**MTRXEngine: An Educational Physics Engine**

Comp 4905 – Honours Project

Oliver Van Kaick

Mohamed Kazma

101019719

December 7th, 2019

# Abstract

The sole purpose of this project is to get a better understanding of the systems and features that a physics engine is supposed to provide, and to provide other students with an educational source when writing their own physics engine. This report will mainly expand upon the purpose of the project, the methodology used to complete said project, as well as the hurdles and knowledge extracted from this endeavor. Documentation of the engine is generated in html and pdf format from latex code using “Doxygen” and is located under the “documentation/” directory of the submission. For that reason, this report will not go very much in-depth into implementation details. It will however, discuss upon the architectural decisions made, the high-level systems and how they are supposed to operate in real-time and how all of that adds up together. Thus, I highly recommend looking at the engine documentation for more information on the implementation details.

Table of Contents

[Abstract 2](#_Toc26913446)

[Introduction 5](#_Toc26913447)

[Summary of Research Question 5](#_Toc26913448)

[Motivation 5](#_Toc26913449)

[High-Level Architecture 6](#_Toc26913450)

[Rigid Body System 6](#_Toc26913451)

[Rigid Body Dynamics 6](#_Toc26913452)

[Force generators 7](#_Toc26913453)

[Rigid Body Manager 8](#_Toc26913454)

[Collision Detection System 8](#_Toc26913455)

[Colliders and collider types 8](#_Toc26913456)

[Sphere Colliders 9](#_Toc26913457)

[Capsule Colliders 9](#_Toc26913458)

[Axis Aligned Bounding Box Colliders 9](#_Toc26913459)

[Oriented Bounding Box Colliders 9](#_Toc26913460)

[Convex Shape Colliders 9](#_Toc26913461)

[Collision Detection and Collision Utilities 10](#_Toc26913462)

[Sphere Sphere Collision Detection 10](#_Toc26913463)

[Sphere Capsule Collision Detection 10](#_Toc26913464)

[Sphere AABB Collision Detection 10](#_Toc26913465)

[Sphere OOBB Collision Detection 10](#_Toc26913466)

[Sphere Convex Shape Collision Detection 10](#_Toc26913467)

[Capsule Capsule Collision Detection 10](#_Toc26913468)

[Capsule AABB Collision Detection 10](#_Toc26913469)

[Capsule OOBB Collision Detection 11](#_Toc26913470)

[Capsule Convex Shape Collision Detection 11](#_Toc26913471)

[AABB AABB Collision Detection 11](#_Toc26913472)

[AABB OOBB Collision Detection 11](#_Toc26913473)

[AABB Convex Shape Collision Detection 11](#_Toc26913474)

[OOBB OOBB Collision Detection 11](#_Toc26913475)

[OOBB Convex Shape Collision Detection 11](#_Toc26913476)

[Convex Shape Convex Shape Collision Detection 11](#_Toc26913477)

[GJK Collision detection algorithm 12](#_Toc26913478)

[Raycasting Utilities 13](#_Toc26913479)

[Ray Sphere Collision Detection 13](#_Toc26913480)

[Ray Capsule Collision Detection 13](#_Toc26913481)

[Ray AABB Collision Detection 13](#_Toc26913482)

[Ray OOBB and Convex Shape Collision Detection 13](#_Toc26913483)

[BVH and other optimizations 13](#_Toc26913484)

[Collision System 14](#_Toc26913485)

[Demos (Renderer) 14](#_Toc26913486)

[Projectile Demo 14](#_Toc26913487)

[Collision Demo 14](#_Toc26913488)

[Buoyancy Demo 15](#_Toc26913489)

[Fluid Simulation Demo 15](#_Toc26913490)

[Future of the Engine 16](#_Toc26913491)

[Collision Resolution Systems 16](#_Toc26913492)

[Contact calculations 16](#_Toc26913493)

[Continuous Collision Detection systems 16](#_Toc26913494)

[Soft bodies 17](#_Toc26913495)

[Conclusion 17](#_Toc26913496)

[References 18](#_Toc26913497)

ADD A Screenshots of the engine

# Introduction

## Summary of Research Question

The project included the writing of a physics engine from scratch using C++ and other libraries such as GLM GLAD SPDLOG GLFW etc. This engine is written for the purpose of documenting the process of writing a physics engine, the barrier of entry (the difficulty, availability of resources), as well as the amount of effort/time required to achieve a decent enough engine that can be used by Real-Time games in any meaningful manner. In addition, the resulting engine will also be used as another source or a starting point for people interested in the field of Real-Time physics. For this reason, the engine will have to thoroughly documented and written with readability and user-friendliness in mind rather than extreme optimization and performance. Furthermore, as my project is only an entry point into physics programming and is used to decrease the barrier of entry for students interested in the subject matter, I will mention the resources I have used to write the current codebase as well as other resources that I have and will be using to further iterate on the project and add new features to it.

## Motivation

My motivation behind the engine is my own desire to write a physics engine and finding it to be a very daunting task. That is, due to the lack of proper resources on the subject, not many existing engines that are open-source to be used as inspiration and the lack of interest and difficulty of the subject material. Although physics makes a lot of games tick, it is not the most interesting subject that a gamer or game developer would want to go into and for that reason it is a shunned as a topic of research or only a select few of non-open source engines like PhysX, Havok, and others are used instead.

# High-Level Architecture

A basic physics engine would need to at least have 2 basic systems, A Rigid body system and a collision system. The rigid body system would mainly handle all rigid bodies created within the simulated and update them in Real-Time using some Update ticks. The Collision system would do something similar by handling colliders and trying to report collisions and sometimes resolving those collisions. A graphical representation of the architecture of the engine is in the html of the generated documentation in the class hierarchy section of the website.

# Rigid Body System

## Rigid Body Dynamics

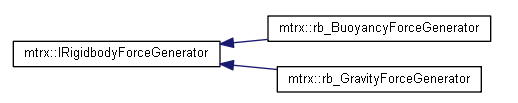


Rigid bodies are objects that remain solid and do not have any deformation applied to them or are too small for it to apply to them. Thus, we can say any point on the rigid body will remain the same distance away throughout the entirety of the simulation.

To simulate this behavior, I implement Newtonian physics integration by providing a force addition API and a force accumulator for all rigid bodies either at a certain a point in the rigid bodies’ space or on its center of gravity. These 2 applications differ as a force applied on its center of gravity does not generate a rotational force (torque forces). However, the one applied at a point on the rigid body will generate a torque that will then be integrated into a rotational force that will be added to the object’s orientation. So, mainly the way we interact with rigid bodies is via impulse forces and having them integrated using the Newtonian physics formulas that we know of. This also brings up the issue with forces dying out as we do need to simulate drag and air resistance and its effect on the velocity of the rigid body. I have opted into just using a float value to simulate a coefficient upon which velocity is lost as a function of time. There are many ways that this value is integrated into the system however, using it as a coefficient of the velocity value gives us a more than reasonable result while being extremely inexpensive an operation to perform.

Furthermore, we can also very similarly talk about rotational forces and torques that forces/impulses can generate by also integrating angular torques into angular accelerations and then velocities etc. However, it is a more complicated process than adding up linear forces. With rotations, we will need an inertia tensor matrix. An inertia tensor matrix defines some values about the rigid body’s shape that affect the way rotational forces are calculated and thus applied. For instance, a cuboid shaped rigid body uses an inertia tensor matrix that is different than that of a spherical rigid body. This matrix can be cached on creation and applied to the accumulated torque to generate angular accelerations which are then integrated in the same manner as linear forces. So, we can also apply drag on rotational forces in the same manner as before by multiplying our float value by the angular velocity in the same manner as before.

## Force generators



Simulating forces on rigid bodies cannot be done within the context of a rigid body itself. We want to provide users with a user-friendly choice to add forces to rigid bodies to simulate realistic or maybe unrealistic phenomena. Thus, I added in a force generation interface that receives a rigid body, a value for the time elapsed since the last frame and applies a certain force to a rigid body. This is a decoupled and scalable process of just adding a force as a component within a component system and having the rigid body handle the applied forces using the aforementioned rigid body integration interface. The main force generators implemented are gravity, drag, and buoyancy generators. However, we can add as many of them as we would want. There are some untested generators for different spring couplings in engine which can also show how scalable this interface can be!

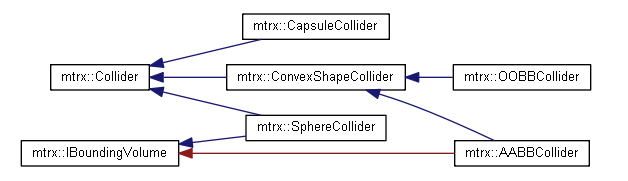
Now that we have a nice interface for adding forces, we need a way to hold a collection of rigid bodies and the force generators that apply to them. This has led to the creation of the rigid body force registry used for holding all the force generators that apply to a certain rigid body and an interface upon which we can apply the forces to said rigid body.

## Rigid Body Manager

Now that most of the puzzle pieces are fitting together, we just need to finish the puzzle by creating the manager that would handle this entire process. As most of the classes implemented are mostly self-sufficient, the amount of work that the rigid body manager needs to do is not too insurmountable. All that needs to be done is the manager needs to Update all of the force generator registries for each rigid body and integrate the result of the modified rigid body to achieve what we would call a rigid body update tick.

# Collision Detection System

## Colliders and collider types



Collision detection and resolution has been a pretty large subject of research in physics engines and games generally. We cannot check for collision between meshes at this fine grain of a level as it would be extremely expensive. Since we need to simulate collision between a very large amount of entities, we need to find a better approximation. Thus, we use entities known as bounding volumes or colliders. A bounding volume defines a certain simple or simpler object in space that we determine to be close enough to the original shape that we can use to check for collisions that are close enough and are extremely fast to check for collisions due to some characteristics and assumptions that we can make about these shapes that lend themselves to collision detection.

### Sphere Colliders

We use sphere colliders as they are the easiest to check for collision and require very little information to store. However, they are one of the most inaccurate colliders to use. You can generally define a sphere collider by a center and radius which is just 4 floating point values.

### Capsule Colliders

Capsule Colliders are better versions of a sphere collider as they can be thought of as a bunch of spheres defined over 2 center points and a line segment connecting them. The reason capsule colliders are very viable as colliders is that they can always be reduced to sphere colliders depending on the circumstance which makes them very performant colliders.

### Axis Aligned Bounding Box Colliders

Axis Aligned Bounding Box Colliders define a box collider that abides by world space’s axes. This assumption makes them very performant as we can make many assumptions about the collider.

### Oriented Bounding Box Colliders

Oriented Bounding Box Colliders are more expensive colliders they should be used sparingly as it is expensive to check for collision using them however, they are one of the more accurate of colliders you can generally use.

### Convex Shape Colliders

Are used for fine collision detection when accuracy is extremely important or very noticeable to users. It is expensive to calculate collision as we are defining an arbitrary convex shape and checking collision on them is an expensive process that I will talk about later on.

## Collision Detection and Collision Utilities

### Sphere Sphere Collision Detection

Checks the distance between the sphere centers and their radii

### Sphere Capsule Collision Detection

Finds the closest point on the capsule’s line segment to the center of the sphere and then just applying a sphere sphere collision detection

### Sphere AABB Collision Detection

Finds the closest point on the AABB from the sphere’s center and then does a sphere point distance check

### Sphere OOBB Collision Detection

Finds the closest point on the OOBB to the sphere and then does a sphere point distance check

### Sphere Convex Shape Collision Detection

Triangulate convex shape using fan triangulation and then check the minimum distance with the radius of the sphere

### Capsule Capsule Collision Detection

Find the minimum distance between the 2 line segments and check it with the radii of the capsules

### Capsule AABB Collision Detection

Based on this paper: <https://www.researchgate.net/publication/275025293_Collosion_Detection_Research_Based_on_Capsule_Bounding_Volume>

### Capsule OOBB Collision Detection

Checks for the minimum distance between the line segment and the triangles of the OOBB and check that distance with the radius of the capsule

### Capsule Convex Shape Collision Detection

Checks for the minimum distance between the line segment and the triangles of the convex shape and check that distance with the radius of the capsule

### AABB AABB Collision Detection

Check that the difference in center positions is less than the total of the half extents of the AABB

### AABB OOBB Collision Detection

Same as convex shape collision detection

## AABB Convex Shape Collision Detection

Same as convex shape collision detection

### OOBB OOBB Collision Detection

Same as convex shape collision detection

### OOBB Convex Shape Collision Detection

Same as convex shape collision detection

### Convex Shape Convex Shape Collision Detection

Check for collision using the GJK algorithm which will be mentioned in the next section

## GJK Collision detection algorithm

There are some occasions where would want to calculate finer collisions between objects in the physics simulation. Thus, we would need to use an algorithm powerful enough to give us collision information about any convex shape. This were GJK comes into the picture. It is an algorithm based upon a paper written by Gilbert, Johnson, and Keerthi generally used to find the distance between 2 convex shapes, however, it can be modified to be used as a collision detection algorithm. The main basis behind the algorithm is defining what we would call as Minkowski Space. The input of algorithm is 2 collections of vertices which are essentially just 3d points in world space. Finding the difference between any 2 points in each collection gives us a point in a completely different space which we call Minkowski Space and this process is called the Minkowski Sum. The main argument of this algorithm is that if 2 convex shapes collide and let’s say for simplicity’s sake collide at exactly a certain point in both shapes. The Minkowski Sum of these 2 vertices is the origin. This argument can strongly inform the approach we would need to take to check for collision. To make this argument more general we can easily conclude that if the convex hull of the minkowski space of the 2 vertex lists encapsulates the origin then these 2 shapes are colliding. The main issue with this approach is that it is very expensive and memory intensive. It runs in O(mn) time and O(mn) space assuming that m and n are the size of each list respectively. As that is not very doable in Real-Time, we would need a better solution to the issue. We can achieve that by realizing that we don’t need the entire convex hull to check whether it contains the origin within 3d space we just need 4 points as we can encapsulate the origin with a tetrahedron. We call this structure a Simplex, it is the minimum structure required to encapsulate a point in the space we are operating in. A line in 1d a triangle in 2d and tetrahedron in 3d. However, how would we find the correct tetrahedron to use? We would need an algorithm that can give us a point in Minkowski space that is farthest in a certain direction. This is known as a Support function. We can achieve this algorithm using the dot product. Now that we can filter in the points that we want, we can now see that we can iteratively build our simplex and modify it in a decently smart manner by modifying the direction of the Support function to match the direction that we want the simplex to grow to. It does also support early out functionality when the point is not within the direction that was given to the support, we can definitely be sure that a collision cannot occur.

Another bonus of the algorithm is that it lends itself to collision resolution using an algorithm called EPA (Expanding Polytope algorithm) which uses the simplex that was used in the collision as input and uses it to generate the minimum vector needed to move out of collision, however that is not a subject I will be talking about.

## Raycasting Utilities

### Ray Sphere Collision Detection

Find the minimum distance between sphere center and ray which is similar to a line segment and check it against the radius of the sphere

### Ray Capsule Collision Detection

Find the minimum distance between the line segment and the ray and check that distance with the radius of the capsule

### Ray AABB Collision Detection

Based on this website:

<https://tavianator.com/fast-branchless-raybounding-box-intersections/>

### Ray OOBB and Convex Shape Collision Detection

Can be done using GJK as a line is a convex shape.

## BVH and other optimizations

Collision detection querying is expensive to run at Real-Time as it is an O(n^2) algorithm to run as all colliders need to check collision with all other colliders. However, it is almost definitely the case that many of these colliders do not collide at this point in time and would want to remove these colliders from being checked as it would be a waste of time to do so. This were many spatial optimizations can be introduced to optimize the volumes that we are searching. There are many optimizations that can be done (Quad Trees, Oct Trees, BSPs etc.). However, we will be talking about Bounding Volume Hierarchies. As the name states, it is essentially a hierarchy of bounding volumes. As we know that generally a hierarchy traversal is an O(logn) search in many situations we can then make this traversal into an O(nlogn) algorithm which is a much nicer runtime performance than before. The main idea behind the algorithm is being able to construct the hierarchy by creating bounding volumes of bounding volumes while ensuring minimal growth from this construction.

## Collision System

The collision detection system is then just a collection of colliders which check and report collisions as they occur. This system is one of the main systems that requires a lot of additions as more features are added in such as collision resolution. For instance, we would need some parameter on handling collision resolution as well as a link to the rigid body that the collider handles if one exists. We would also need some event system for propagating collision events and setting up collision handlers etc.

# Demos (Renderer)

Executables of these demo are available in the “demos” directory

## Projectile Demo

A demo that shows projectile movement of different types of projectiles. The demo includes 4 different types of projectiles (a regular box projectile, a bullet projectile, a fireball like projectile, a laser projectile). It shows the different behavior that can be simulated with each projectile. Projectiles would generally have a force applied to them and then the acceleration and velocity generated would be affected by gravity until it starts descending when gravitational acceleration is higher than the acceleration caused by the force. However, the other types of projectiles act differently due to differences in mass and acceleration. Bullets for example has very low mass which makes them move a lot more before they start descending due to gravity. Lasers do not have any mass and are thus not affected by gravity at all.

This demo seems to work as intended as the rigid body system simulates these projectile’s behavior properly without the need of any medication.

## Collision Demo

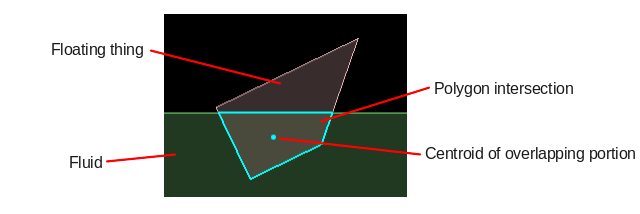
A demo that shows collision detection algorithms and rigid body force application. The main collider used in this demo is an OOBB collider and when a box type bullet collides with the box a force is applied to the rigid body causing an acceleration and a change in orientation(torque). So, this demo would essentially show how the 2 systems should work together to simulate realistic body behavior.

The main way I would update this demo is via implementing proper collision resolution algorithms that would allow us to properly simulate the collision of the bullet on the box obstacles.

## Buoyancy Demo

A demo that shows the buoyancy force generator applied to a rigid body according to its submergence into the liquid’s plane. This force is determined by volume of the object submerged the density of the liquid and the gravitational acceleration of the world.

For the moment, the buoyancy generator generally works for aligned objects that mostly act like a AABB shape as we cannot exactly calculate the volume submerged very easily or efficiently and we also need to apply buoyancy forces not only the center of the of the object at least not at the very beginning but at the center of buoyancy which is the center of the submerged volume. So, the engine thus requires some extra functionality for finding the submerged volume and its new center and getting the ratio of the volume to the entire convex shape and doing that all efficiently.



## Fluid Simulation Demo

A demo that shows a fluid simulation based upon Smoothed Particle Hydrodynamics. SPH uses a large number of particles and has then simulate the motion of liquid using some parameters such as density and pressure based upon other particles around it and then calculating the acceleration of the particles using the mass, density, pressure, and some smoothing kernels and values that we can define. In addition, we also use viscosity of the liquid to influence the acceleration of particles using some viscosity coefficients density and masses.

The main bottleneck of this simulation is having the double loop functionality for finding and checking neighbors of particles to update their forces and accelerations. So, using some spatial partitioning algorithm(octtrees, BSPs etc…) would make this simulation a lot faster than before and increases the performance of the simulation and then we would be able to increase the number of particles and create a more accurate simulation.

# Future of the Engine

## Collision Resolution Systems

One of the main features that the engine needs is handling collision resolution as we can detect when a collision has occurred between 2 colliders, however, we do not have a collision resolution system that allows us to resolve this collision and bring our colliders back to a position where they do not collide. The main issue with a system like that is the complexities that it will introduce as we need such a system to be highly tunable by users as some users may or may not want a collision resolution to occur or want it to occur in a certain manner or just do not want collision resolution and want us to report a collision instead. This adds the complexity of creating something like an Event System and then some event handlers for collisions. In addition, we also have the issue of linking colliders with rigid bodies and how collisions would affect rigid bodies with collision forces being applied to these rigid bodies and translating that effect between both the body and collider.

## Contact calculations

Calculating contacts between colliding objects is pretty helpful for collision resolution and when reporting collisions to users about their colliders. It can also help us with getting more realistic information about the colliders as in most situations colliders are overlapping instead of just colliding so now where they did collide can give us some information about the collision.

## Continuous Collision Detection systems

Continuous collision detection is a pretty helpful technique when objects are either too small that collision can be missed on these objects or objects have are too fast that within an update tick, they penetrate the object completely before it can be detected. This phenomenon is called tunneling. For that reason, we would need a continuous collision detection system that would help alleviate this issue at least to a reasonable degree by finding out when a collision can occur and that can be done by sweeping along the start and end position of the object and checking an object exists along the way.

## Soft bodies

Although rigid bodies can help us simulate many phenomena to an acceptable degree of accuracy, it does make the assumption that no deformation occurs on the body which is not entirely realistic. The main use of soft bodies can be Cloth simulation and simulating some basic deformation of an object due to collision with another object (like a car collision or something similar to that).

# Conclusion

In Conclusion, the result of this project although not complex enough to be used solely within the context of a Real-Time game seems to be promising enough that it can be scalable to achieve that goal given more time and development. I can say that this engine can definitely be used as a foundation to write a performant physics engine that can be used within the context of Real-Time games and hopefully, I have illustrated some of the directions that the engine can take and the ones that I will be taking it within the near future. As for within the context of this project, most if not all of the objectives of this endeavor I would say have been achieved and I have gained a lot of insight into the process of writing a physics engine and have hopefully provided a good enough starting point for other physics programmers to be able to get started within the field and not get frustrated at the lack of resources and have some foundation to start from in writing their own physics systems.

# References

* Millington, I. (2010). *Game Physics Engine Development: How to Build a Robust Commercial-Grade Physics Engine for your Game*. Boca Raton, FL: CRC Press.
* Ericson, C. (2004). *Real-Time Collision Detection*. Boca Raton, FL: CRC Press.
* Barnes, T. (2011, May 3). Fast, Branchless Ray/Bounding Box Intersections | tavianator.com. Retrieved December 9, 2019, from <https://tavianator.com/fast-branchless-raybounding-box-intersections/>
* bulletphysics/bullet3. (n.d.). Retrieved December 9, 2019, from <https://github.com/bulletphysics/bullet3>
* Buoyancy - Box2D tutorials - iforce2d. (n.d.). Retrieved December 9, 2019, from <http://www.iforce2d.net/b2dtut/buoyancy>
* Catto, E. (n.d.). erincatto/Box2D. Retrieved December 9, 2019, from <https://github.com/erincatto/Box2D/tree/master/Box2D/Collision>
* Controlling Objects With Forces and Torques | Digital Opus. (n.d.). Retrieved December 9, 2019, from <http://digitalopus.ca/site/pd-controllers/>
* Distance between Lines, Segments and their CPA (2D & 3D). (n.d.). Retrieved December 9, 2019, from <http://geomalgorithms.com/a07-_distance.html#dist3D_Segment_to_Segment()>
* Fix Your Timestep! | Gaffer On Games. (n.d.). Retrieved December 9, 2019, from <https://web.archive.org/web/20180321070852/https://.gafferongames.com/post/fix_your_timestep>
* Gamasutra - Simple Intersection Tests For Games. (n.d.). Retrieved December 9, 2019, from <https://www.gamasutra.com/view/feature/131790/simple_intersection_tests_for_games.php?print=1>
* Implementing GJK | Shiny Pixels. (n.d.). Retrieved December 9, 2019, from <http://vec3.ca/gjk/implementation/>
* Souto, N. (2015, March 2). Video Game Physics Tutorial - Part II: Collision Detection for Solid Objects. Retrieved December 9, 2019, from <https://www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects>
* Muratori, C. (n.d.). Implementation of GJK (2006). Retrieved from <https://caseymuratori.com/blog_0003>
* Collosion Detection Research Based on Capsule Bounding Volume. (n.d.). Retrieved from <https://www.researchgate.net/publication/275025293_Collosion_Detection_Research_Based_on_Capsule_Bounding_Volume>
* Cato, E. (n.d.). Computing Distance. Retrieved from <http://box2d.org/files/GDC2010/GDC2010_Catto_Erin_GJK.pdf>
* Dynamic Collision Detection using Oriented Bounding Boxes. (n.d.). Retrieved from <https://www.geometrictools.com/Documentation/DynamicCollisionDetection.pdf>