Rigid Body Physics

Dean Perillo
Undergraduate Computer Science Capstone
Capstone Advisor: Dr. Reale
SUNY Polytechnic Institute
Spring 2023

Introduction

I have always been interested in how real-world physics can be accurately modeled on a computer. From scientific simulations to video games, there are many instances of different types of physics being realistically simulated on various devices. I am setting out to create a 3D rigid body physics simulation from scratch with C++ and OpenGL.

For me, this is no simple endeavor. There is a lot of math involved to achieve something like this. Various things need to be accomplished such as 3D rendering, collision detection, realistic physics response, linear motion, angular motion, gravity, friction, restitution, and physics integrators.

Being able to accurately simulate aspects of the real world has various important uses, specifically in various engineering fields, where various physics simulations can be used to help design better prototypes of various technologies. One good example of this is using aerodynamics simulations to make the best shape for an aircraft to reduce the amount of drag being applied.

Related work

Rigid body simulations are used in various game engines to accurately simulate how objects behave when interacting with each other. Unreal Engine and Unity are great examples of this. Similarly, 3D modeling and animation software such as Blender and Maya use physics simulations to model these physics simulations to produce realistic physics for movies and shows.

Method

I went about this by breaking the task down into smaller components, completing prototypes for these components, then bringing everything together for the final product. Additionally, I decided to first start with two-dimensional physics, getting things working on a small-scale prototype then scaling to 3D, which made a much easier process than just starting with the additional complexities that came with 3D. These first prototypes are written in java, as it is very easy for me to get graphics going. The first thing I set out to do is collision detection, this is probably the most fundamental part of simulation physics like this. The main complexities with collision detection are that we need to account for many scenarios, where objects are in various positions and orientations, and can be completely different shapes. There are two collision detection algorithms that I considered which meet these criteria. The Separating Axis Theorem (SAT) algorithm and the Gilbert-Johnson-Keerthi (GJK) algorithm. On the surface they meet the criteria, they both can detect if two convex ships in any orientation are colliding. However, the GJK algorithm can easily be extended to include additional information that we need to determine a proper physics response. That is the collision depth, normal, and intersection point, which I will return to later.

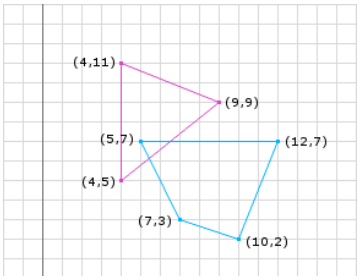

Figure 1: Two shapes intersecting [1]

The GJK algorithm makes use of a concept known as the Minkowski difference. If two objects are colliding, two points within each shape, when subtracted with each other, will equal the origin. The entire set of every point on one shape subtracted by every point of

another, if the two objects are colliding, the origin will be contained by the resulting shape. Figure 1 shows two colliding shapes, and figure 2 shows it's Minkowski Difference, which since they're intersecting, contains the origin. The goal of the GJK algorithm is to determine whether the origin does or doesn't contain the origin. However, calculating the entire Minkowski difference can be costly, especially if shapes have an extremely high number of vertices. Instead, it will try to use a simplex, which is essentially a triangle, to try to use points on the Minkowski difference to surround the origin. The simplex is built through a support function where given an angle; the function returns a point on the Minkowski difference that is based on a direction vector d. We can use this support function to build a simplex, then we check if the origin lies in the simplex, and if not, we can change points of the simplex until we are completely certain that the two objects are or are not intersecting.

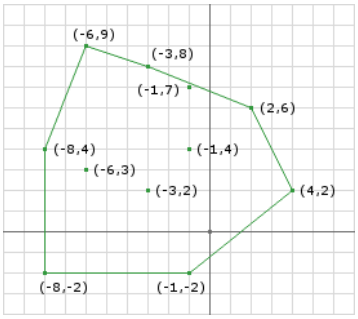

Figure 2: Minkowski difference of the two intersecting shapes [1]

Figure 3 is a screenshot of my GJK collision detection prototype in action, where on the left the shapes aren't colliding, and the program states as such, and on the right, they are colliding, with the program correctly determining those results.

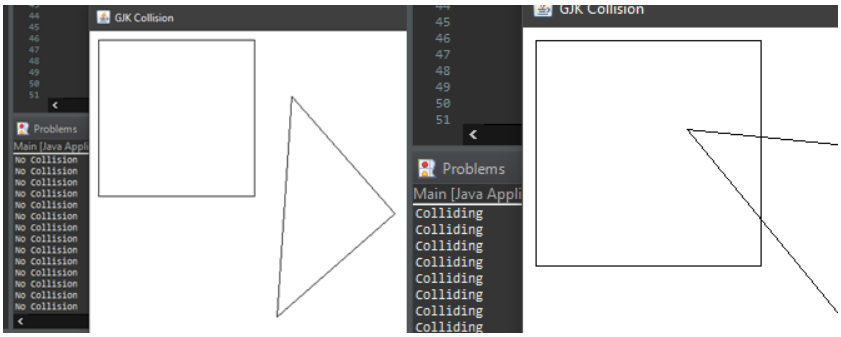

Figure 3: Screenshot of my prototype of the GJK collision algorithm in action

With the GJK algorithm working, we can build upon this to gather some important details about the collision that has occurred. For physics calculations, the point of intersection needs to be known when two objects are colliding. This likely will not occur on any specific frame of the simulation, so when an object is detected to be colliding, we need to know the exact point at the exact time that the two objects collided. Some additional useful information is the collision normal and collision depth, which tells the program the direction and minimum distance the object must move to no longer collide with the object. The GJK algorithm has an extension, known as the Expanding Polytope Algorithm (EPA) used to determine the aforementioned information that we need to calculate.

The EPA algorithm allows the calculation of the collision normal, and collision depth. This can be used to move both objects the minimal amount of distance, so that the two objects are no longer colliding. Additionally, it's critical to calculate this to calculate accurate physical responses. This is done by taking the simplex that we calculated, which surrounds the origin, to be expanded to a polygon with points of the Minkowski distance until the face with the shortest distance to the origin is found [2]. An additional point of information that is needed is the point of intersection, which is the exact point of collision on the surface of both objects at the very first instance during collision. This can be done by taking the closest face of the Minkowski difference with the intersection point, and computing the barycentric coordinates of that point. Using the points that make up the face, we can use the Barycentric coordinates to calculate the intersection point locally on both shapes [3]. With all this information, we can now calculate realistic physics responses for collisions.

Another important thing that we need to create realistic physics is choosing a numerical integrator to use. Once a physics state is resolved, applying everything, and physically moving each object might not seem as simple as it sounds. Computers deal with discrete steps, and we are trying to simulate a continuous system in these discrete steps. From one timestep to the next, these forces are still working, which would cause the simulation to deviate from a realistic result. There are two integrators that will be compared here, Euler integration and Runge-kutta 4 (RK4). Euler integration is the most well-known, you add your acceleration to the velocity, then add the velocity to the position. This does not try to compensate. RK4 on the other hand does some clever math, and recalculates the simulation in between timesteps, to produce a result which is one thousand times more accurate than a timestep of Euler.

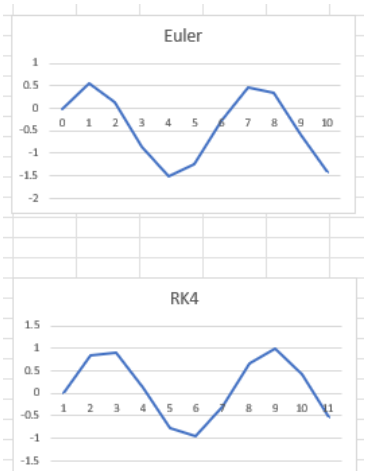

Figure 4: Euler vs. RK4 Experiment. 10 Timesteps

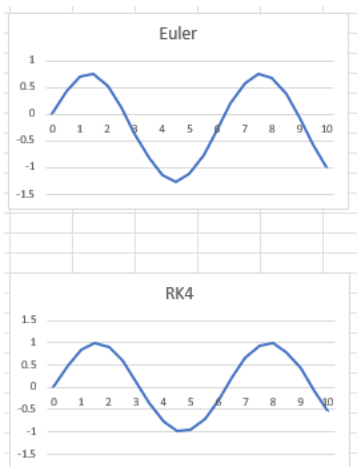

Figure 5: Euler vs. RK4 Experiment. 20 Timesteps

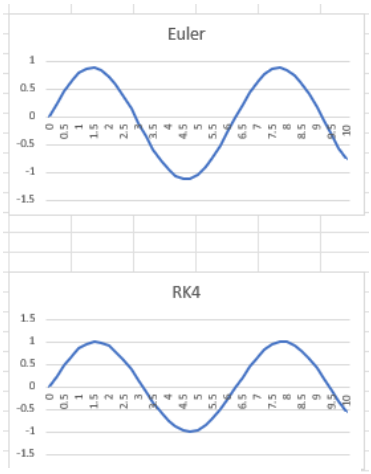

Figure 6: Euler vs. RK4 Experiment. 40 Timesteps

I decided to compare the two, writing a program which uses both methods to try to estimate the function sin(x) in discrete timesteps. I ran three trials of the experiment, increasing the amount of timesteps, which in both cases, will produce more accurate results. In the images above, are the resulting estimations of both Euler and RK4 in increasing timesteps. When looking at the results, it is apparent that RK4 is producing substantially better estimations than Euler, it is most prevalent in the peaks of the sine function, as the Euler estimation consistently undershoots the peaks, as they should be at heights of 1 and -1 respectively. Now, when deciding which to use, it's important to consider the added complexity of implementing RK4 compared to the simple implemention of Euler. While RK4 is substantially more accurate, a system needs to be constructed which runs the simulation more than once at different time steps, and averaging the results together with different weights. Due to time constraints, I will be proceeding with Euler's integration, as it will still produce relatively decent results, especially if the timestep count is increased. This will result in a performance loss, but the application's job isn't to be efficient or even real-time.

After this, I decided to attempt to make a larger 2D physics test, incorporating many physics objects, with collision detection and linear physics. I used the collision algorithm to get the relevant information, and calculated the resulting physics impulse with the following formula:

$$F_1 = \quad collisionNormal * collisionDepth \ / \ 2$$
$$F_2 = -collisionNormal * collisionDepth \ / \ 2$$

This is a crude approximation for applying forces for linear physics, where F_1 and F_2 are are forces applied to the both intersecting objects respectively. The F_2 force is being negated to apply an equal and opposite force to the second colliding object. This produces semi-realistic linear physics.

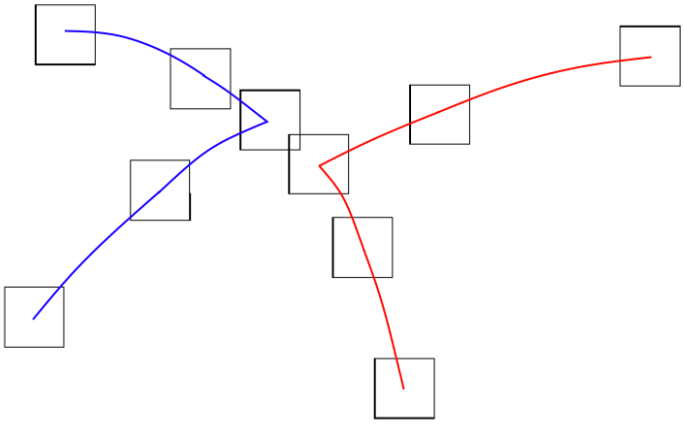

Figure 7: 2D Linear Physics Simulation Result

Figure 7 shows en example of that simulation, where I have combined multiple screenshots of the simulation of two colliding cubes, and tracing their path. Since this only implements linear physics, the objects do not rotate in a realistic way after the collision like the should, however the results of the linear physics are promising. After completing this, I was having issues attempting to calculate the intersection point needed for rotational physics, so I dedided to move on to 3D.

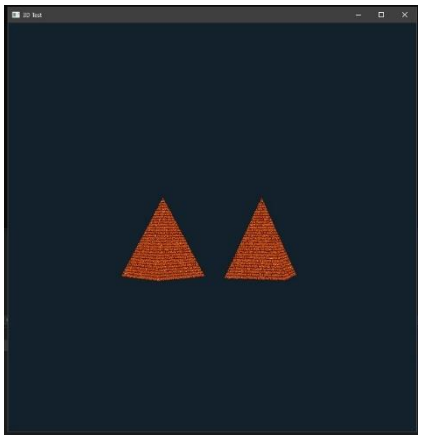

Figure 8: 3D Rendering Complete

While the previous tests were implemented in Java, for the final 3-dimensional project, I decided to go with C++ and OpenGL, as the performance loss with Java wouldn't be favorable, and I am already familiar with graphics in OpenGL, so it only made sense to switch to C++. After a bit of assistance from an online source [4], I was able to get a 3D application going, able to render various meshes to the screen, and with texture support. Figure 8 shows a screenshot after achieving 3D rendering, as well as abstracting out everything to be able to easily add, manipulate, and remove as many objects as I wanted in a 3D scene. The 3D rendering was accomplished by taking three dimensional vertices and indices data and putting this information through three transformations. The three transformations were the model transformation, which took the local vertices information, along with positional and rotational data, and converted the local vertex locations into global vertex locations. After that the global vertices were applied by a view transformation. This transformation converts the global information that we calculated into camera space, which are locations in the world in respect to the camera's position, and direction. This allows the user to move about the world in a camera class that will be created. The final transformation is the perspective transformation, which takes the camera coordinates, and uses them to calculate two-dimensional normalized device coordinates. Unlike an orthographic view, a perspective view gives the scene an apparent depth, where objects further away from the camera appear smaller in relative size than objects closer to the camera. In addition, perspective transformation can also take place in a field of view, which defines how narrow or wide the camera is viewing the world.

Before moving on to simulating any physics, a lot of architecture needs to be setup in order to be able to create, manage, bake, and replay simulations. The first main issue is being able to create

simulations in the first place. This could be hard coded, but this is tedius and wouldn't be available to an end user. There needs to be a way for the user to create various shapes, place them in the world, and a way to mark which objects should be locked in place and which should be simulated. This would involve a lot of work creating a 3D editor where the user can place pre-made shapes in a 3D world to be simulated. Instead, I opted to use a popular 3D editor to construct the simulation, known as Blender. Blender is primarily a 3D modeling and animation software. It has all the tools we need in order to create various simulation worlds. The main problem is figuring out how to translate the 3D blender scene to a scene in the physics application. Models and scenes can be exported from blender using various file formats, I looked through a bunch of them and determined that the wavefront OBJ (.obj) file format would fit the best, it contains the vertex and index data neatly grouped together by object with simple prefixes. Additionally, each object has a name associated with it, which can be used to translate physics specific information with. In order to read this information into the application, a .obj parser needed to be created. The parser works in several stages, reading in the important data necessary for the simulation. The parser detects what data it's reading in by the first character of each line. If the characrter is an 'o', the parser will read in the name of the object, if it's a 'v' the parser will read in the vertex information, and if the character is an 'f' the parser will read in the index information. Whenever the initial character changes from one to another, the program knows that the stage corresponding to the previous character is complete, and it should switch to the next block of information. After reaching the index information, it will check when the next piece of information is an o, signaling that the previous object's data has been compeltely parsed. The object is then created and the process repeats for the next object. This continues until the entire file is processed, reading in the entire simulation.

```
o Cube
v -11.583015 -61.257225 11.583015
v -11.583015 -5.463013 11.583015
v -11.583015 -61.257225 -11.583015
v -11.583015 -5.463013 -11.583015
v 11.583015 -61.257225 11.583015
v 11.583015 -5.463013 11.583015
v 11.583015 -61.257225 -11.583015
v 11.583015 -5.463013 -11.583015
s 0
f 1 2 4 3
f 3 4 8 7
f 7 8 6 5
f 5 6 2 1
f 3 7 5 1
f 8 4 2 6
```

Figure 9: An example of how the .obj file format stores an objects vertex and index data.

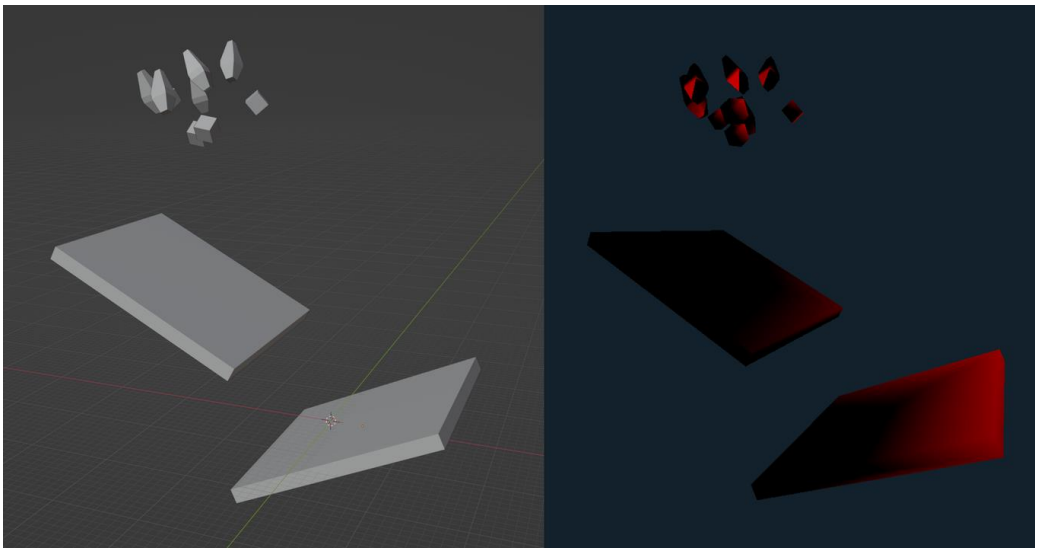

Figure 10: Parser loading a scene from blender (left), into the physics simulator (right).

The file itself is opened through the application through a windows call. The code, modified from Microsoft's official documentation [5], configures the file editor to only open .obj files. It specifies where the file path should be stored and ensures that the selected file and path that it returns are valid. Once the file is selected, the application will then load the simulation described previously.

In addition to loading simulations, we need a way to switch between various "scenes", for the application, there are three separate scenes that are needed. A main menu to serve as an entry point, a simulation scene which loads in a simulation, and can play out the simulation. Additionally, a scene is needed to load and playback baked simulation files. This will be gone into more detail, but the main idea is that the user can precompute a computationally intensive simulation to a file, which can be loaded back into the application to play back a complicated simulation in real time. These three scenes all need to be able to perform updates and draw to the screen. To solve this issue, I created a scene object which contained various functions that all scenes need or would be useful for. The update and render functions can be overridden by other objects which inherit the scene class. Objects which inherit this class would be the three scenes that were previously discussed. Switching between scenes is as easy as calling a function with the pointer to a specific scene. This pointer is set to a generic scene pointer, and when this scene calls the update, render, or other overridden function. The child's implementation of the function is instead called.

```
class Scene {

public:

    ImVec4 clear_color = ImVec4(0.45f, 0.55f, 0.60f, 1.00f);
    Shader* defaultShader;

    bool doPhysicsOnUpdate = true;

    std::vector<Object> objects;

    Scene();

    virtual void InitializeScene();
    virtual void OnSceneSwitch();
    virtual void Update(float dt);
    virtual void Render();

    int AddObject(glm::vec3 position, glm::vec3 rotation, GLfloat* meshData,
                  int meshSize, GLuint* indicesData, int indicesSize, Shader* shader);
    int AddDefaultObject(DefaultObjectType objectType, glm::vec3 position,
                         glm::vec3 rotation, Shader* shader);

    virtual void Cleanup();

};
```

Figure 10: Structure of parent scene class.

The final addition that is needed is a way to control the simulation, as well as to trigger file selection from the application. Essentially, we need a simple user interface to perform these functions. While programming a UI system for OpenGL would be an option, there is a well-established UI system known as "Dear ImGui" which is a "bloat-free graphical user interface for C++" [6], it's a performant and easy to use library to perform this function. Using the library involved downloading its most recent release and moving its relevant C++ and header files into the program, making sure to add the files which provided support for OpenGL and GLFW (an OpenGL Framework) [7]. After some setup, I was able to add some various buttons responsible for playing, stopping, resetting, and initiating baking for the simulation scene. Figure 11 is a screenshot of that very UI. The other two scene UIs contain functionality to open .obj and .bake files, and the ability to play, stop, and reset the playback of baked simulation files. This is just about everything that is needed, we can now build, load, and control any physics simulation scene that is supported.

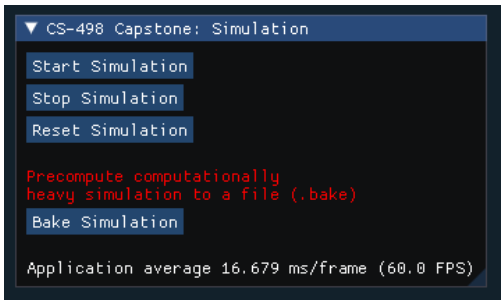

Figure 11: Screenshot of the simulation scene UI.

To begin simulating realistic physics, we need to recreate the GJK-EPA collision detection algorithm in 3D. All vertex data that was given to the algorithm need to be changed to 3 dimensions, and instead of the GJK algorithm trying to surround the origin with a two-dimensional triangle as the simplex, it instead needs to use a three-dimensional tetrahedron instead. For the EPA portion of the algorithm, instead of extending a triangle simplex to find the closest edge to the origin, it needs to instead extend a tetrahedron simplex to find the closest face to the origin. The process for determining these results is extremely similar, but due to time constraints I used an already implemented version of the GJK algorithm I found on GitHub [8]. While the program was implemented in C++, various changes had to be made as the author of the algorithm did not use the glm (OpenGL Mathematics) library [9] for their vector and matrix math. Figure 12 shows the implemented 3D collision detection in action, where two bouncing cubes are periodically intersecting each other, with the output displayed.

Figure 12: Screenshots of the 3D GJK collision detection algorithm in action.

Now that collision detection is implemented, the next step is actual physics. Specifically, linear physics. There are two steps that are needed to make this happen. When the simulation update method is called, it loops through every object and calls the resolve function. The resolve function will detect if any two objects in the scene are colliding. For each of these collisions, the object's position will be altered based on the collision normal and collision depth, which as discussed earlier, is the direction and distance an object must move to no longer be colliding. This distance is the shortest distance possible for the object to move. So, to calculate the new positions the following equations are used.

$$\Delta A = \quad collisionNormal * collisionDepth \,/\, 2$$
$$\Delta B = -collisionNormal * collisionDepth \,/\, 2$$

ΔA is the change of position of the first object, and ΔB is the change of position of the second object. Instead of moving one object the entire distance, moving both objects half the distance in opposite directions allows the least impact on the simulation. Once this is done, we can now apply the forces that occurred during the simulation. They both calculated with the following formulas [10].

$$F_A = \quad \frac{j}{M_A} n$$

$$F_B = -\frac{j}{M_B} n$$

F_A is the force to be applied to the first colliding object, and F_B is the force to be applied to the second colliding object. M_A and M_B are the masses of both objects (which will just be 1), and n is the collision normal.

j is the impulse magnitude which is defined as follows [10]:

$$j = \frac{-(1+e)v_{AB} \cdot n}{n \cdot n\left(\frac{1}{M_A} + \frac{1}{M_B}\right)}$$

Where e is restitution (elasticity), or how bouncy the object is (ranged 0 – 1), v_{AB} is the relative velocity of the two objects, ($v_{AB} = v_A - v_B$), and n, M_A and M_B are the same as defined above. The impulse magnitude determines the strength of the forces applied to both objects, a higher value of j and the stronger the forces are. To close out the resolve section of linear physics, every object will have a downward gravity force applied to it. It is also important to note that any anchored object can have forces applied to it, but it will remain stationary.

The second and final step is integrating the velocities. As stated earlier in the Euler vs. RK4 section, I will be using Euler's integration method due to time constraints. After adding all the applied forces together, that final cumulative force will be multiplied by a delta time value and added to the velocity. The velocity will then be multiplied by that same delta time value and added to the object's position. It's worth noting that the delta time value is just the time since the previous frame. This is usually used in game engines to give a similar simulation result regardless of if a computer has a high framerate, or a low framerate. However, for this usage, the delta time will be constant, and the framerate will be limited based on the delta time, dropping if calculating the simulation takes too long. This ensures a more consistent result regardless of device. Note that this does not mean the simulation is entirely deterministic, but it gets closer to it. Figure 13 shows an example simulation, with screenshots at four points of time during the simulation. Objects in the air get pulled by gravity and collide with the anchored object which acts like a ramp. The objects slide off and collide with the second anchored ramp, where they splash outward, sliding down the ramp, then falling for eternity.

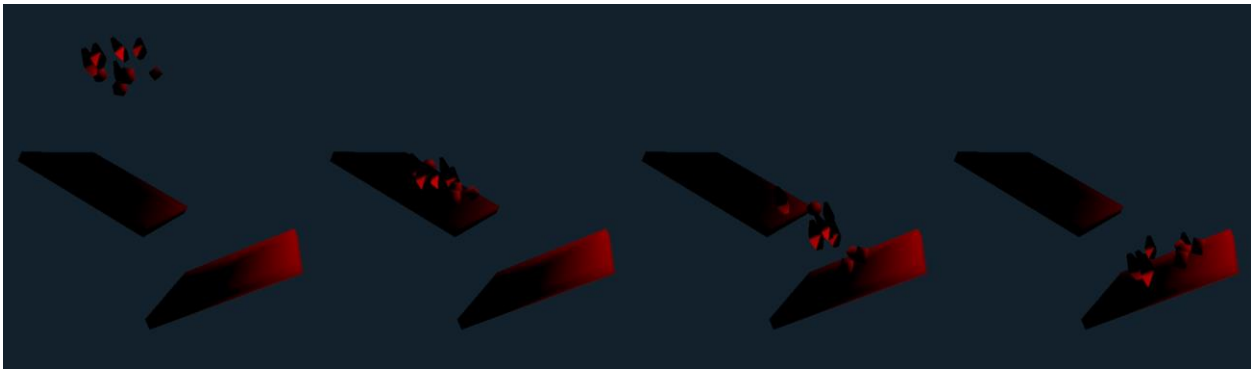

Figure 13: Linear physics simulation.

The simulation showcased in Figure 13 can easily be simulated in real time. However, as you add more and more objects, the amount of time that it takes to compute the simulation increases significantly. To watch these complicated simulations in real-time, the program will "bake" the results of the simulation to a file. The file contains the original .obj file, as well as positional data for each object. When a baked simulation file (.bake) is loaded in, the program first goes through the process of loading in the wavefront obj portion, then it loads in the simulation data. The parser knows when to switch from loading the model data when it encounters a '!' symbol in the first character of a line. This doesn't appear in .obj files, so there won't be any issues with going to the next stage prematurely. Figure 14 shows that structure of the .bake file, right at the moment when it switches from the .obj information to the positional data. The first character in the positional data (bottom half of figure 14), is an ID or index for an object, it coincides with a C++ vector (dynamic array), where the index points to the coinciding object. The following three values represent the xyz coordinates of the object's position. In this case, the object remained the saime likely because it was anchored. However for objects that are physically simulated, they will be different values.

```
f 4009 4011 4015 4013
f 4004 4002 4010 4012
f 4001 4003 4011 4009
f 4003 4004 4012 4011
f 4013 4015 4016 4014
f 4011 4012 4016 4015
f 4010 4009 4013 4014
f 4012 4010 4014 4016
!
0 -17.6277 34.6123 -11.3522
0 -17.6277 34.6123 -11.3522
0 -17.6277 34.6123 -11.3522
0 -17.6277 34.6123 -11.3522
0 -17.6277 34.6123 -11.3522
0 -17.6277 34.6123 -11.3522
0 -17.6277 34.6123 -11.3522
0 -17.6277 34.6123 -11.3522
```

Figure 14: Structure of .bake file.

In order to demstrate the .bake feature in action, I decided to make an extremely large simulation to compute. The simulation has 642 separate objects situated in a simular simulation as in Figure 13. Figure 15 is a screenshot of the simulation loaded into the program, ready to bake to a file. The simulation took roughly twelve minutes to complete on my Intel 10th generation i7, and the file size of the .bake file is rouhgly 105 megabytes. Figure 16 was taken after the simulation completed. It's the simulation being played back after falling off of the first ramp onto the second. Some of the objects fly off in all sorts of directions, but most fight the second incline upward and all fall off the edge at the top. The baked simulation ran in realtime at roughly 120 frames per second. A link to the video of the large simulation can be found here: https://www.youtube.com/watch?v=bgIRZiJN8EI

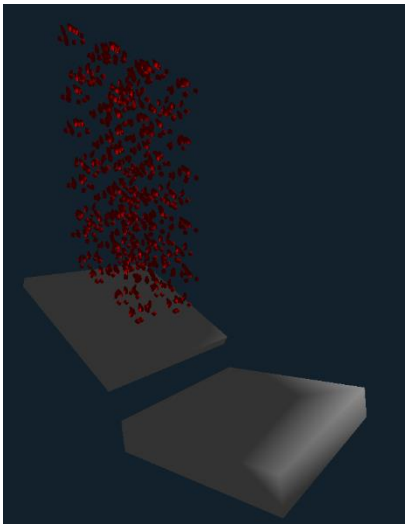

Figure 15: Large simulation initial scene

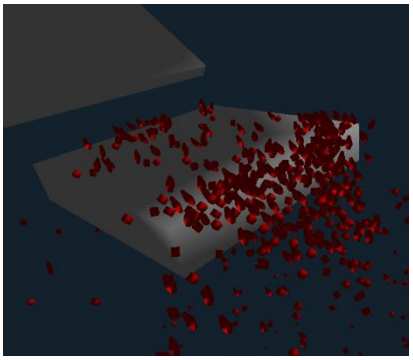

Figure 16: Large simulation mid-playback

Results

This section will go over the process from creating the simulation on blender, to finally having a precomputed .bake file. This requires some basic Blender skills but it's overall a quick and easy process. Before starting, make sure everything is installed and works properly.

Start by opening Blender. Once opened, the default world should be seen. select each object in the scene, the camera, light, and cube by left or right clicking, and press the delete key to delete each item, they're not needed. Once you have deleted everything, your blender application should look like figure 17. This is where the simulation will be created. Additionally, there are two main points you should keep in mind while building your simulation. The first point is to ensure that every shape you create is convex.

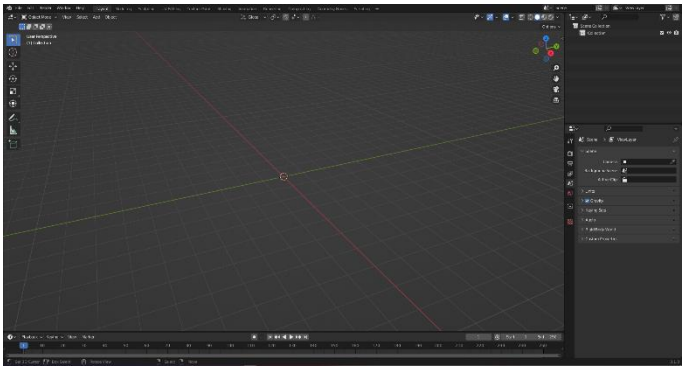

Figure 17: An empty blender file.

Convex shapes have all interior angles less than 180 degrees. If your shape is concave, you can draw a line between two points of that shape, and one of these lines will at some point exit the shape's boundary [11]. Shapes like cubes, diamonds, and spheres are convex, but a torus (donut shape), is concave. The other point is to not create what are known as n-gons. An n-gon is a shape in blender which contains five or more vertices [12]. The physics application can't handle this, so make sure to only produce triangles or quads.

In order to begin creating a scene, press shift + A to open the add menu, under the mesh tab, you can select various starting objects to create. Once you select an object, that object will be created at your cursor location, which a point in 3D space when can be set by left or right clicking. Figure 18 shows the add menu and the various options for meshes. You can select these objects and move them around in various ways. After selecting an object, press G and move your mouse to move the object around in 3D space. Similarly, R will rotate the object, and S will scale the object. By pressing tab after selecting an object you can enter edit move, here you can use similar processes to manipulate points on the mesh itself. This isn't a Blender course, so if you want to look more into this then there are plenty of tutorials online which go over these topics in further depth.

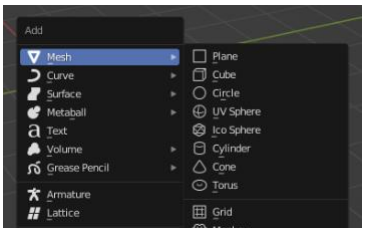

Figure 18: Blender Add menu.

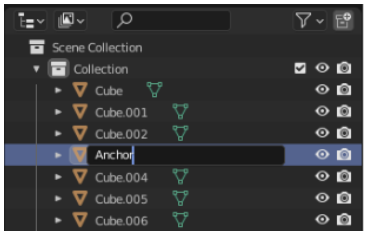

Figure 19: Blender outliner and renaming an object to "Anchor".

Once your happy with your simulation scene, there are a couple of steps to take to export it into the .obj file. Firstly, identify what shapes you want to act as anchors, which will be locked to it's current position and can act as a floor, ramp, obstacle, etc. One at a time, select an object that you want to be an anchor. On the top right of the screen there is the outliner menu. Scroll up or down in the menu and find the highlighted object. Double-click on it's name and rename it to exactly "Anchor", and press enter. If the name automatically changes to something like "Anchor.001", this is fine, the physics application will account for this. Figure 19 shows the outliner with various objects and how you should rename an object to make an anchor. Make sure to repeat the previous process for every object you want to act as an anchor. When you're ready to export, go to the top left of the screen and select, File > Export > Wavefront OBJ (.obj). On the right side menu in the popup, under geometry export, deselect all the checked options. Figure 20 shows how that menu should look. After completing that, export the .obj file at your preffered location and filename.

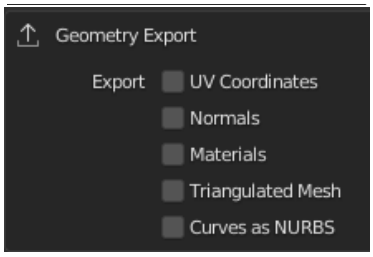

Figure 20: Geometry export menu.

Open the physics application. There will be a simple menu with two buttons, press the “Open Simulation Scene (*.obj)” button. A file selector will open and prompt you to select a .obj file. Choose your created simulation file and open the file. This will automatically load the file and take you to the simulator scene of the application. Pressing start stop and reset buttons play the simulation, pause the simulation, and bring the simulation back to its original state respectively. If a scene is too complicated and is causing lag while simulating, press the bake simulation button. This will automatically reset and start playing the simulation, recording all the positional information. This will continue untill you press the stop button, when it will save the file at the exact file path with the same file name. The only difference is that the file extension is .bake instead of .obj.

Once you have created your .bake file, close and reopen the application. Press the “Open Baked Simulation (*.bake) and chose your .bake file. A similar menu will open with a start, stop, and reset simulation buttion. They perform exactly the same as the previous buttons in the simulation scene.

Discussion

Overall, throughout the working on this project, I learned a lot more about how rigid body physics is simulated on computers. Specifically, learning about physics integrators, specifically RK4, was extremely interesting as before I came to this project, I didn’t know that integrating various velocities was more complicated than just adding the velocity to the position. Additionally, I learned a lot about three-dimensional rendering, how to convert a 3D scene into a 2D image. Some smaller additional things that I learned was how to call a file opener through C++, how to parse a .obj file, and methods on more complicated collision detection, specifically GJK and SAT.

A lot went right with the project and quite a bit went wrong. Most of the goals that I set out to do are accomplished. However, with all that I managed to accomplish I am still missing rotational physics, which is one of the main things I wanted to figure out how to do. Lots of information was available for projects like this, but there wasn’t one main source that allowed me to finish the project. Most sources had bits and pieces of useful information, and I had to take all of it to combine it to get a working result. The simulation overall was a bit buggy, some of the object’s glitch into anchored floors or ramps and start spazzing out. However, that was to be expected, it takes a very long time to be able to create a relatively stable physics engine.

If I was to start again, I would take more time finalizing the two-dimensional simulations, after completing linear physics for 2D, I jumped straight to 3D thinking I could figure out the rotational aspect later. Additionally, I would have managed my time a bit better, I found myself having large gaps where I didn’t work on the project, followed by very long marathons of work. Spreading it out more evenly throughout the semester would have been more beneficial.

Conclusion and Future Work

Overall, I managed to get a working, three-dimensional, linear physics simulation. If I was to continue working on the project, I would start by finally figuring out the rotational physics, being able to implement that would bring this project to a more complete finish. Some additional features that I would implement are a way to adjust parameters about specific objects in blender such as mass, friction, and restitution. This could be easily done, but I wasn’t able to fit it in with more high priority features. I did really enjoy working on this project though, it has been something that I have been wanting to do for years.

References

[1] W. Bittle, “GJK (Gilbert–Johnson–Keerthi),” *dyn4j*, Apr. 13, 2010. https://dyn4j.org/2010/04/gjk-gilbert-johnson-keerthi/

[2] “EPA: Collision response algorithm for 2D/3D.” https://blog.winter.dev/2020/epa-algorithm/

[3] A. Chou and A. Chou, “Game Physics: Contact Generation – EPA | Ming-Lun ‘Allen’ Chou | 周明倫,” *Ming-Lun “Allen” Chou | 周明倫*, May 2018, [Online]. Available: https://allenchou.net/2013/12/game-physics-contact-generation-epa/

[4] freeCodeCamp.org, “OpenGL Course - Create 3D and 2D Graphics With C++,” *YouTube*. Apr. 27, 2021. [Online]. Available: https://www.youtube.com/watch?v=45MIykWJ-C4

[5] Microsoft, “Using Common Dialog Boxes - Win32 apps,” *Microsoft Learn*, Sep. 21, 2021. https://learn.microsoft.com/en-us/windows/win32/dlgbox/using-common-dialog-boxes

[6] Ocornut, “GitHub - ocornut/imgui: Dear ImGui: Bloat-free Graphical User interface for C++ with minimal dependencies,” *GitHub*. https://github.com/ocornut/imgui

[7] *GLFW*, Jul. 22, 2022. https://www.glfw.org/

[8] Kevinmoran, “GitHub - kevinmoran/GJK: Basic 3D collision detection implementation using the Gilbert–Johnson–Keerthi distance algorithm along with the Expanding Polytope Algorithm,” *GitHub*. https://github.com/kevinmoran/GJK

[9] G-Truc, “GitHub - g-truc/glm: OpenGL Mathematics (GLM),” *GitHub*. https://github.com/g-truc/glm

[10] C. Hecker “Rigid Body Dynamicshttp://www.chrishecker.com/Rigid_Body_Dynamics

[11] Math Monks, “Convex and Concave Polygons - Definition, Differences, Examples,” *Math Monks*, Mar. 28, 2023. https://mathmonks.com/polygon/convex-and-concave-polygons

[12] D. Paterson, “Blender N-gon guide,” *Artisticrender.com*, Jan. 2022, [Online]. Available: https://artisticrender.com/blender-n-gon-guide/

[13] Reducible, “A Strange But Elegant Approach to a Surprisingly Hard Problem (GJK Algorithm),” *YouTube*. Mar. 24, 2021. [Online]. Available: https://www.youtube.com/watch?v=ajv46BSqcK4

[14] LearnChemE, “4th-Order Runge Kutta Method for ODEs,” *YouTube*. Apr. 23, 2015. [Online]. Available: https://www.youtube.com/watch?v=1YZnic1Ug9g

[15] N. Souto, “Video Game Physics Tutorial - Part I: An Introduction to Rigid Body Dynamics,” *Toptal Engineering Blog*, Jan. 2015, [Online]. Available: https://www.toptal.com/game/video-game-physics-part-i-an-introduction-to-rigid-body-dynamics

Appendix

Instructions to download software:

- Download Blender from: https://www.blender.org

- Download Microsoft Visual Studio from: https://visualstudio.microsoft.com

- Download everything from the project GitHub. https://github.com/perilldj/CS-498-RigidBodyPhysics

- Open the “Project1.sln” file.

- For the best performance, make sure that you are on Release mode rather than Debug mode.

- Run Program.