicity on the surface. Simulating heuristic phenomena in computer graphics is an overlooked challenge that not many computer science enthusiasts consider understanding on a deeper level.

Although it is a subject that has been discussed extensively, I am interested in testing the basic features with certain variations on their properties. To simplify things, the project will focus on cubes and planes. Cube to cube collisions and forces will be the main focus. Additional features such as ray casting will add some fun by allowing the user to manually apply forces on the cubes. An example that inspired me for this project, was a Unity tutorial, where plastic cups are stacked up like a pyramid, and the user pushes them, resulting to a crumbling set of cups. I was inspired to attempt creating similar interactions between the cubes in my program.

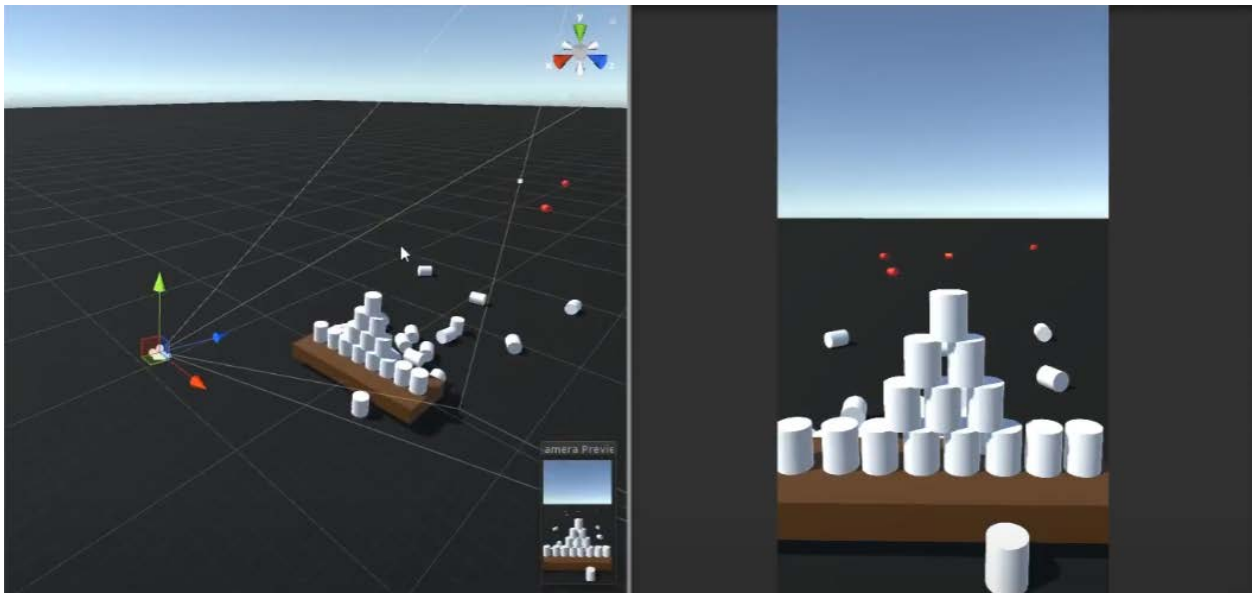

Fig 1: Made in Unity by DitzelGames on Youtube

2. INTRODUCTION

Rigid body Dynamics Simulation is a mature topic in computer graphics and has been the core part of creating a game engine from scratch. The theory behind this topic is heavy, but simple to understand. What makes it very challenging is its translation in code, which is what this project attempts to do. In the following chapters, the mechanics behind the project will be explained visually through diagrams and code snippets. Moreover, the features of the engine will be discussed and give a reason as to why they were implemented. Finally, different results and their difficulties will be examined alongside with a possible solution in order to improve the final product. Personal experiences with the engine's creation will also be expressed.

3. THEORY BEHIND ENGINE

A. ORIENTATION

Basically, these parameters that position the box into the virtual world. Basic measurements like position, rotation and scale are interpreted into a 3D space. All these parameters update the World Transformation Matrix of the Box, a 4x4 matrix that follows the Model-View-Projection model. The world Transformation matrix is able to position the box in the world as well as project its view into the camera window, this case being a perspective view. The following snippets of code can be found into the source code. Check the source code for more details:

```
// ORIENTATION
glm::vec3 mPosition;
glm::vec3 mRotation;
float mRotationAngleInDegrees;


glm::vec3 mScaling;
glm::vec3 mHalfWidths;
float mRadius;

glm::mat4 mWorldTransformationMatrix;
```

Fig 2: Orientation variables (more information in source code)

B. MECHANICS

Mechanics use certain equations to find the new orientation of an object into the world. Applied forces and torques are what make an object move into space. In programming terms, these functions

are updating the box's orientation every frame at time dt. Most of these are pretty obvious of what they are. Gravity and mass remain constant while linear, angular velocities and inertia are accelerating and angulating the box.

```
// PHYSICS:
float mGravity;
float mMass;
glm::vec3 mLinearVelocity;
glm::vec3 mAngularVelocity;
float mAngularVelocityInDegrees;
glm::mat4 mInvInersia;
```

Fig 3: Physics variables (more information in source code)

C. COLLISION DETECTION

Collision detection is implemented through OBB, more specifically, mesh collision which for this case works perfectly since it's the exact shape of the box. The way it works is the world iterates through all the objects and checks for any vertex penetrations between them. With a simple comparison, the exact coordinates of the points are detected. From there, the engine calculates the direction in which the objects will bounce towards. The whole process of checking collisions has a time complexity of $O(n^3)$ for every frame. Although it is not optimized, it is a simple solution for Bounding Box principles. Here is a simple visualization of the process (Image visualized in blender):

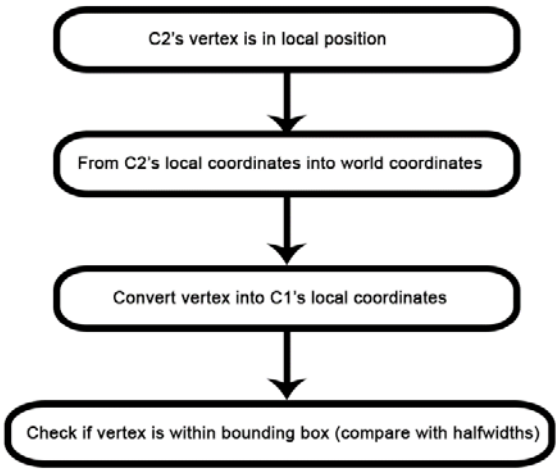

Fig 4: Short pipeline of OBB Detection Collision

One other important thing that must be done before collision resolution, the detection must reposition the boxes to a point where there is no plane intersection, but only vertex. This means the vertex that penetrated the other box's surface must lie on top of the normal surface it penetrated at time dt. In simplified terms, this can be solved by projection. For simplifying the explanation, the following diagram gives an example of box repositioning right before collision detection has ended.

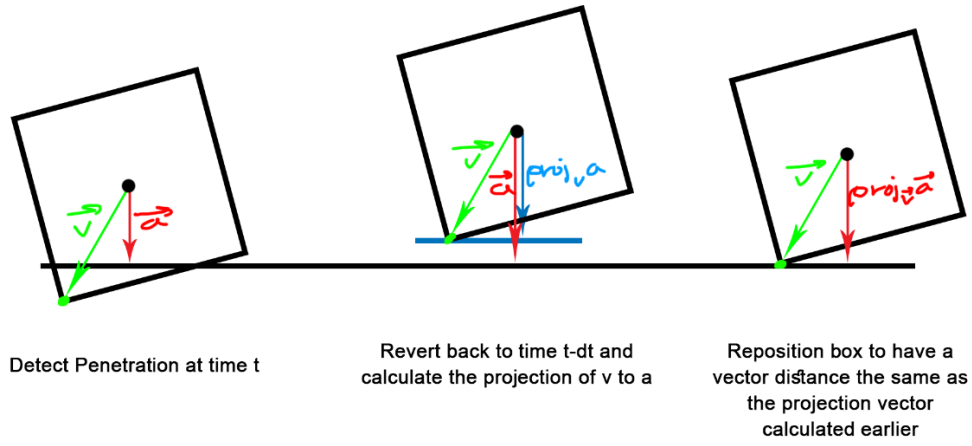

Fig 5: Short pipeline of OBB Detection Collision

```
/*
    IMPORTANT: OBB Collision detection between two cubes, returns boolean
*/
bool World::CheckPenetration(Cube* o1, Cube* o2)
{
    // Iterate through second cube and turn each vertex into world coordinates
    for (int i = 0; i < o2->GetVertexPositions().size(); i++) {
        vec3 vertexWorldCoordinates = o2->GetWorldTransformationMatrix() * vec4(o2->GetVertexPositions().at(i), 1.0f);

        vec4 worldSpacePoint(vertexWorldCoordinates, 1.0f);

        //We will first transform the position into Cube Model Space
        mat4 mTranslate = translate(o1->GetPosition());
        mat4 mRotation = rotate(o1->GetRotationalAngle(), o1->GetRotation());
        mat4 mScale = scale(o1->GetScaling());

        // Put vertex coordinates from o2 into local space of o1
        vec3 cubeModelSpacePoint = glm::inverse(mTranslate * mRotation * mScale) * worldSpacePoint;
        vec3 halfSize = o1->GetScaling() * 0.5f;

        // Check if the vertex from o2 is contained within the first cube's bounding box
        bool containsPoint =    cubeModelSpacePoint.x >= -halfSize.x && cubeModelSpacePoint.x <= halfSize.x &&
                                cubeModelSpacePoint.y >= -halfSize.y && cubeModelSpacePoint.y <= halfSize.y &&
                                cubeModelSpacePoint.z >= -halfSize.z && cubeModelSpacePoint.z <= halfSize.z;

        if (containsPoint) {
            return true;
        }
    }
    return false;
}
```

Fig 6: Following code snippet is responsible for the pipeline of OBB collision detection in Fig 4

D. COLLISION RESOLUTION:

Once the detection has been detected, the point of collision has to be calculated and represented in world coordinates. The coordinates are calculated similarly to the function mentioned in Figure 6, but this time we return the world coordinates instead of a boolean. The following figure gives a visual representation of how a collision reaction works. New velocities are calculated for both boxes based on the collision point's coordinates and normal.

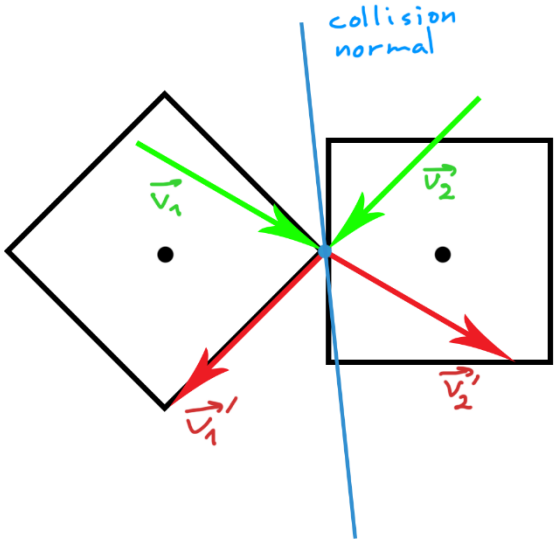

Fig 6: An ideal cube to cube impulse

Impulse is the change in linear momentum when objects collide together. The Online SIGGRAPH 2001 course notes by David Baraff provided the full equation for applying impulses on collided objects, right after collision detection. The source xbdev.net provided a code implementation for this equation. To see the full code and references, please look at the source code at World.cpp file.

$$j = \frac{-(1+\epsilon)v_{rel}^{-}}{\frac{1}{M_a} + \frac{1}{M_b} + \hat{n}(t_0) \cdot \left(I_a^{-1}(t_0)\left(r_a \times \hat{n}(t_0)\right)\right) \times r_a + \hat{n}(t_0) \cdot \left(I_b^{-1}(t_0)\left(r_b \times \hat{n}(t_0)\right)\right) \times r_b}$$

Fig 7: The change in linear momentum calculates a new linear velocity that is applied to both objects at the exact instance they collide

4. PROJECT FEATURES

The engine contains different features that simulate object physics at ideal conditions. Although heuristic phenomena are not part of it, the basic principles are applied to give the user an introduction to the concept and open the doors for more opportunities for the engine to grow. The following objectives were followed in order to complete the project:

A. THE CAMERA

The camera has a first-person perspective view and is initialized at an elevated angle with a minor pitch downwards. The player can use the Event Manager to control the camera's rotation using mouse movement and WASD keys to translate camera in all axes. The free movement of the camera allows the user to interact with the environment and view the scene from all angles. In addition, the ray casting feature is used with the camera, which will be explained later.

B. THE LIGHT

Having its own shader and properties, the light source is a static object that its sole purpose in the scene is to provide pleasing visuals for the viewer. It facilitates on interpreting the box's orientation in the scene by looking at the reflected light from its surfaces.

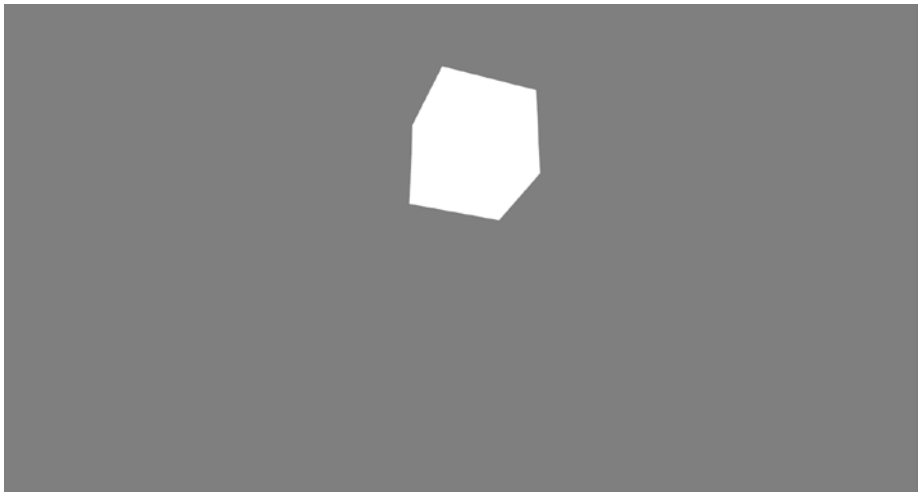

Fig 8: Light source placed on top of the center of the scene

C. THE WORLD

The world, or the scene, loads the objects and manages their properties over time, including all *.obj* files and *.png* files for modeling and texturing. It manages their orientation, their rendering execution and the box's collisions, using global functions with the respective equations. The world stores the boxes in a container, where it iterates through all of them to modify their properties at time dt, for each frame. Additionally, the world receives the input features received from the user at the Demo Manager. One note to remember, the boxes can bounce from an infinite XZ plane and won't fall off if they bounce beyond the wooden surface.

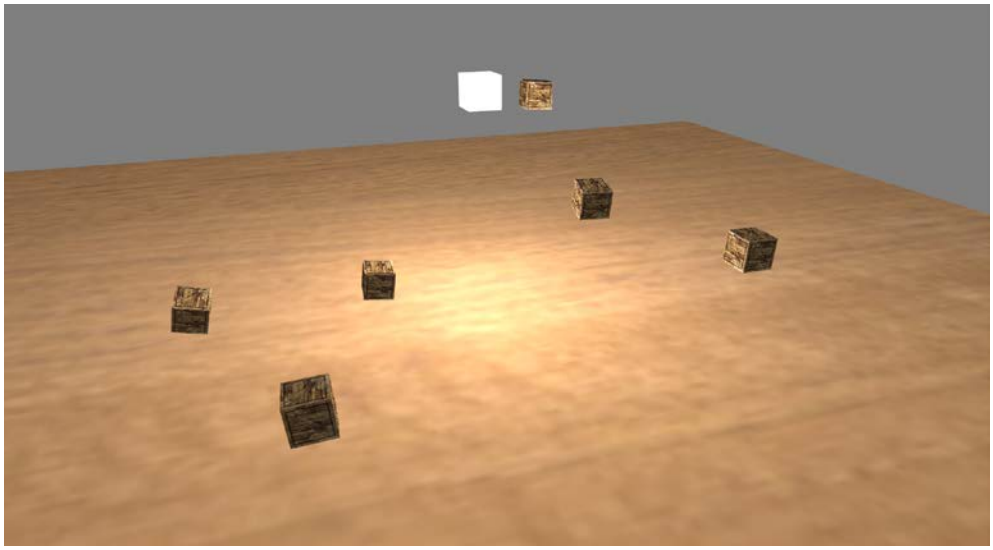

Fig 9: The world scene in action! Boxes are bouncing around the map

D. THE RENDERER

In a few words, the renderer is just a manager that used to draw the objects in the scene. It loads all the necessary shaders in the program and manages the frames rendered through the window. Rendering technique and framework was provided by the instructors of COMP 371.

E. THE CUBE

The main feature of the engine. Contains internally all the physics attributes that label the object as a rigid body with a dynamic behavior. The box can work with two features: either bounce infinitely or simulate gravity by using its sleeping state. The sleeping state prevents unnecessary collisions and freezes the object when its speed has reached a very low magnitude. Through code, there could be added an infinitely number of boxes, but ideally it is recommended for their number to be kept within a range of 10 to 15, depending on the frame rate and the limits of the algorithm's complexity.

F. THE RAY

The ray is a hypothetical line that is shot by the camera and has an infinite distance, depending if it intersects any boxes. The ray attempts to intersect with the closed box. Once intersection has been detected, the ray applies force on the direction of the box's surface that it hit. Similarly, to the cube-to-cube collision, the ray attempts to find the normal surface that it's hitting and applies a force for every frame the user holds the trigger. Using the camera, the player can fly next to cubes and shoot at them. This feature was implemented to allow the user to control the box's movement, in addition to gravity and impulses by cube-to-cube collisions.

5. RESULTS AND DIFFICULTIES

In the beginning of the semester, I felt like this subject could easily be tackled with the right resources and research done. Since mechanics is a subject that has been extensively taught and studied throughout my academic years, it felt encouraging to work on something familiar and easy to research on. I was able to find plenty of papers and articles that were tackling rigid body physics as well as collision handling in an effective manner. Little did I realize the number of cases I had to consider while working on this project.

What I should have done is work from the beginning with minimal conditions and slowly increment the complexity. Although I initially started working with spheres, I was unable to get some properties right. One of the most important properties that cost me the completion of the project was getting the local axes right, a very easy task that I was unable to implement properly. Another issue encountered was repositioning of the box during collision detection, like in figure 5.

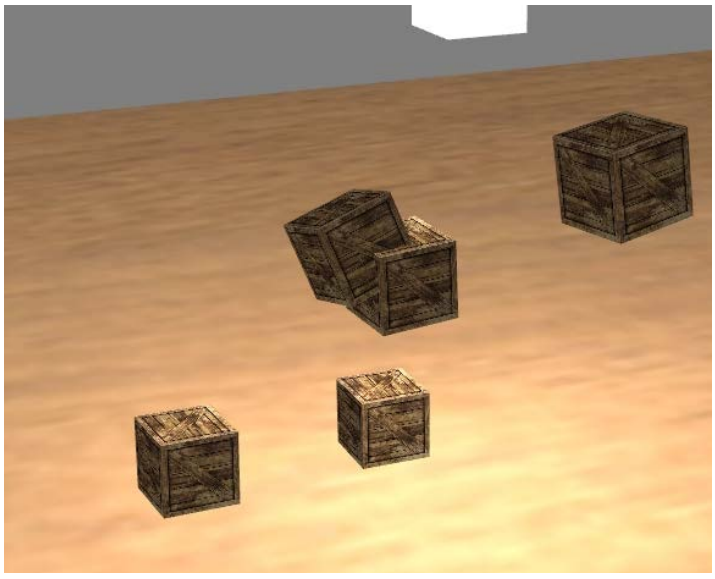

Fig 10: When boxes collide like this, they keep orbiting one another until the force is too great and the cubes are no longer penetrating each other. Ironically, this is one of the laws of physics, similar to planets orbiting one another.

Because of this unfinished feature, when the cube penetration is deep, the box may occasionally interlock with another one, creating a chaotic mass of cubes with no intended direction. The impulse direction gets repeatedly calculated within their bounding boxes. In figure 10 bellow, an accurate example of this occurrence is presented. This usually happens on either too high or too low velocities, where the next step after the impulse still results in the cubes still being locked together, creating some hilarious and aggressive outcomes.

In general, most of the time the results are very satisfying to watch. After implementing a "vacuum force" that pulls the boxes at the middle of the scene, the collisions are a lot more frequent. With eight boxes implemented in the scene, their motion feels harmonic, yet very chaotic. This vacuum force was initially used to test the collision handling part of the engine, but since it produces interesting results, it is now a feature that can be toggled at the start of the program.

5. POTENTIAL SOLUTIONS AND CONCLUSION

Like mentioned previously on figure 10, a potential fix to this issue is to properly reposition the box penetration so the collision point rests exactly upon the cube's surface, rather than inside its bounding box. This would prevent the penetrated vertices to remain inside another mesh, disabling any further collisions the next frame after the impulse has been applied.

For the rotation, Quaternions should have been the primary type for rotations, instead of separating the angle and axis tot wo different types (vec3 and float). OpenGL utilizes transformation matrices and uniform vec3's to move an object in space. Alternative libraries like DirectX9 or DirectX12 already contain a Quaternion type that works just fine when it comes to calculating local spaces. During the final days of the project, I encountered a complete open-source Quaternion library that could be used in OpenGL. Fixing this problem also allows for proper implementation of Inertia and its inverse, leading to finding the kinetic energy of the box at time dt. In addition, this allows for net forces and torques to be implemented, leading to a proper gravitational force, rather than the current one, which is just reversing the velocity vector towards the other direction. This is not realistic and this is why I was unable to set up the scene as shown in the motivation of the project.

During the holidays, I decided to try and restart the project and follow a more steady and realistic approach for rigid body physics or built upon the current one and continue to add features that make the program more enjoyable. Special thanks to Sheldon Andrews, a guest speaker at SCA 2020, who provided me with the research paper for SIGGRAPH 2001's online course notes for rigid body dynamics and collision handling. I would also like to thank Matt Carpenter and Ben Kenwright for letting me use their demo projects from XBDEV.net to better understand the integration of such complex equations into C++ code. All of their sources will be linked at the reference section and the README.txt file of the project.

6. REFERENCES

1. Baraff, D. (2001). Physically based modeling: Rigid body simulation. SIGGRAPH Course Notes, ACM SIGGRAPH, 2(1), 2-1.

2. Kenwright, Ben. “Game Physics Programming: Simple Rigid Body Physics.” Xbdev.net, 2005, xbdev.net/physics/RigidBodyImpulseCubes/?leftbarHIDE=1.

3. Carpenter, Matt “Box Rigid Body Physics Engine. ” derived from Xbdev.net, 2016, youtube.com/watch?v=pVUJXz3Hh0E.

4. Bergeron, Nicolas “COMP 371 OpenGL Framework for Labs”, 2014, Concordia University

5. Ditzel Games “Touch Shoot | Find touched object in Unity”, May 8, 2018, Youtube, youtube.com/watch?v=G8cePMSkALI&t=338s

For any additional information, do not hesitate to contact me at:

- Email: gdmavroeidis@hotmail.com
- Discord: Adamadon #0079
- Concordia ID: 40065356