Claes Twannes

Building a custom 2D physics engine in C++

Supervisor: Vanden Abeele Alex

Coach: Verspecht Marijn

Graduation Work 2023-2024

Digital Arts and Entertainment

Howest.be

CONTENTS

ABSTRACT & KEY WORDS

This paper goes in detail on how I made a custom 2D physics engine from scratch in C++. It is about learning about the inner workings, calculations, and techniques that come into play when developing a physics engine. I will discuss how to implement different shapes that interact with each other, and what behavior they have in relation to each other, by conducting different series of experiments with these shapes. I will address and research on how to make a physics simulation deterministic with dos and don'ts. I will research on how to make a physics simulation accurate and what techniques can be utilized on how to make it more or less accurate. This paper contributes to the community of people eager to start learning about physics programming and physics engines in general, and may be a great starting point to do research for this broad topic.

Dit artikel gaat in detail hoe ik een eigen 2D physics engine in C++ heb gemaakt. Het gaat over het leren over de innerlijke werking, berekeningen en technieken die betrokken zijn bij het ontwikkelen van een physics engine. Ik zal bespreken hoe ik verschillende vormen implementeer die met elkaar interageren, en welk gedrag ze hebben in relatie tot elkaar, door verschillende series experimenten met deze vormen uit te voeren. Ik zal onderzoeken hoe je een physics simulatie deterministisch maakt met do's en don'ts. Ik zal onderzoeken hoe je een simulatie nauwkeurig kunt maken en welke technieken je kunt gebruiken om het meer of minder nauwkeurig te maken. Dit artikel draagt bij aan de gemeenschap van mensen die graag meer willen weten over physics programmeren en physics engines in het algemeen, en kan een goed startpunt zijn voor onderzoek naar dit brede onderwerp.

PREFACE

Recently in my studies of game development, I was introduced to Nvidea PhysX used in a framework built by a teacher. In this framework we needed to play around with different objects, meshes, and their properties. We made some small assignments to cover the basics. Quickly after, we moved on to different subjects.

However, my curiosity didn't stop there, I was eager to learn more about physics programming and physics engines. During the summer I wanted to start a side project to delve deeper into physics programming and get to know the inner workings. Unfortunately, I didn't have the time due to personal reasons, and was disappointed that I wasn't able to pursue this. Then the perfect opportunity presented itself in the form of this bachelor thesis assignment. This paper is a humble attempt from me trying to understand the field of physics programming and engines. I hope other people that are sparked by the same curiosity as I find this paper interesting and see this as a starting point to go on their own journey of physics programming.

INTRODUCTION

In this paper I will delve deeper into how to build a physics engine and learn about the inner workings of other engines. This paper covers exploring different shapes and researching different techniques, to see how all those different implementations affect each other. I will talk about the essential mathematics, collision detection and resolving, implementing different shapes, how to achieve a more accurate simulation, techniques to make a simulation deterministic, find what affects performance, and what optimizations can be implemented into the engine.

The question my research tries to answer is **"How do diverse shapes within a 2D physics engine impact simulation accuracy, computational efficiency, and deterministic physics?".**

The hypotheses of my research are the following:

1. **Simulation accuracy:**
   Diverse shapes, including normal shapes (squares, circles) and custom polygons, significantly influence simulation accuracy. Increased shape complexity may not necessarily lead to improved simulation accuracy.

2. **Computational efficiency/complexity**
   Incorporating custom convex polygon shapes adds complexity to the physics logic, impacting computational simulation efficiency. Including concave polygons introduces even greater complexity. Handling convex polygons will demand performance and increase the workload of the physics logic.

3. **Determinism**
   The inclusion of diverse shapes will challenge the maintenance of deterministic physics calculations. It may result in events of non-deterministic behavior, where the same inputs do not consistently produce equal outputs.

LITERATURE STUDY / THEORETICAL FRAMEWORK

POLYGONS

I will implement polygons into my physics simulation, but you must think about it before you implement them, because there are 2 different kinds.

For my simulation I will only use **convex** polygons, convex polygons are shapes where the interior angle of a point must be less than 180 degrees. For a triangle the total inner angle is 180 degrees, and for convex polygons there is a formula needed to calculate the total inner angle [1].

- (n – 2) * 180 == total inner angle in degrees

Then you have **concave** polygons, those are shapes where the inner angle of a point exceeds 180 degrees.

The reason why I will only use convex polygons for my simulation is because the math needed to calculate collisions for concave polygons is way out of scope for my research. This means I will focus on convex polygons.

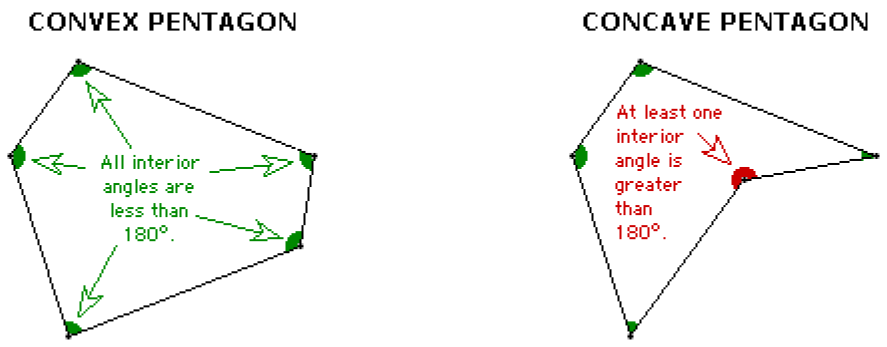

CONVEX PENTAGON
All interior angles are less than 180°.
CONCAVE PENTAGON
At least one interior angle is greater than 180°.

Figure 1: Convex vs concave polygons [2]

PHYSICS EQUATIONS

A physics equation in this context means how the objects position, velocity and acceleration will be calculated. Each method has its own pros and cons.

The following graph describes how close the physics equations are to the real-time equation. The graph represents the real-time equation of sin(t)^2. Each equation tries to be as close as possible to this real-time equation [3].

The problem is that a simulation is done over steps in time. On the x-axis we can see the time. Each time we would calculate the equation, a given amount of time has already elapsed. This is why certain physics equations try to calculate the values in between, to make that line smoother so they can have a more accurate representation. The y-axis is the actual value at the given time, this would represent position, velocity etc [4].

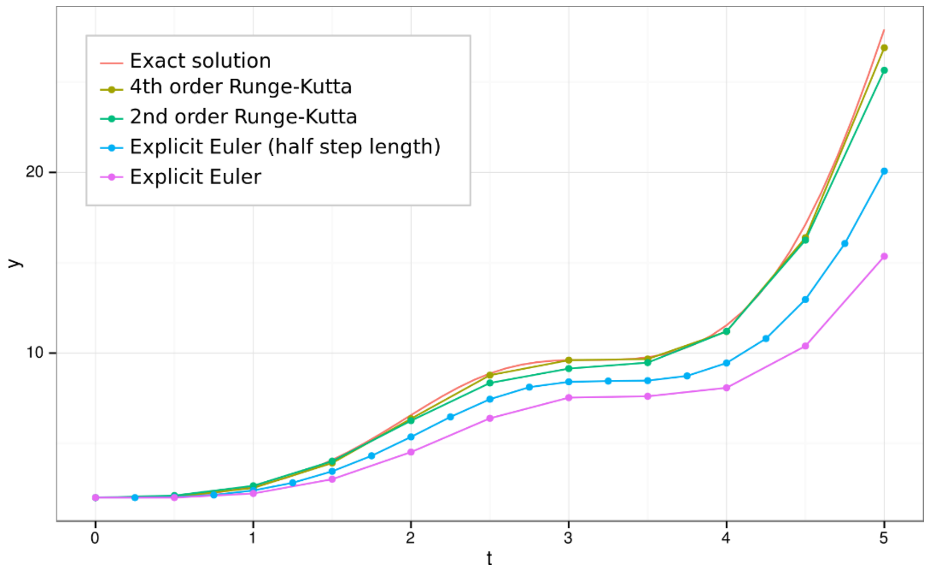
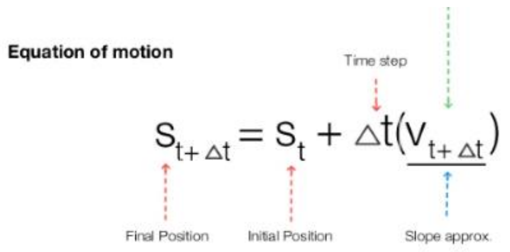


Figure 2: Different physics equations graph [5]



$$S_{t+\Delta t} = S_t + \Delta t(\underline{V_{t+\Delta t}})$$

Figure 3: The equation [6]

EULER

The Euler method results in more of an approximation of physics rather than an extremely accurate representation.

INTEGRATION

```
Velocity += Acceleration * deltaT
Position += Velocity * deltaT
```

Figure 4: Euler integration

This is a very easy way, to understand how the objects will move and will result in an intuitive physics simulation. It also very easy to compute for the processor [4].

VERLET

The Verlet method is lesser known, but still used a lot.
It is an improvement on the Euler method offering better accuracy for containing energy between points. This method is mostly used in things like cloth simulations, web simulations, chains, soft bodies.

The trick of the Verlet method is that it uses the previous position and velocity to calculate the new one. This is already more complicated than the Euler method because it doesn't look intuitive at first sight [7].

INTEGRATION

```
newPos = pos + velocity * deltaT + acc * (deltaT^2 / 2)
newVel = vel + acc * (deltaT / 2)

pos = newPos
vel = newVel
```

Figure 5: Verlet integration [8]

RUNGE KUTTA 4 / RK4

The RK4 method tries to be close to the approximation of the real-time equation by considering the slope at multiple points at each time step. It then divides into smaller steps and calculates 4 values in between. Then it combines them to take a weighted average. This results in a more accurate approximation of the real-time equation [5].

The RK4 should then also be a higher cost to calculate compared to previous methods [5].

INTEGRATION

```
vec2 k1xy = velocity * deltaT
vec2 k1v = acceleration * deltaT

vec2 k2xy = (velocity + 0.5 * k1v) * deltaT
vec2 k2v = (acceleration + 0.5 * k1v) * deltaT

vec2 k3xy = (velocity + 0.5 * k2v) * deltaT
vec2 k3v = (acceleration + 0.5 * k2v) * deltaT;

vec2 k4xy = (velocity + k3v) * deltT
vec2 k4v = (acceleration + k3v) * deltaT

position = position + (k1xy + 2.0 * k2xy + 2.0 * k3xy + k4xy) / 6
velocity = velocity + (k1v + 2.0 * k2v + 2.0 * k3v + k4v) / 6
```

Figure 6: Runge Kutta 4 / RK4 integration for position and velocity [5]

RIGID BODY PHYSICS / KINEMATICA

Rigid body physics is the study of the movement of a given body, which external forces can act on. The reason for the name "rigid" is that they can't be deformed. The shape will always stay the same in every circumstance. For example, I will have circles, boxes, and custom convex polygons that will interact with each other. These interactions will not deform the shapes in any way.

For moving the rigid bodies, the chosen physics equation is used, for example, Euler integration.

Rigid bodies have different properties: shape, mass, elasticity, friction, inertia. The combination of all these properties creates all sorts of unique rigid bodies that will all react in another way.

Rigid bodies are also under the effect of forces, both linear and rotational. They should react to it and move in the desired way. Torque can be added for a rotational force, for example to make a ball roll, or a linear force like gravity can be added to make the ball go down.

These are the principles of kinematics that get applied for rigid body physics [9] [4] [10] [11].

COLLISION DETECTION

When running a simulation, you need to have a way to know whether the shapes are colliding with each other. This consists of checking whether lines of a shape are intersecting with lines of another shape.

For my simulation I will need to know how to detect collisions between circles vs circles, circles vs convex polygons, and convex polygons vs convex polygons [11] [4].

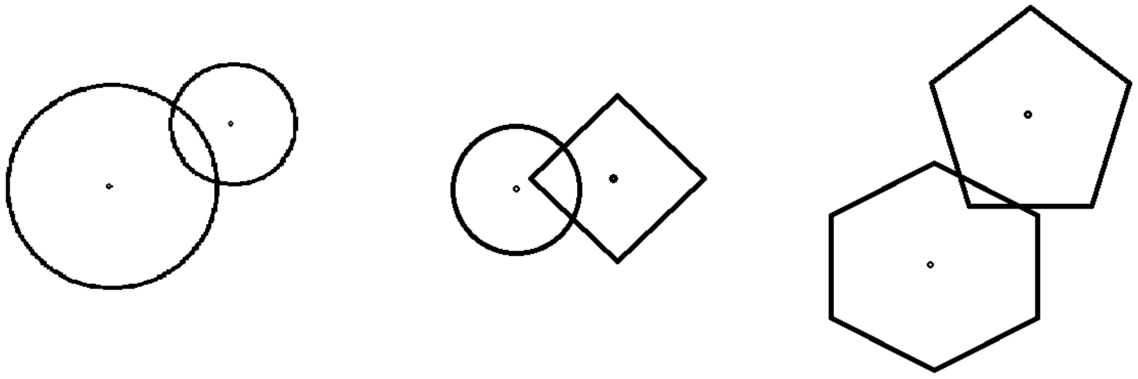


Figure 7: All different collision detections

CIRCLE VS CIRCLE

Circle versus circle collision detection is the easiest implementation of collision detection and very efficient. Because of this, a lot of optimizations can be done for collision detection.

This detection is done by calculating the distance between 2 circles. If the distance between them is smaller or equal to the sum of the radiuses, then we know that the 2 circles are colliding [4] [11].

```
if(distanceBetweenCircles <= a.radius + b.radius)
{
  //Colliding
}
```

Figure 8: Circle vs Circle collision detection

BROAD PHASE VS NARROW PHASE

Because the circle vs circle detection is very efficient, a small but powerful optimization can be added by putting circles around every object that is not a circle.

This is called the broad phase. It can also be implemented by putting boxes around them, because they are also efficient to calculate, but because implementation for circles is already needed, I will use circles [12] [13] [14].

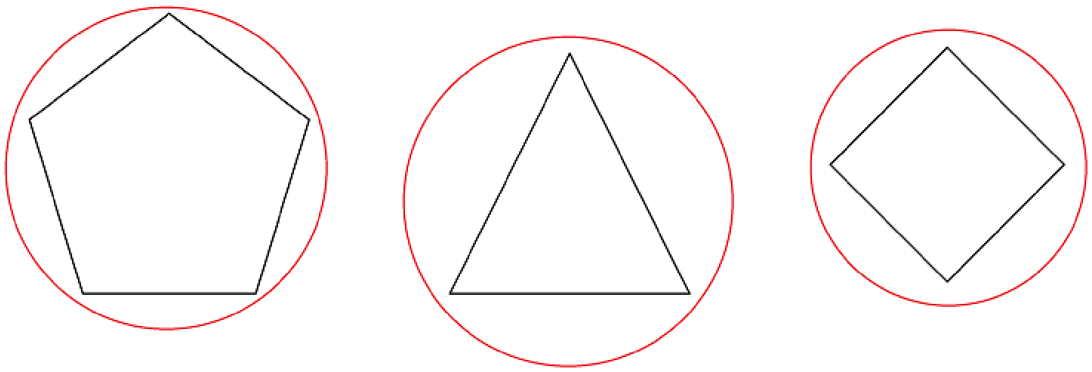


Figure 9: Broad phase

If the circles with polygons inside them are colliding, then we know we might have a collision of a polygon or might be near a collision of polygon because there is a little bit of space where they might not collide. This is why some shapes could instead have a box around them, because it would be a better fit for a given shape.

So, if the broad phase succeeds, the narrow phase can start and that is just simply checking collisions for the polygons themselves. This way we safe some efficiency by not calculating polygon collisions if they are not near each other [12] [13] [14].

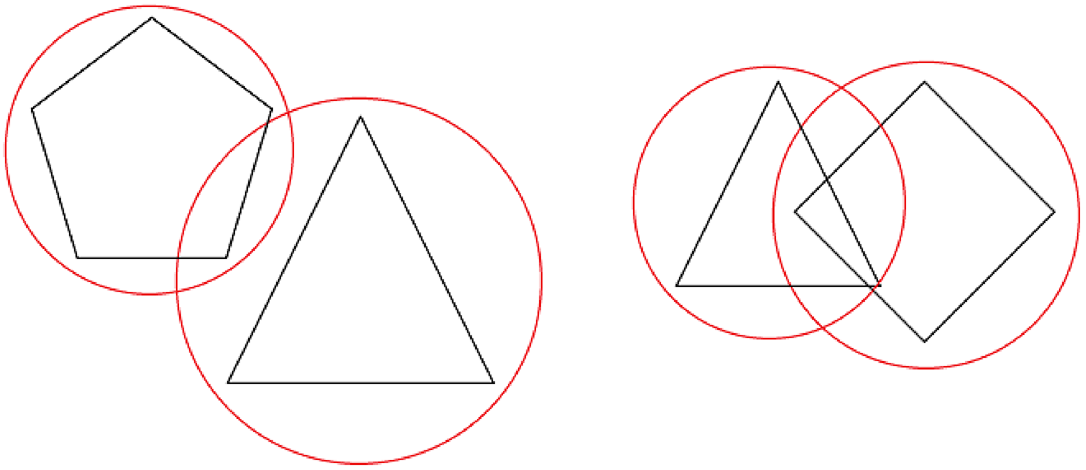


Figure 10: Narrow phase

CONVEX POLYGON VS CONVEX POLYGON

Polygon vs polygon collisions are the most expensive to calculate because you must iterate trough all points and lines of your shapes to find the collision. This means the more points you have, the more expensive your calculation will become.

The best approach to this, is called Separating Axis Theorem (SAT).

This works by projecting each edge of the first polygon onto the perpendicular axis of that edge and checking if the projection of the second polygon overlaps. If no overlap is found, then we already know no collision is possible, this implies that we can return early without checking any other points. If they keep overlapping, we check the second polygon with its edges and the perpendicular axes. If they all overlap, we know that there is a collision between the 2 polygons [4] [15].

This is why if you have more edges on your shape, collision is more complex to calculate because it needs to iterate through all the points.

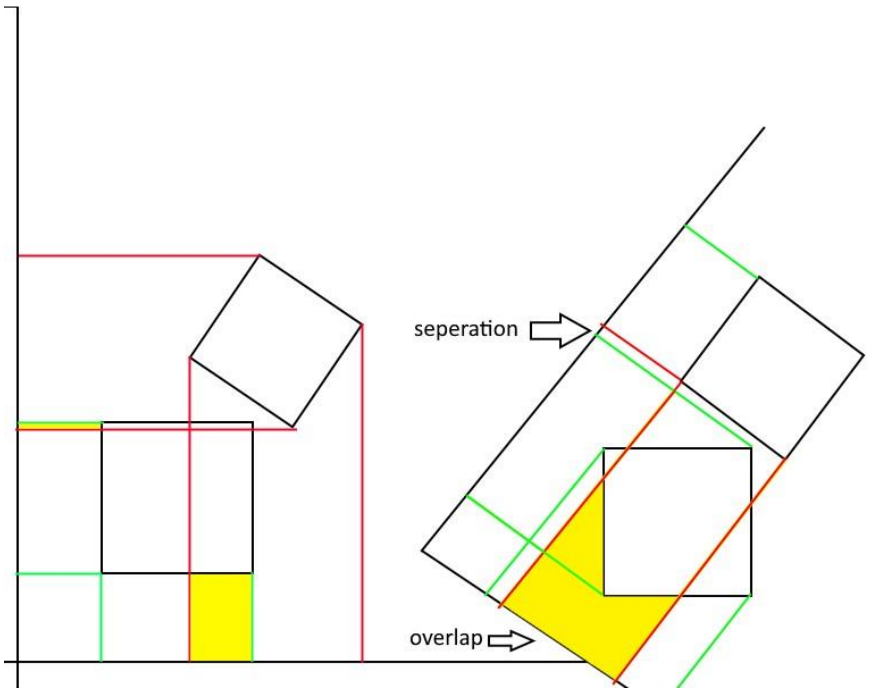


Figure 11: Separating Axis Theorem

CIRCLE VS CONVEX POLYGONS

For this type of collisions SAT can also be applied, but in a slightly different way.

The way of projection onto the perpendicular axis stays the same, but we can't leave early. We loop through all the edges and find the best projection [4].

We divide our edge with 1 perpendicular axis on each point. This results in 3 regions A, B, and C.

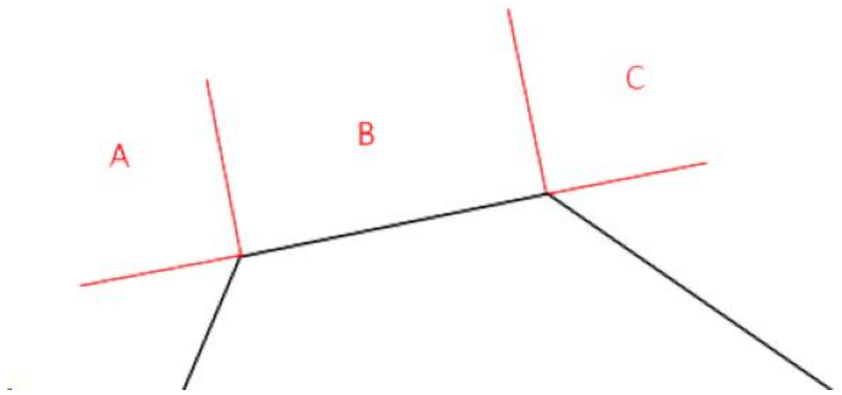


Figure 12: Zone A, B, C

For zone A and C, we can calculate if the circle resides in that zone by calculating the dot product between the axis and the vector between the circle and the point. If that is true, we check if the distance between the circle and the point is less than the radius of the circle. If it is smaller, we have found a collision between a circle and a polygon.

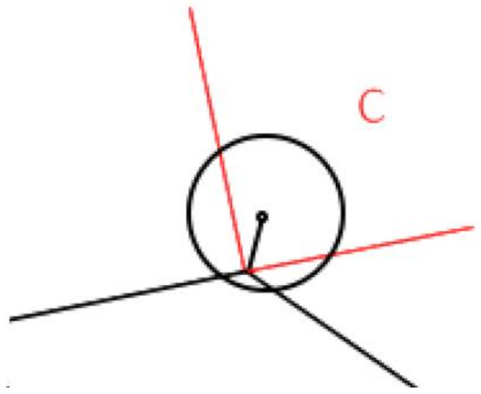


Figure 13: Distance smaller than the radius

If the previous is false, it means the circle is in zone B, to check if it collides there, we check if the radius is bigger than the length of the projection. If that is true, we have found a collision between a circle and a polygon.

COLLISION RESOLVING

After a collision has been detected, you need to resolve the collision. This means that the shapes must react in a natural way to one another. Each collision has data we can save. This contains direction, depth, start point, end point, body A, body B.

Because of this we can resolve all the collisions in one function instead of separately. There are two steps in resolving collision: positional correction and adding impulses.

Positional correction is moving the bodies away from each other using the data given. The purpose of this is to separate the bodies so that they do not collide anymore. This is calculated by moving the bodies away trough the direction and the depth received from the data [4] [11] [16].

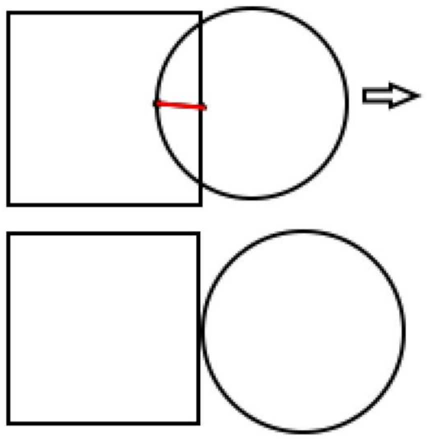


Figure 14: Positional correction

The second step is the impulse response. This results in adding a force directly onto the velocity instead of adding it to the acceleration. This is to generate a quick response between the objects.

The formula to generate the impulse force, is based on all the properties of the rigid bodies: mass, elasticity, friction, inertia. It also accounts for the collision data to find the direction and depth of the collision [4] [11] [16].

$$j = \frac{-(1+e)((V^A - V^B) * t)}{\frac{1}{mass^A} + \frac{1}{mass^B} + \frac{(r^A \times t)^2}{I^A} + \frac{(r^B \times t)^2}{I^B}}$$

Figure 15: Impulse formula [11]

RESEARCH

1. TESTING ENVIRONMENT

The environment where I will be conducting my experiments will be my self-written C++ 2D physics engine from scratch.

1.1 LIBRARIES

The first question for my physics engine was how I was going to do the math calculations. I had the option to write my own math library or use an existing one. Because this project is for research purposes, writing a custom math library is out of scope. In addition, existing math libraries are optimized and made for external usage.

This is why I decided to use the **GLM** math library. I have used it in previous projects, and it proved to be user friendly and comprehensive. GLM contains all the functionality I need for my physics engine: linear algebra, trigonometry functions, and matrices [17].

As for the graphics side, I have had some prior experience with **SDL** and decided to use this as my graphics library. I implemented opening a window and rendering text, circles, boxes, and polygons.

SDL can also be used to handle events, for example closing the window, mouse position, keyboard/mouse button presses. I used these events to spawn objects when clicking with my mouse in the window [18].

1.2 ENGINE

The basic mechanism of my engine consists of a simple game loop, with the following order of events.

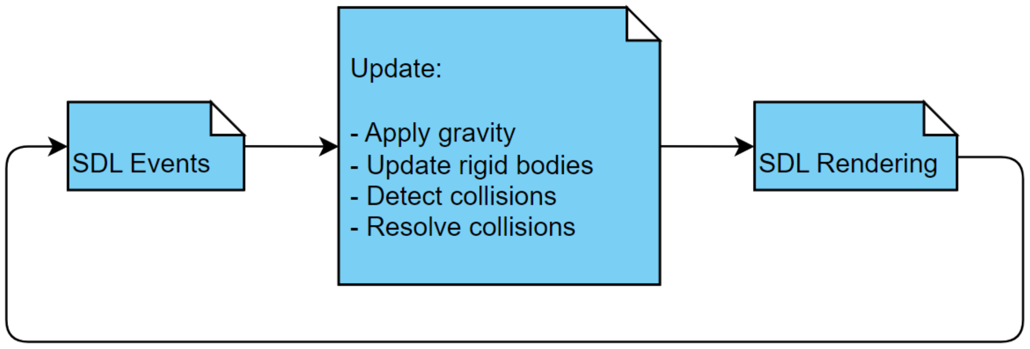


Figure 16: Game loop

1.2.1 SDL EVENTS

Here I check if SDL has registered a mouse click on the screen. If it has, I will spawn a circle, box or a random convex polygon based on what mouse button I pressed on the mouse position.

1.2.2 UPDATE LOOP

The update loop consists of different steps in a specific order. All the rigid bodies get updated here. This is the heart of the simulation where all the math comes together in one piece.

1.2.2.1 EXTERNAL FORCES

I will first add a specified gravity force to every rigid body. A different example of an external force could be a wind force that gently pushes your rigid bodies to the right. This simple step consists of adding the correct force to the bodies each frame.

1.2.2.2 RIGID BODY/SHAPE UPDATE METHOD

Each rigid body has an update method, as well as a shape, which represents whether it is a circle, box, or polygon. Each shape also has an update method. This gets called in the update of the rigid body.

In the update method of the rigid body, the position, velocity, rotational velocity, and rotation get calculated based on the chosen physics equation.

After the new position and rotation have been calculated through the physics equation, the update method of the shape gets called. This is where the points of the shape get converted from local to world space based on the rigid bodies position and rotation.

```
UpdateRigidBody(float dt)
{
    //Update position and velocity based on the chosen physics equation
    //Euler integration
    Acceleration = AccumulatedForce * InvMass;
    Velocity += Acceleration * deltaTime;
    Pos += Velocity * deltaTime;

    //Update the points of the shape to get the correct world space
    Shape.Update()
}
```

Figure 17: Body update method

1.2.2.3 COLLISION LOOP

The last step in the update loop is to check if the updated rigid bodies which have received new positions and rotations, are colliding with other rigid bodies now that they have been changed.

To make this happen each rigid body must loop through all the other rigid bodies in the simulation, and check collisions between each other based on what shape they are.

To make sure that each rigid body doesn't check another body twice or more, it only checks the rigid body to the right inside the list of bodies. This solution avoids **duplicate** checking completely [4].

```
//Loop trough all rigidbodies, except the last one
for(int i = 0; i <= bodies.size() - 1; ++i)
{
  //Each rigidbody checks all the bodies right of it, to avoid duplicate
checking
  for(int j = i + 1; j <= bodies.size(); ++j)
  {
    //Check collision between body at index i and body at index j
  }
}
```

Figure 18: Collision loop

This obviously means that the more rigid bodies you have in your simulation, the more bodies must be checked by each individual body. This results in a big O notation of **O(n^2)**.

1.2.3 DRAW LOOP

Here the rigid bodies get rendered by SDL. The only small special thing in the draw loop, is that I also draw debug information to visualize collision data for debugging, like the normal, and the end/start point.

1.3 SIMULATION

The core of the simulation exists out of kinematics and collision responses. To start coding the simulation I had to start with kinematics, so that I can have moving objects that have their own properties and can be influenced by linear and rotational forces.

1.3.1 PARTICLE PHYSICS

The basics of rigid body physics without collisions is called particle physics, and I started with implementing this first.

This contains of making a rigid body class that contains a position, velocity, acceleration, rotation, and mass. Those values get updated based on the desired physics equation.

However, without a change of acceleration, the body won't move. That is why we need to apply force to the object. This consists of having a function (AddForce) where you can accumulate forces and then add them to the acceleration scaled with the bodies mass. This means that the heavier your object, the less it is affected by the forces.

However, we must be careful with one thing while applying force. That is considering gravity. Objects that are under the influence of gravity fall at the same speed even if other objects are heavier.

Therefore, you must take the mass of the object into account to correctly scale the gravity force. This makes sure that all the objects in the scene fall at the same speed, no matter their mass. This is what happens in a vacuum, in the real world there are other factors like air drag that have influence on how a rigid body will behave in a free fall.

There is also a second way to apply forces, called impulses. The implementation of this is very straightforward. Instead of adding forces to the acceleration we add it directly to the velocity. This results in a quick response later needed to implement collisions [9].

1.3.2 COLLISION SHAPE

To have collisions you need to have a shape for your body, I implemented circles, boxes, and polygons. All shapes derive from the class "Shape". The class "Box" derives from the class "Polygon" because a box is just a polygon with 4 sides and all equal inner angles. Each shape also has a function that returns which type they are circle, box, or polygon.

The only information these shapes have is the representation of their shape: a circle only has a radius, polygons have their local points that get translated into world space based on the position and rotation. They also have a function to return their inertia. The calculation of inertia is dependent on each shape. Inertia represents the likeliness the shape is going to rotate. You can imagine it as rotational mass [4].

```
CircleInertia = 0.5 * Radius^2 * Mass
```

Figure 19: Circle inertia calculation [19] [4]

```
BoxInertia = 1/12 * Width^2 + Height^2 * Mass
```

Figure 20: Box inertia calculation [19] [4]

```
float totalArea
float totalMass

for(int i{}; i < verticesSize); ++i)
{
    const Vec2 a = vertices[i]
    const Vec2 b = vertices[i + 1]

    const float crossDistance = abs(Cross(a, b))

    totalArea += crossDistance * (Dot(a, a) + Dot(b, b) + Dot(a, b))
    totalMass += crossDistance
}

Inertia = totalArea / 6 / totalMass
```

Figure 21: Convex polygon inertia calculation [20]

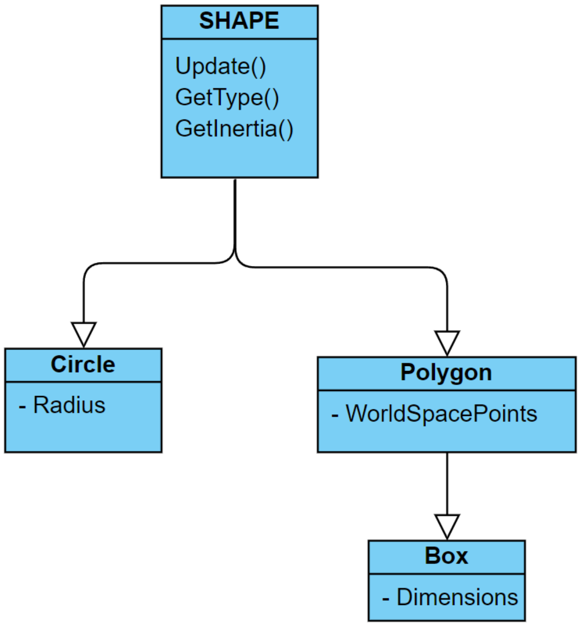


Figure 22: Shape UML

1.3.3 COLLISION SYSTEM

After all the rigid bodies have been updated with their new position and rotation, we need to check if they have collided with other rigid bodies in the scene.

I already discussed in **1.2.2.3** Collision **loop** on how we iterate through all the rigid bodies to avoid duplicate checking.

The next step is detecting collisions between different shapes, and then resolving collisions if they have been detected.

The first step in the collision detection function is to differentiate between the 2 colliding shapes and call the correct function based on what types the shapes are.
I implemented Circle vs Circle, Polygon vs Circle, and Polygon vs Polygon collision detection/resolving.

To call the correct function, I check the shape types with a switch statement instead of doing a lot of if checks. This is a small optimization I can utilize in my hot code path.

```
switch(typeA)
{
    case Circle:
        switch(typeB)
        {
            case Circle:
                CircleVsCircle()
            case Box:
            case Polygon:
                PolygonVsCircle()
        }
    case Box:
    case Polygon:
        switch(typeB)
        {
            case Circle:
                PolygonVsCircle()
            case Box:
            case Polygon:
                PolygonVsPolygon()
        }
}
```

Figure 23: Shape differentiation

Before we call the correct function, we can add a little optimization. If the combination of shapes is not a circle vs circle collision, we can first calculate if the bounding spheres around the polygons or boxes are colliding. If that succeeds, we can go on to the correct function **(Broad phase vs Narrow phase)**.

Once we are in the correct function, we use the correct calculation to detect a collision between the 2 shapes. After we know that they have collided we need to fill in the data of the collision.

The data consists of a pointer to the first and second shape, the normal direction of the collision, the depth of the collision, and then the start/end point calculated by the depth, the normal, and point of contact.

```
struct CollisionData
{
    RigidBody* a
    RigidBody* b

    vec2 normal
    float depth

    vec2 start
    vec2 end
};
```

Figure 24: Collision data struct

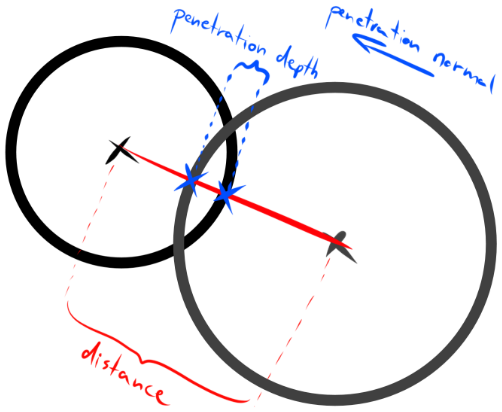


Figure 25: Visualization of collision data [21]

After storing the data, we can resolve the collision. The first thing to do is the positional correction. Next, we add impulses. These functions both need the collision data to determine the correct values to update the bodies **(Collision resolving)**.

```
//Get the length of the correction
//Let the bodies penetrate lightly without moving to lower jitter
Vec2 correctionLength = max(depth - minPenetration, 0.f) / totalInvMass *
normal * percent
//Apply correction to the positions of the bodies
a.Pos -= a.InvMass * correctionLength;
b.Pos += b.InvMass * correctionLength;
//Update the bodies
a.Update()
b.Update()
```

Figure 26: Positional correction implementation [16]

The impulse part is a bit more complicated. This is because we need to combine linear and rotational forces while also taking elasticity and friction into account. I convert these 2 formulas into code, and then add the correct impulses to the bodies.

This is the last part of the simulation. After this we need to loop again to see if the new position and rotation have detected another collision.

$$j = \frac{-(1+e)((V^A - V^B) * t)}{\frac{1}{mass^A} + \frac{1}{mass^B}}$$

Figure 27: Linear impulse [11]

$$j = \frac{-(1+e)((V^A - V^B) * t)}{\frac{1}{mass^A} + \frac{1}{mass^B} + \frac{(r^A \times t)^2}{I^A} + \frac{(r^B \times t)^2}{I^B}}$$

Figure 28: Rotational impulse [11]

```
//Function to add the impulses directly to the velocity
AddImpulse(Vec2 impulse, Vec2 direction)
{
    Velocity += impulse * invMass
    AngularVelocity += Cross(direction, impulse) * invInertia
}

//Multiply linear and angular impulse with their corresponding directions
Vec2 impulse = (normal * linearImpulse) + (tangent * angularImpulse)

//Add the impulses to the bodies
a.AddImpulse(impulse, contactDirectionA)
b.AddImpulse(-impulse, contactDirectionB)
```

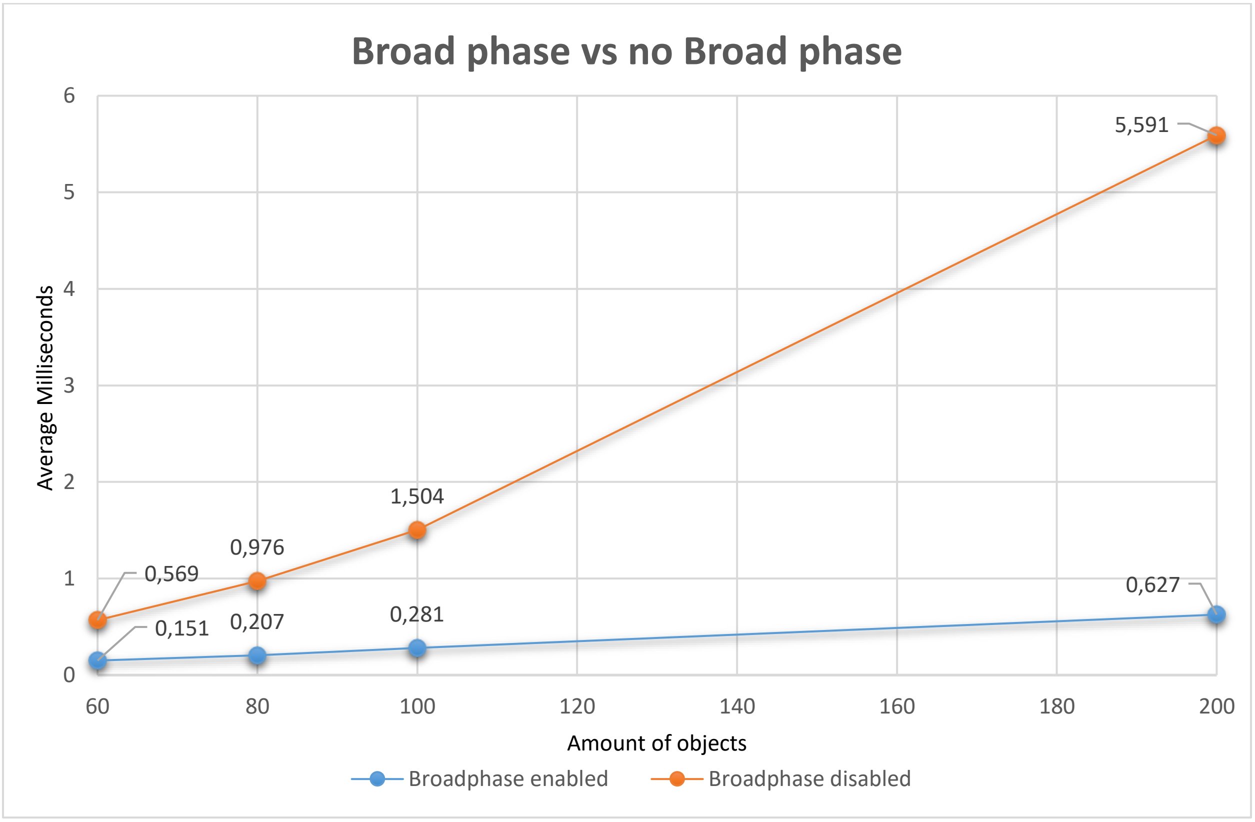
Figure 29: Implementation of the impulse [4] [11]

1.2.2.3 Collision

2. OPTIMIZATIONS

I only added some small optimizations, the biggest one of them all was the Broad phase collision detection. This method can only optimize your simulation if there are polygons or boxes because a simulation with only circles is already the most optimized one.

I ran some tests to visualize the difference broad phase can make in a simulation with only polygons. I tested with different amounts, to see if it would behave linearly or exponentially. Then I got the average time it takes the update loop to calculate the entire simulation in milliseconds.



We can conclude from this that when using Broad phase collision detection, we can go from an exponential growth in frame time to an almost linear growth. This shows that broad phase is a really easy way to optimize your collision detection while reusing code.

Other small optimizations include caching variables, avoiding the usage of square root in hot code path, using switches when doing a lot of if checks.

3. EXPERIMENTS

3.1 PERFORMANCE/COMPLEXITY

As part of my research question, I wanted to see how different shapes influence performance and complexity of the simulation.

To set up this experiment I will have different simulations on these given series of shapes.

- Circles
- Circles & polygons
- Polygons
- All shapes

All simulations will have 150 objects, with equal amounts of the different shapes involved.

Then I will do the same experiments but with different physics equations (Euler, Verlet, and RK4), to see what difference in performance/complexity they give.

To calculate performance, I will get the time of how long the update loop takes to calculate the entire simulation in milliseconds.

To get the complexity I will use Visual studio's performance profiler to see how much CPU usage the update loop takes [22].

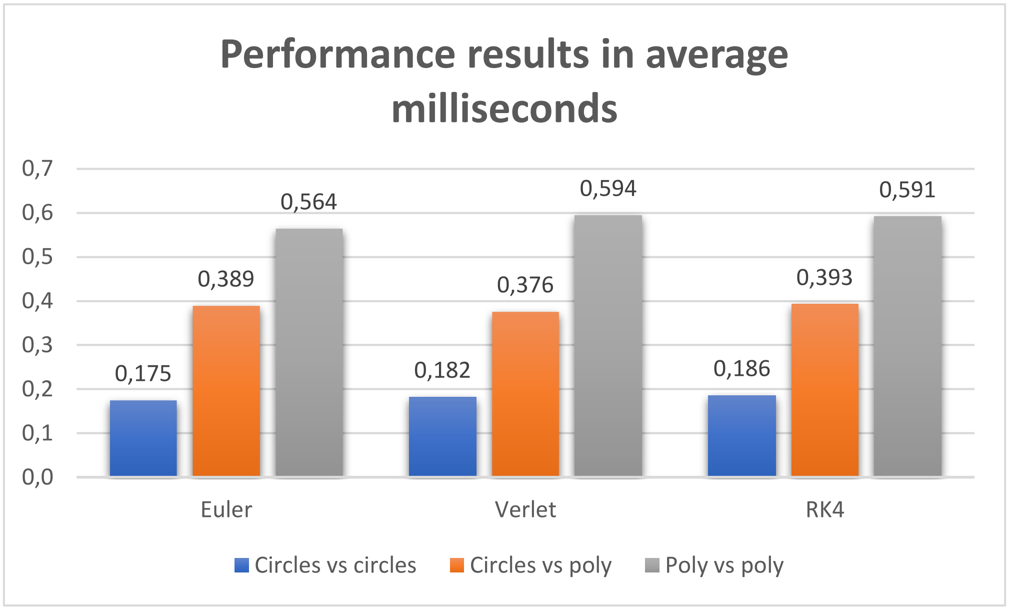


Figure 30: Performance results

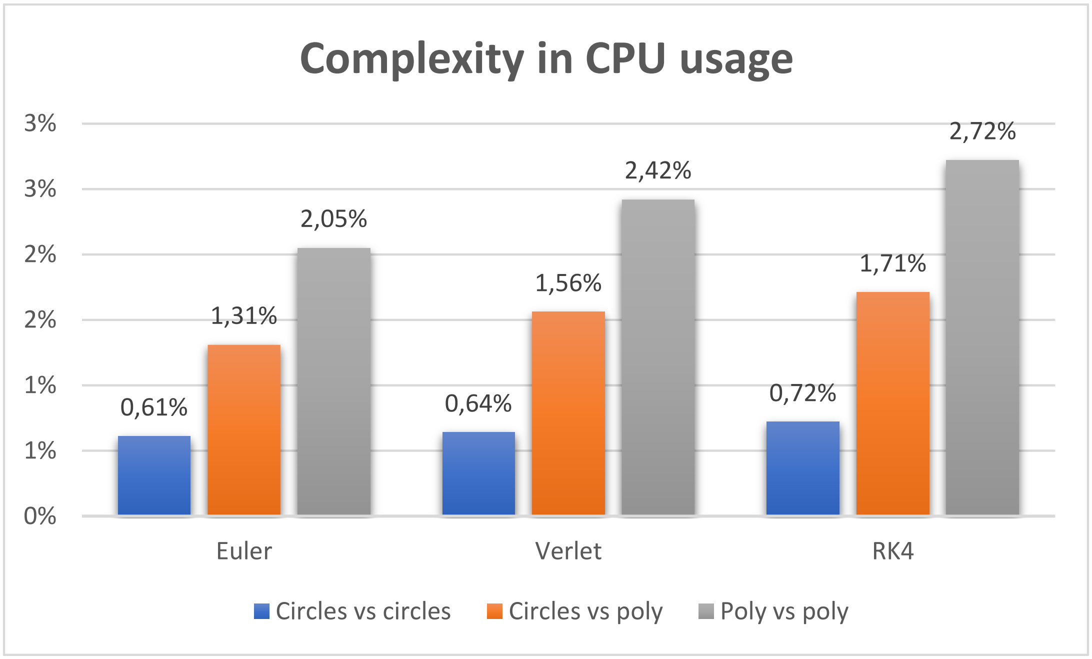


Figure 31: Complexity results

The results of the performance test illustrate that there is little difference between the different physics equations. I was surprised by this result because I expected it to be more differentiated.

As for the different shapes, we can conclude that when introducing polygons our performance drastically lowers. This is best seen in the polygon vs polygon simulation where it 4 times slower than the circles simulation. In a circle vs poly simulation, the value meets in the middle of the circles simulation and the polygons simulation.

This same conclusion is also illustrated in the complexity test, where the different shapes cause the same behaviour in CPU usage. However, when implementing a different physics equation, we can conclude that for the CPU it becomes more complex, no matter what shapes we simulate with.

The behaviour I find interesting here is that when implementing a different physics equation, the CPU instructions become more complex, but it has little to no effect on the performance.

When simulating more objects, I would like to research further on whether this difference becomes more prominent when simulating more rigid bodies. Unfortunately, I got restricted by my graphics API which caused the simulation lag quite badly when trying to simulate larger quantities of rigid bodies.

3.2 DETERMINISM

Determinism is the idea of giving something the same inputs and expecting the outputs to be the same too. In this case, a deterministic simulation means that when we have 2 simulations that are set up identically, they should both behave in the exact same way [23] [24] [25] [26].

When we pause the 2 simulations at the same time, they would need to look identical. If they do, then we can conclude that we have a deterministic simulation.

Before starting my experiment, I researched how to make a physics simulation deterministic. I concluded that having a fixed time step is the most important part of making a simulation deterministic. Other parts include not using random numbers anywhere and using no multi-threading.

Having a fixed time step means that the simulation will always run at the same speed. The recommended time step for physics is 60 frames per second, or in a more understandable notation, 16,67 milliseconds. That means that the simulation will be updated every 16,67 milliseconds or 60 times each second.

The normal way for a time step is using delta seconds, this time step will be calculated on how long the frame took. This can result in one problem. If the frame takes more than 16,67 milliseconds the simulation will be out of sync. This can be solved by using a fixed time step that accumulates the delta seconds, and if it exceeds the 16,67 milliseconds it will run until it catches up.

```
float FixedTimeStep = 1000/60 //16,67ms or 60 frames per second
//Check if the accumulated time reached 16,67 milliseconds or more
while (AccumulatedDelta >= FixedTimeStep)
{
     //if it has update the simulation with the fixed time step
     UpdateRigidBodies(FixedTimeStep)
     //Substract the fixed time step from the accumulation
     AccumulatedDelta -= FixedTimeStep
}
```

Figure 32: Fixed time step loop

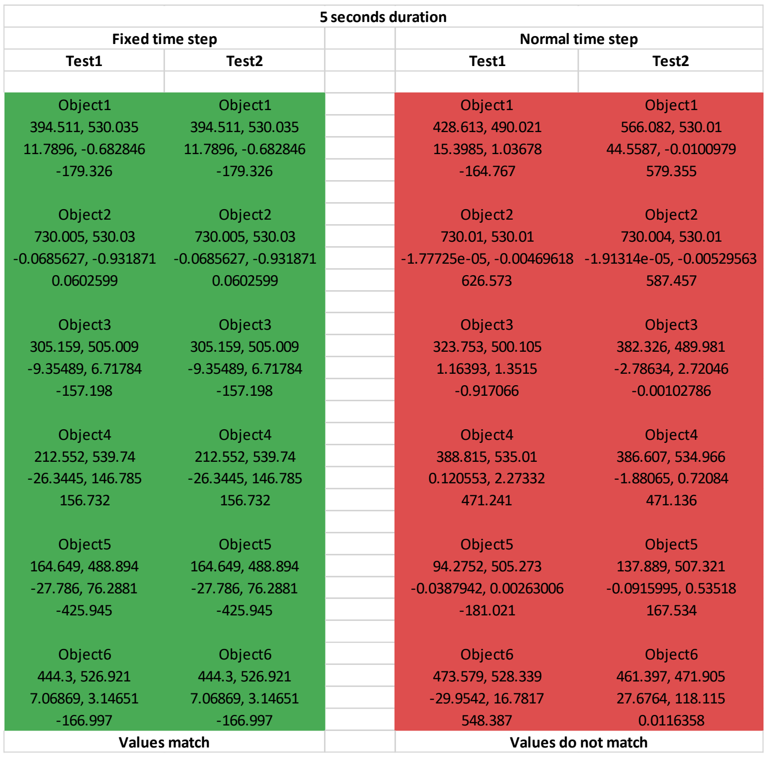
To test if using a fixed time step will make my simulation deterministic, I will run 2 simulations with identical setup. The first test will be using a fixed time step, and the second test will be using a normal time step.

To determine if the output is identical, I will get the position, velocity, and rotation of each body after a given duration, and check if the variables between simulations are matching.

I originally intended to compare different physics equations, but after researching the physics equations I figured out it won't have an effect because it is just another way to calculate the position of the objects.

15 seconds duration				
Fixed time step			Normal time step	
Test1	Test2		Test1	Test2
Object1	Object1		Object1	Object1
433.679, 530.034	433.679, 530.034		433.992, 530.01	680.02, 530.01
2.08715, -0.682613	2.08715, -0.682613		-0.134426, 0.104995	4.27837e-05, -0.00963083
0.113582	0.113582		-618.395	332.874
Object2	Object2		Object2	Object2
730.001, 530.037	730.001, 530.037		730.002, 530.01	730.01, 530.01
-0.00134687, -0.358832	-0.00134687, -0.358832		-4.88681e-05, -0.0212458	5.4294e-05, -0.00519457
0.158915	0.158915		107.539	528.893
Object3	Object3		Object3	Object3
298.344, 504.998	298.344, 504.998		380.101, 485.161	356.255, 490.026
-0.11123, 1.96425	-0.11123, 1.96425		-0.342603, 0.752295	0.080492, 3.94452
-157.336	-157.336		30.417	1,17E+00
Object4	Object4		Object4	Object4
212.267, 534.934	212.267, 534.934		329.379, 535.012	384.639, 535.014
0.416259, 11.0681	0.416259, 11.0681		0.150191, 1.5354	0.241482, 2.08114
157.486	157.486		471.239	471.239
Object5	Object5		Object5	Object5
123.887, 512.932	123.887, 512.932		185.587, 522.208	100.818, 522.208
0.282165, 7.26049	0.282165, 7.26049		-0.0111868, 0.607991	-0.0186068, 0.21237
-548.335	-548.335		-405.831	-405.831
Object6	Object6		Object6	Object6
482.535, 526.896	482.535, 526.896		493.183, 526.899	396.139, 439.857
0.132491, 0.931821	0.132491, 0.931821		0.00477879, 0.41026	-0.0927713, 3.07904
-166.126	-166.126		461.746	326.595
Values match			Values do not match	

10 seconds duration				
Fixed time step			Normal time step	
Test1	Test2		Test1	Test2
Object1	Object1		Object1	Object1
423.403, 530.034	423.403, 530.034		680.02, 530.01	730.008, 530.01
3.33605, -0.682613	3.33605, -0.682613		4.01895e-05, -0.00863029	-0.000501285, -0.0105592
-0.423198	-0.423198		217.535	-315.012
Object2	Object2		Object2	Object2
730.001, 530.037	730.001, 530.037		730.01, 530.01	680.016, 530.01
-0.00134687, -0.358832	-0.00134687, -0.358832		5.12145e-05, -0.00467832	-0.00037313, -0.00620404
0.118664	0.118664		536.904	165.831
Object3	Object3		Object3	Object3
298.257, 504.987	298.257, 504.987		296.742, 505.01	286.991, 530.01
-0.119983, 1.96063	-0.119983, 1.96063		-0.00161718, 0.144345	0.0367888, 1.32378
-157.418	-157.418		-15.708	-31.416
Object4	Object4		Object4	Object4
212.215, 534.952	212.215, 534.952		374.315, 535.01	469.174, 535.01
0.439792, 11.0681	0.439792, 11.0681		0.058332, 0.320741	0.0702157, 0.802725
15.741	15.741		471.239	157.079
Object5	Object5		Object5	Object5
124.159, 512.898	124.159, 512.898		183.391, 522.208	266.527, 472.219
-0.292645, 7.26182	-0.292645, 7.26182		-0.0137087, 0.253546	0.155415, 1.30096
-548.102	-548.102		-40.583	-405.832
Object6	Object6		Object6	Object6
473.76, 526.907	473.76, 526.907		449.307, 531.533	558.227, 531.533
2.10692, 0.32002	2.10692, 0.32002		-0.0583331, 0.123248	-0.0151251, 0.402246
-166.802	-166.802		202.225	202.225
Values match			Values do not match	



5 seconds duration				
Fixed time step			Normal time step	
Test1	Test2		Test1	Test2
Object1	Object1		Object1	Object1
394.511, 530.035	394.511, 530.035		428.613, 490.021	566.082, 530.01
11.7896, -0.682846	11.7896, -0.682846		15.3985, 1.03678	44.5587, -0.0100979
-179.326	-179.326		-164.767	579.355
Object2	Object2		Object2	Object2
730.005, 530.03	730.005, 530.03		730.01, 530.01	730.004, 530.01
-0.0685627, -0.931871	-0.0685627, -0.931871		-1.77725e-05, -0.00469618	-1.91314e-05, -0.00529563
0.0602599	0.0602599		626.573	587.457
Object3	Object3		Object3	Object3
305.159, 505.009	305.159, 505.009		323.753, 500.105	382.326, 489.981
-9.35489, 6.71784	-9.35489, 6.71784		1.16393, 1.3515	-2.78634, 2.72046
-157.198	-157.198		-0.917066	-0.00102786
Object4	Object4		Object4	Object4
212.552, 539.74	212.552, 539.74		388.815, 535.01	386.607, 534.966
-26.3445, 146.785	-26.3445, 146.785		0.120553, 2.27332	-1.88065, 0.72084
156.732	156.732		471.241	471.136
Object5	Object5		Object5	Object5
164.649, 488.894	164.649, 488.894		94.2752, 505.273	137.889, 507.321
-27.786, 76.2881	-27.786, 76.2881		-0.0387942, 0.00263006	-0.0915995, 0.53518
-425.945	-425.945		-181.021	167.534
Object6	Object6		Object6	Object6
444.3, 526.921	444.3, 526.921		473.579, 528.339	461.397, 471.905
7.06869, 3.14651	7.06869, 3.14651		-29.9542, 16.7817	27.6764, 118.115
-166.997	-166.997		548.387	0.0116358
Values match			Values do not match	

Figure 33: Determinism experiment results

My expected outcome for this experiment was that using a normal time step was going to break the determinism, and when using a fixed time step the simulation would prove to be deterministic.

After analyzing the results, the outcome I expected proved to be correct.

The breaking of determinism when using a normal time step was already visible by eye when observing the 2 different simulations. The objects behaved differently and collided differently each simulation.

From these results we can conclude that when using a fixed time step, we can make our simulation deterministic when not using multi-threading or random numbers.

3.3 ACCURACY

My original plan to conduct this experiment was to start a simulation with all variables noted down. This way I would know what objects were going to be simulated with their specific variables. After that I would write down all needed formulas and manually calculate the results after a specific time. That way I could compare the mathematical results and the result of my simulation to see if it was close to each other or not.

After implementing collisions, I found out that this was not needed anymore because I could already tell by eye it was not even accurate to simulation collisions. This is the result of me not being able to make a proper simulation to conduct my experiment in.

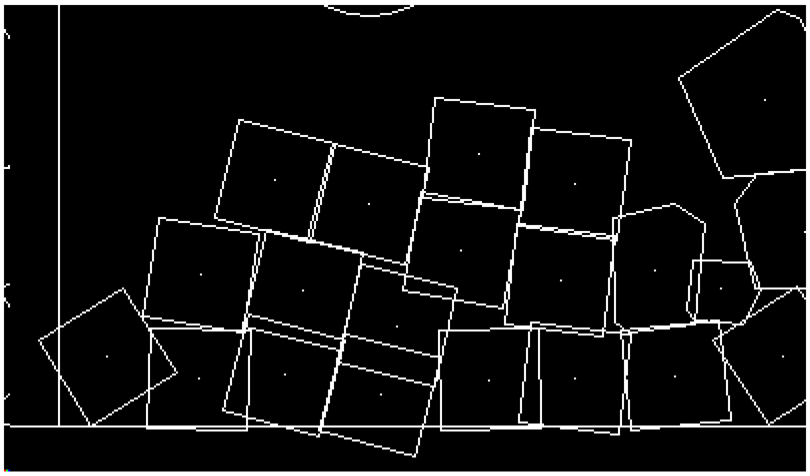


Figure 34: Unstable/Clipping boxes

Because I didn't want to drop this experiment, I researched on how to make simulations more stable/accurate, and what variables, techniques make a simulation more stable/accurate.

I already knew that different physics equations will have an impact on accuracy, but if you don't have an accurate simulation yet, this will not help.

After some further research I found out that one of the main reasons my simulation was not stable, is because I didn't implement constraints, and that my polygons only allow for one contact point at a time.

In order to have a more stable simulation where I could maybe try to run this experiment, I would need to implement constraints and let my polygons support multiple contacts points [4] [11] [27].

Currently when I simulate a stack of boxes, the boxes start shaking and then fall off each other. This is because the force that gets applied to the box only comes from one point, instead of the points of the edge.

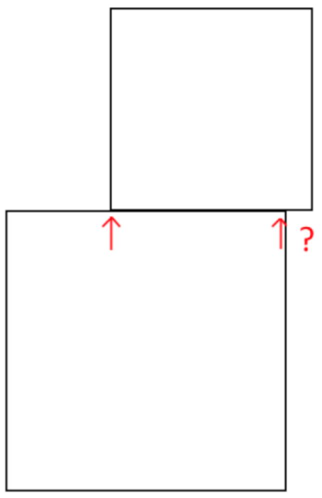


Figure 35: Unstable boxes

If multiple contact points are supported the boxes won't start shaking anymore when on top of each other [4] [15].

The most important addition is having constraints. After researching constraints, I found out that the pure essence of constraints was that they define certain rules between objects and restrict their movement.

The most important constraint I would need to implement would be a non-penetration constraint. This is defining a rule where rigid bodies need to solve their penetration before they can move on, this would be implemented by caching the collision data and checking after the collisions if they are still penetrating [4] [15] [11].

Because of a limited time, and personal learning curve, I was not able to implement this in time to have a stable environment to conduct this experiment.

The expected results of this experiment were that the particle physics system would be accurate. However, when collisions would be implemented this would result in more of an approximation.

The results now after researching would be that the particle physics would be close to accurate, but it would depend on what physics equation would be implemented **(Physics Equations)**.

For collisions the results would have to be that the more constraints you can apply, the more accurate it would behave, but it would still be very hard to calculate if they are exactly accurate to real-time collisions.

At this moment, this is more a speculation than actual results as I would still need to conduct an experiment to get the satisfied results.

DISCUSSION

The results of my experiments can be partially explained by my theoretical framework. The performance results can be explained by the different calculations I had to implement to resolve collisions. The results conclude that when having more complex shapes a loss of performance occurs. When experimenting with different physics equations there was little to no difference in performance. Maybe when simulating larger amounts of objects we could conduct a better experiment, however my resources were limited.

The complexity results conclude the same as the performance results. However, the physics equations had a difference in CPU usage, like expected from the theoretical framework. The CPU needs to do more calculations for each body, even if they are not colliding because the physics equation applies to the bodies' position, velocity, and rotation that need to be calculated every frame. So, when using a different physics equations, the CPU will have a difference in complexity.

The determinism results were also as expected, when using a normal time step it would break the determinism, and when using a fixed time step, it will make the simulation deterministic under the condition of prohibiting the use of random numbers and multi-threading as researched.

The failed accuracy experiment showed that when introducing more complex shapes it becomes harder to get an accurate simulation, and that it is hard to determine if something is truly accurate or not. I was not able to test if the different physics equations were going to help my simulation be more accurate or not.

CONCLUSION

After looking back on the research question and hypotheses, we can conclude that introducing more complex shapes in your simulation results in more complexity, less accuracy, less performance, and require a better understanding of how physics engines work and what optimizations can be applied to solve these issues.

The merits of my research for the academic field can be utilized for people that want to get into physics programming and get a better understanding of how physics engines work. This research symbolizes my learning curve on how I got more knowledge and experience in this field.

FUTURE WORK

As for future directions of my project and research, I would make my simulation more stable, with the usage of constraints [27]. That way the simulation lives under a set of rules, where the objects can no longer penetrate each other. When this is implemented, I could conduct the accuracy test with the different physics equations and have results to prove my hypothesis. Other directions include more optimizations like spatial portioning, optimizing rendering, adding multi-threading, incorporating the physics engine into a 2D game engine to try to make a custom 2D version of Polybridge for example [28].

CRITICAL REFLECTION

I am content that this assignment gave me the opportunity to learn more about the topic I have been interested in lately. I certainly want to continue working on my project after this assignment to get even more experience and knowledge of this broad topic.

I have also learned more time management skills when doing an independent study and gained more experience in academic writing because I had little to no experience with this.

LIST OF FIGURES

REFERENCES

[1] ‘Convex Polygon - Definition, Formulas, Properties, Examples’, Cuemath. Accessed: Jan. 14, 2024. [Online]. Available: https://www.cuemath.com/geometry/convex/

[2] ‘Polygon (Convex)’. Accessed: Jan. 14, 2024. [Online]. Available: https://www.learnalberta.ca/content/memg/division03/Polygon%20(Convex)/index.html

[3] ‘Runge–Kutta methods’, *Wikipedia*. Dec. 15, 2023. Accessed: Jan. 14, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Runge%E2%80%93Kutta_methods&oldid=1189959645

[4] ‘Creating a Game Physics Engine with C++’. Accessed: Jan. 14, 2024. [Online]. Available: https://pikuma.com/courses/game-physics-engine-programming

[5] ‘Runge–Kutta methods’, *Wikipedia*. Dec. 15, 2023. Accessed: Jan. 14, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Runge%E2%80%93Kutta_methods&oldid=1189959645

[6] ‘Visualizing the Runge-Kutta Method — Harold Serrano - Game Engine Developer’, Harold Serrano. Accessed: Jan. 14, 2024. [Online]. Available: https://www.haroldserrano.com/blog/visualizing-the-runge-kutta-method

[7] *Verlet Integration*, (Feb. 15, 2023). Accessed: Jan. 14, 2024. [Online Video]. Available: https://www.youtube.com/watch?v=-GWTDhOQU6M

[8] ‘Verlet integration’, *Wikipedia*. Apr. 24, 2023. Accessed: Jan. 14, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Verlet_integration&oldid=1151499884

[9] ‘Video Game Physics Tutorial - Part I: An Introduction to Rigid Body Dynamics | Toptal®’, Toptal Engineering Blog. Accessed: Jan. 14, 2024. [Online]. Available: https://www.toptal.com/game/video-game-physics-part-i-an-introduction-to-rigid-body-dynamics

[10] ‘Rigid body’, *Wikipedia*. Sep. 28, 2023. Accessed: Jan. 14, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Rigid_body&oldid=1177643890

[11] ‘How to Create a Custom 2D Physics Engine: Oriented Rigid Bodies | Envato Tuts+’, Code Envato Tuts+. Accessed: Jan. 14, 2024. [Online]. Available: https://code.tutsplus.com/how-to-create-a-custom-2d-physics-engine-oriented-rigid-bodies--gamedev-8032t

[12] ‘Broad and narrow phases’. Accessed: Jan. 14, 2024. [Online]. Available: https://lavalle.pl/vr/node262.html

[13] *Broad Phase & Narrow Phase - Let’s Make a Physics Engine [22]*, (Aug. 31, 2022). Accessed: Jan. 14, 2024. [Online Video]. Available: https://www.youtube.com/watch?v=2tS-3C8ty1I

[14] ‘Tips for developing a Collision Detection System — Harold Serrano - Game Engine Developer’, Harold Serrano. Accessed: Jan. 14, 2024. [Online]. Available: https://www.haroldserrano.com/blog/tips-for-developing-a-collision-detection-system

[15] E. Catto, ‘Fast and Simple Physics using Sequential Impulses’.

[16] ‘How to Create a Custom 2D Physics Engine: The Basics and Impulse Resolution | Envato Tuts+’, Code Envato Tuts+. Accessed: Jan. 14, 2024. [Online]. Available: https://code.tutsplus.com/how-to-create-a-custom-2d-physics-engine-the-basics-and-impulse-resolution--gamedev-6331t

[17] ‘OpenGL Mathematics’. Accessed: Jan. 14, 2024. [Online]. Available: https://glm.g-truc.net/0.9.9/index.html

[18] ‘Simple DirectMedia Layer - Homepage’. Accessed: Jan. 14, 2024. [Online]. Available: https://www.libsdl.org/

[19] ‘2D circle and OBB moments of inertia’, GameDev.net. Accessed: Jan. 14, 2024. [Online]. Available: https://gamedev.net/forums/topic/682689-2d-circle-and-obb-moments-of-inertia/5312918/

[20] J. Alexiou, ‘Answer to “Computing tensor of Inertia in 2D”’, Stack Overflow. Accessed: Jan. 14, 2024. [Online]. Available: https://stackoverflow.com/a/41618980

[21] ‘sphere-sphere-2.png (683×562)’. Accessed: Jan. 14, 2024. [Online]. Available: https://turanszkij.files.wordpress.com/2020/04/sphere-sphere-2.png

[22] Mikejo5000, ‘Analyze CPU usage in the Performance Profiler - Visual Studio (Windows)’. Accessed: Jan. 14, 2024. [Online]. Available: https://learn.microsoft.com/en-us/visualstudio/profiling/cpu-usage?view=vs-2022

[23] ‘Fix Your Timestep!’, Gaffer On Games. Accessed: Jan. 14, 2024. [Online]. Available: https://gafferongames.com/post/fix_your_timestep/

[24] jvn91173, ‘How can I perform a deterministic physics simulation?’, Game Development Stack Exchange. Accessed: Jan. 14, 2024. [Online]. Available: https://gamedev.stackexchange.com/q/174320

[25] ‘Game Engines and Determinism’. Accessed: Jan. 14, 2024. [Online]. Available: https://www.duality.ai/blog/game-engines-determinism

[26] ‘Low-latency, determinism, and multithreading’, GameDev.net. Accessed: Jan. 14, 2024. [Online]. Available: https://gamedev.net/forums/topic/693401-low-latency-determinism-and-multithreading/5362173/

[27] ‘Video Game Physics Tutorial - Part III: Constrained Rigid Body Simulation | Toptal®’, Toptal Engineering Blog. Accessed: Jan. 14, 2024. [Online]. Available: https://www.toptal.com/game/video-game-physics-part-iii-constrained-rigid-body-simulation

[28] ‘Poly Bridge by Dry Cactus’. Accessed: Jan. 14, 2024. [Online]. Available: http://polybridge.drycactus.com/

ACKNOWLEDGEMENTS

I would like to thank Maïté Lejeune for helping me formulate my thoughts on paper, proofreading, and coaching me in academic writing.

I would also like to thank Lee Vangraefschepe on giving me advice on how to benchmark performance, because he had some prior benchmarking experience.

Finally, a big thanks to my supervisor Alex Vanden Abeele for giving me feedback.

APPENDICES

The following GitHub link contains my full visual studio project and my excel sheet with all my results, data, and graphs.
https://github.com/Twannes-Claes/CPP-2D-Physics-Engine