UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 3043

3D PHYSICS ENGINE

Filip Husnjak

Zagreb, June 2022

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 3043

3D PHYSICS ENGINE

Filip Husnjak

Zagreb, June 2022

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Zagreb, 11 March 2022

MASTER THESIS ASSIGNMENT No. 3043

Student:	**Filip Husnjak (0036506711)**
Study:	Computing
Profile:	Computer Science
Mentor:	prof. Željka Mihajlović

Title: **3D Physics Engine**

Description:

Explore physical foundations of the dynamic behavior of a solid body. Explore various collision detection techniques. Develop program implementation of 3D collision detection of objects and physically based reactions to the collision. Implement a simulation model that demonstrates collisions of objects in a scene. Provide testing on a series of examples. Discuss the influence of parameters of the models on the output. Evaluate the results and the implemented algorithms. Implement the appropriate software solution. Use C++ programming language and graphics and computing API OpenGL. Make the results of the thesis available online. Supply algorithms, source codes and results with appropriate explanations and documentation. Reference the used literature and acknowledge received help.

Submission date: 27 June 2022

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 11. ožujka 2022.

DIPLOMSKI ZADATAK br. 3043

Pristupnik:	**Filip Husnjak (0036506711)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentorica:	prof. dr. sc. Željka Mihajlović

Zadatak: **Fizikalni pogon za 3D objekte**

Opis zadatka:

Proučiti fizikalne osnove dinamičkog ponašanja čvrstog tijela. Proučiti razne tehnike detekcije kolizije. Razraditi programsku implementaciju ostvarivanja sudara čvrstih tijela te fizikalno temeljene reakcije na sudar. Implementirati simulacijski model koji demonstrira sudare objekata u sceni. Načiniti testiranje na nizu primjera. Analizirati i ocijeniti ostvarene rezultate. Diskutirati upotrebljivost ostvarenih rezultata kao i moguća proširenja. Izraditi odgovarajući programski proizvod. Koristiti programski jezik C++ i grafičko programsko sučelje OpenGL. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 27. lipnja 2022.

CONTENTS

1. Introduction

One of the essential characteristics of any video game is the ability of the user to interact with the surrounding objects in the scene, which results in expected and somewhat realistic behavior. In many cases, that can be achieved with custom-made animations captured by motion trackers, which then transfer sensed data to an application for further processing. Animations are saved as collections of snapshots, where each snapshot stores the corresponding time stamp when it occurred, and the positions and rotations of all the tracked objects in the scene at that particular time.
The first and obvious problem with that approach arises when the user's interactions with the objects are not predictable. Therefore, the animations cannot be prerecorded. In that case, we need a robust system that will correctly simulate the behavior in all scenarios. This is where the physics-based animation comes into play. It is an animation produced by numerical computations applied to the theoretical laws of physics. There are many types of physics-based animations, of which the most notable ones are: *rigid body simulation*, *soft body simulation*, *fluid simulation* and *particle systems*. In this paper, the focus will be aimed towards *rigid body simulation*.

Physics Engine is the core software component that provides an approximate simulation of specific physical systems. It receives a specification of the bodies that will be simulated, as well as some additional configuration parameters, and steps the simulation forward. Each step advances the simulation by a fraction of the second calculating numerical solutions to equations of motion.
The following figure 1.1 gives a high-level overview of the phases the physics engine goes through before the objects can be displayed.

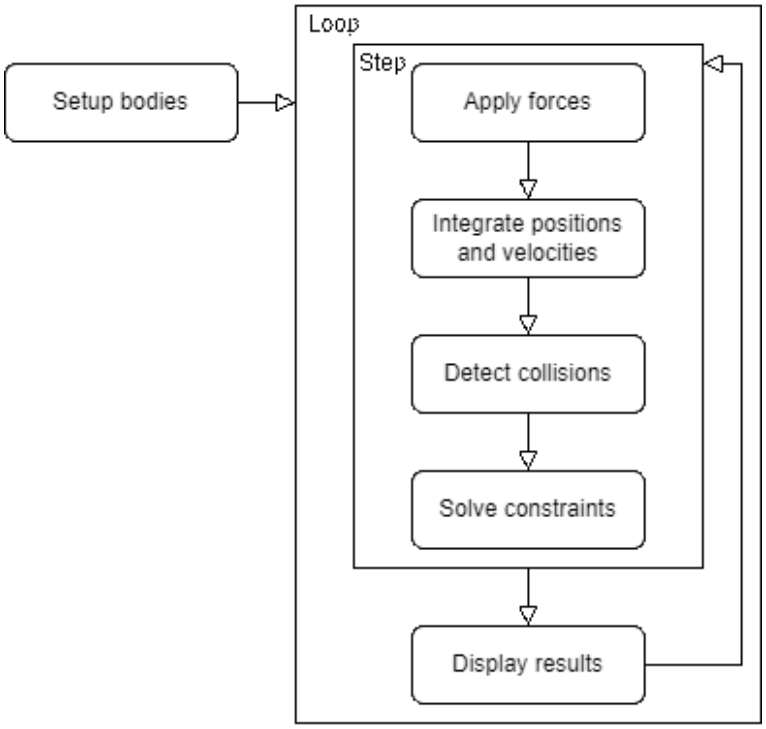

Figure 1.1: Physics Engine Phases

This paper explores rigid body simulation and interactions between the bodies through collisions and constraints. It covers the integration method and different collision algorithms and analyzes all the phases and optimization techniques that are commonly used in physics engines nowadays.

2. Quaternions

In computer graphics, we use transformation matrices to represent the translation, rotation, and scale of the objects in the scene. However, there is an alternative way of describing spatial rotations in three-dimensional space, which is more compact, efficient, and numerically stable. We can represent three-dimensional rotations by using a *quaternion number system*. *Quaternions* are a generalization of two dimensional complex numbers to four dimensions and are represented in the form $a + bi + cj + dk$. They encode the information about the axis of rotation and an angle around that axis. The Irish mathematician Sir William Rowan Hamilton realized the concept of quaternions on Monday, October 16th, 1843, in Dublin, Ireland. He made an important realisation that he immediately carved into the stone of the bridge: $i^2 = j^2 = k^2 = ijk = -1$[5].

Quaternions are an essential part of the physics engine since they avoid two significant concerns surrounding the traditional approach of representing rotations with *Euler angles*. Although *Euler angles* are more intuitive and easier to work with, they can lead to a phenomenon called gimbal lock. Gimbal lock is usually the result of rotating the object by 90 degrees around one of the axes, which causes the loss of one degree of freedom, "locking" the system into rotation in a degenerate two-dimensional space. Another problem that *Euler angles* struggle to deal with is interpolation. Instead of linearly interpolating two rotations represented by three-dimensional vectors, which results in a very unrealistic movement, *quaternions* introduce another way of rotation interpolation called spherical linear interpolation (*slerp*). The difference between *slerp* and *lerp* can be seen in the figure 2.1.

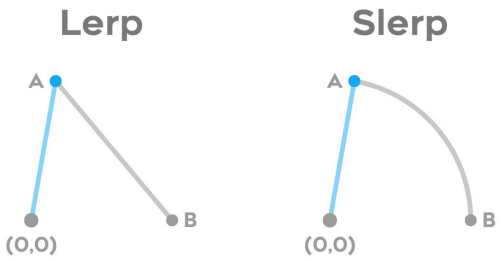

Figure 2.1: Lerp vs Slerp

2.1. Quaternion Math

As already mentioned the general form to express quaternion is:

$$q = s + xi + yj + zk \; s, x, y, z \in \mathbb{R}, \tag{2.1}$$

where according to Hamilton's famous expression:

$$\begin{aligned} i^2 = j^2 = k^2 = ijk = -1 \\ ij = k, \; jk = i, \; ki = j \\ ji = -k, \; kj = -i, \; ik = -j. \end{aligned} \tag{2.2}$$

Imaginary part of the quaternion can also be represented as a three dimensional vector where i, j, k correspond to x, y, z axes in Cartesian coordinate system.
To expose similarities between complex numbers and quaternions we can also represent them as an ordered pair of the real and imaginary part:

$$\begin{aligned} q = [s, \mathbf{v}] \; s \in \mathbb{R}, \mathbf{v} \in \mathbb{R}^3, \\ q = [s, x\mathbf{i} + y\mathbf{j} + z\mathbf{k}] \; s, x, y, z \in \mathbb{R}. \end{aligned} \tag{2.3}$$

2.1.1. Addition and Subtraction

$$\begin{aligned} q_a &= [s_a, \mathbf{a}] \\ a_b &= [s_b, \mathbf{b}] \\ q_a + q_b &= [s_a + s_b, \mathbf{a} + \mathbf{b}] \\ q_a - q_b &= [s_a - s_b, \mathbf{a} - \mathbf{b}] \end{aligned} \tag{2.4}$$

2.1.2. Multiplication

$$
\begin{aligned}
q_a &= [s_a, \mathbf{a}] \\
a_b &= [s_b, \mathbf{b}] \\
q_a q_b &= [s_a, \mathbf{a}][s_b, \mathbf{b}] \\
&= (s_a + x_a i + y_a j + z_a k)(s_b + x_b i + y_b j + z_b k) \\
&= (s_a s_b - x_a x_b - y_a y_b - z_a z_b) + \\
&\quad (s_a x_b + s_b x_a + y_a z_b - y_b z_a) i + \\
&\quad (s_a y_b + s_b y_a + z_a x_b - z_b x_a) j + \\
&\quad (s_a z_b + s_b z_a + x_a y_b - x_b y_a) k \\
&= [s_a s_b - x_a x_b - y_a y_b - z_a z_b, \\
&\quad s_a(x_b i + y_b j + z_b k) + s_b(x_a i + y_a j + z_a k) + \\
&\quad (y_a z_b - y_b z_a) i + (z_a x_b - z_b x_a) j + (x_a y_b - x_b y_a) k].
\end{aligned}
\tag{2.5}
$$

We can substitute:

$$
\begin{aligned}
\mathbf{a} &= x_a i + y_a j + z_a k \\
\mathbf{b} &= x_b i + y_b j + z_b k \\
\mathbf{a} \cdot \mathbf{b} &= x_a x_b + y_a y_b + z_a z_b \\
\mathbf{a} \times \mathbf{b} &= (y_a z_b - y_b z_a) i + (z_a x_b - z_b x_a) j + (x_a y_b - x_b y_a) k,
\end{aligned}
\tag{2.6}
$$

so, we end up with:

$$
[s_a, \mathbf{a}][s_b, \mathbf{b}] = [s_a s_b - \mathbf{a} \cdot \mathbf{b}, s_a \mathbf{b} + s_b \mathbf{a} + \mathbf{a} \times \mathbf{b}]. \tag{2.7}
$$

2.1.3. Multiplication by Scalar

$$
\begin{aligned}
q &= [s, \mathbf{v}] \\
\lambda q &= \lambda [s, \mathbf{v}] \\
&= \lambda [s, \mathbf{v}] \\
&= [\lambda s, \lambda \mathbf{v}].
\end{aligned}
\tag{2.8}
$$

2.1.4. Real Quaternion

Real quaternion is a quaternion with imaginary part equal to zero:

$$
q = [s, \mathbf{0}]. \tag{2.9}
$$

2.1.5. Pure Quaternion

Pure quaternion is a quaternion with its real part equal to zero:

$$q = [0, \mathbf{v}]. \tag{2.10}$$

2.1.6. Unit Quaternion

Unit quaternion is special case of pure quaternion where vector $\mathbf{v}$ is unit vector:

$$q = [0, \hat{\mathbf{v}}]. \tag{2.11}$$

2.1.7. Quaternion Conjugate

Similar to complex numbers, quaternion conjugate is computed by negating the imaginary part:

$$\begin{aligned} q &= [s, \mathbf{v}]. \\ q^* &= [s, -\mathbf{v}]. \end{aligned} \tag{2.12}$$

2.1.8. Quaternion Norm

$$\begin{aligned} q &= [s, \mathbf{v}]. \\ |q| &= \sqrt{s^2 + \mathbf{v} \cdot \mathbf{v}}. \end{aligned} \tag{2.13}$$

Normalized quaternion can be computed as:

$$q' = \frac{q}{|q|}. \tag{2.14}$$

2.1.9. Inverse

From the definition of the inverse:

$$qq^{-1} = [1, 0] = 1. \tag{2.15}$$

We can multiply both sides by conjugate of the quaternion to get the expression for the inverse:

$$\begin{aligned} q^* q q^{-1} &= q^* \\ |q|^2 q^{-1} &= q^* \\ q^{-1} &= \frac{q^*}{|q|^2}. \end{aligned} \tag{2.16}$$

2.2. Describing Rotations with Quaternions

In 2D we can use complex numbers to define rotors which can be used to rotate a point in a 2D complex plane by angle ϕ:

$$z = \cos\phi + i\sin\phi. \tag{2.17}$$

If we have a point p with (x, y) coordinates, we can compute coordinates of a rotated point p' by multiplying those coordinates with that rotor:

$$\begin{aligned}\mathbf{p}' &= (x + yi) \cdot z = (x + yi) \cdot (\cos\phi + i\sin\phi) \\ &= (x\cos\phi - y\sin\phi) + (x\sin\phi + y\cos\phi)i \\ &= (x\cos\phi - y\sin\phi, x\sin\phi + y\cos\phi).\end{aligned} \tag{2.18}$$

Similar to complex numbers, in 3D we can define rotors using quaternions that describe the angle ϕ and unit axis $\hat{\mathbf{v}}$ around which we are rotating the point:

$$q = [\cos\phi, \hat{\mathbf{v}}\sin\phi]. \tag{2.19}$$

It can be shown that in order to rotate an arbitrary point p in a three dimensional space by quaternion $\mathbf{q}$, we have to use the following equation:

$$\mathbf{p}' = q\mathbf{p}q^{-1}. \tag{2.20}$$

The rotation is clockwise if our line of sight points in the same direction as $\hat{\mathbf{v}}$.

2.2.1. Quaternion Derivative

An orientation is described by $q(t+\Delta t)$ at a time $t+\Delta t$. This is after a rotation change Δq during Δt seconds is performed on the local frame. This rotation change about the axis $\mathbf{v} = \frac{\boldsymbol{\omega}}{||\boldsymbol{\omega}||}$ through the angle $\phi = ||\boldsymbol{\omega}||\Delta t$ can be described by a quaternion:

$$\begin{aligned}\Delta q &= \cos\frac{\phi}{2} + \mathbf{v}\sin\frac{\phi}{2} \\ &= \cos\frac{||\boldsymbol{\omega}||\Delta t}{2} + \frac{\boldsymbol{\omega}}{||\boldsymbol{\omega}||}\sin\frac{||\boldsymbol{\omega}||\Delta t}{2}.\end{aligned} \tag{2.21}$$

Since $q(t + \Delta t)$ can be calculated as $\Delta q q(t)$ we can write:

$$\begin{aligned}q(t + \Delta t) - q(t) &= (\cos\frac{\phi}{2} + \mathbf{v}\sin\frac{\phi}{2})q - q \\ &= (-2\sin^2\frac{||\boldsymbol{\omega}||\Delta t}{4} + \frac{\boldsymbol{\omega}}{||\boldsymbol{\omega}||}\sin\frac{||\boldsymbol{\omega}||\Delta t}{2})q.\end{aligned} \tag{2.22}$$

while the derivative then equals to:

$$\begin{aligned}
\dot{q} &= \lim_{\Delta t \to 0} \frac{q(t+\Delta t) - q(t)}{\Delta t} \\
&= \lim_{\Delta t \to 0} \frac{1}{\Delta t}(-2\sin^2 \frac{||\boldsymbol{\omega}||\Delta t}{4} + \frac{\boldsymbol{\omega}}{||\boldsymbol{\omega}||}\sin \frac{||\boldsymbol{\omega}||\Delta t}{2})q \\
&= \frac{\boldsymbol{\omega}}{||\boldsymbol{\omega}||} \lim_{\Delta t \to 0} \frac{1}{\Delta t}\sin(\frac{||\boldsymbol{\omega}||\Delta t}{2})q \\
&= \frac{1}{2}\boldsymbol{\omega} q.
\end{aligned} \tag{2.23}$$

3. Rigid Body Dynamics

A rigid body is an idealization of a body that does not deform or change shape. Formally it is defined as a collection of particles with the property that the distance between particles remains unchanged during the course of motions of the body [2].
The motion of the rigid bodies can be modeled using Newtonian mechanics, which is founded upon Isaac Newton's three laws of motion:

- inertia - every object in a state of uniform motion will remain in that state of motion unless an external force acts on it,
- force, mass, and acceleration - force equals mass times acceleration,
- action and reaction - every action has an equal and opposite reaction.

By applying these three laws of motion, we can recreate the very realistic behavior of the objects in a simulated scene.

3.1. Center of Mass

The Center of mass is a hypothetical point where the entire mass of an object may be assumed to be concentrated. It is calculated as the mean location of a distribution of mass in space:

$$\mathbf{R} = \frac{1}{M} \iiint_V \rho(\mathbf{r})\mathbf{r}dV, \tag{3.1}$$

where M is total mass of the object and $\rho(\mathbf{r})$ is density of the body at the location $\mathbf{r}$ [7].
With the definition of center of mass, we can define the state of the rigid body with the following variables:

- $\mathbf{p}$ - location of the center of mass in world coordinates,
- q - rotation of the rigid body around its center of mass,
- $\mathbf{v}$ - linear velocity at the center of mass location,
- $\mathbf{w}$ - angular velocity of the body around its center of mass.

3.2. Equations of Motion

Equations of motion describe the body's behavior in terms of its motion as a function of time. The formulas define the body's position, velocity, and acceleration relative to a given frame of reference.

3.2.1. Linear Motion

Newton's second law states that the force F acting on a body is equal to the mass multiplied by the acceleration:

$$\mathbf{F} = m\mathbf{a}. \tag{3.2}$$

This is the starting point from which all the other equations can be derived. Acceleration describes the rate of change of velocity with respect to time, so the velocity can be calculated by integrating the acceleration term:

$$\begin{aligned} \mathbf{a} &= \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} = \dot{\mathbf{v}}, \\ \int_{v_0}^{v} d\mathbf{v} &= \int_{0}^{t} \mathbf{a} dt. \end{aligned} \tag{3.3}$$

We can derive the acceleration from the equation 3.2:

$$\begin{aligned} \int_{v_0}^{v} d\mathbf{v} &= \int_{0}^{t} \frac{\mathbf{F}}{m} dt \\ \mathbf{v} - \mathbf{v_0} &= \frac{\mathbf{F}}{m} t. \end{aligned} \tag{3.4}$$

Finally, we end up with the first equation of motion which describes the velocity of a rigid body:

$$\mathbf{v} = \frac{\mathbf{F}}{m} t + v_0. \tag{3.5}$$

The second equation of motion describes the displacement and can be derived from the first equation. Since velocity represents the derivative of displacement, we can compute the position of a rigid body by integrating the velocity equation: 3.5:

$$\begin{aligned} \int_{p_0}^{p} d\mathbf{p} &= \int_{0}^{t} \mathbf{v} dt \\ &= \int_{0}^{t} (\frac{\mathbf{F}}{m} t + v_0) dt \\ \mathbf{p} - \mathbf{p_0} &= \frac{\mathbf{F}}{2m} t^2 + v_0 t. \end{aligned} \tag{3.6}$$

As a result we get a second equation of motion which refers to the position $\mathbf{p}$ of a rigid body:

$$\mathbf{p} = \frac{\mathbf{F}}{2m}t^2 + v_0 t + \mathbf{p_0}. \tag{3.7}$$

The third equation of motion relates velocity to position, and if the acceleration is constant can be derived in a following way:

$$\begin{aligned} \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}\mathbf{p}} &= \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}\mathbf{p}}\frac{\mathrm{d}t}{\mathrm{d}t} \\ &= \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t}\frac{\mathrm{d}t}{\mathrm{d}\mathbf{p}} \\ &= \mathbf{a}\frac{1}{\mathbf{v}} \end{aligned} \tag{3.8}$$

We can further expand the expression:

$$\begin{aligned} \int_{v_0}^{v} \mathbf{v} d\mathbf{v} &= \int_{s_0}^{s} \mathbf{a} dp, \\ \frac{1}{2}(\mathbf{v}^2 - {\mathbf{v_0}}^2) &= \mathbf{a}(\mathbf{p} - \mathbf{p_0}), \end{aligned} \tag{3.9}$$

finally, we have an expression for the third and last equation:

$$\mathbf{v}^2 = {\mathbf{v_0}}^2 + 2\mathbf{a}(\mathbf{p} - \mathbf{p_0}). \tag{3.10}$$

3.2.2. Angular Motion

When dealing with particles, it is enough to define linear motion equations since all their mass is concentrated in a single point. However, when simulating the motion of a rigid body, we have to consider angular motion around its center of mass. Here, we can introduce the angular properties of a rigid body, which are analogous to its linear properties. In order to rotate, a body needs some angular velocity $\boldsymbol{\omega}$, which is a three-dimensional vector that defines an axis of rotation and the rotation rate. To gain angular velocity, the body needs to receive some rotational force called torque, represented by the greek letter τ. Thus, the Newton's Second Law applied to rotation can be written as:

$$\boldsymbol{\tau} = I\boldsymbol{\alpha}, \tag{3.11}$$

where $\boldsymbol{\alpha}$ represents angular acceleration and I is called moment of inertia [9].

Moment of inertia defines how hard it is to change the angular velocity of a rigid body. For rotations, it is analogous to mass for linear motion. The moment of inertia is

defined as the product of mass m of the section and the square of the distance between the reference axis and centroid of the section r [9]. For a single particle it has the following form:

$$I = m\mathbf{r}^2. \tag{3.12}$$

For a rigid body, we can calculate the moment of inertia by integrating the entire volume, effectively summing $m_i\mathbf{r}^2$ for every infinitely small piece with mass m_i:

$$\iiint_V \rho(x, y, z)\mathbf{r}^2 dV, \tag{3.13}$$

where $\rho(x, y, z)$ represents density of an rigid body at position with coordinates x, y, z, while r is the distance between that point and axis of rotation.

For the same object, different axes of rotation will have different moments of inertia around those axes if the object is not symmetric about all axes. For this reason, we describe the rotational properties of a rigid body using the moment of inertia tensor. In three-dimensional space, a moment of inertia tensor is given by:

$$I = \begin{bmatrix} I_{11} & I_{12} & I_{13} \\ I_{21} & I_{22} & I_{23} \\ I_{31} & I_{32} & I_{33} \end{bmatrix}. \tag{3.14}$$

For a rigid object of N point masses m_k, the components are defined as:

$$I_{ij} = \sum_{k=1}^{N} m_k(\mathbf{r_k}^2\delta_{ij} - x_i^{(k)}x_j^{(k)}), \tag{3.15}$$

where r_k is the vector to the point mass m_k from the point about which the tensor is calculated and δ_{ij} is Kronecker delta [4]. If the axes of the local reference frame are aligned in such a way that the mass of the body is evenly distributed around the axis, thus, the products of inertia all vanish, the off-diagonal elements become 0. In that case, we can define the properties of a rigid body using only diagonal elements of the matrix. The non-zero diagonal elements of the inertia tensor are then called *the principal moments of inertia*.

Similar to linear motion, angular acceleration and angular velocity are related using the following equation:

$$\boldsymbol{\alpha} = \frac{\mathrm{d}\boldsymbol{\omega}}{\mathrm{d}t}, \tag{3.16}$$

while the angular velocity represents the derivative of a rotation ϕ of a rigid body:

$$\boldsymbol{\omega} = \frac{\mathrm{d}\boldsymbol{\phi}}{\mathrm{d}t}. \tag{3.17}$$

3.3. Numerical Integration

Now that we have defined all the equations necessary for simulating unconstrained rigid body motion, we have to somehow translate that into a program that will step the simulation in discrete time steps. The problem is that since we are operating on a discrete domain, obtaining acceleration, velocity, and position values only at specific points, we cannot use a standard analytical approach to find the antiderivative. However, some methods can operate on such data and give satisfactory approximations. Those methods are collectively known as *numerical integration*. The different numerical integration algorithms assume different function behaviors in between discrete time steps.

3.3.1. Semi-Implicit Euler Method

In most physics engines, the simplest numerical integration method is used, known as *semi-implicit Euler*. It assumes that the value of a function we are integrating does not change between discrete time steps t_k and t_{k+1}.
If we know the value of our function at time t_k, then the value at a time t_{k+1} can be calculated as:

$$f(t_{k+1}) = f(t_k) + \dot{f(t_k)}(t_{k+1} - t_k). \tag{3.18}$$

Using the equation 3.18 we can solve equations of motion in the discrete domain. Suppose we know the forces acting on a rigid body and the velocity and position at time t_k. In that case, we can calculate the new velocity and position at time t_{k+1} using the following set of equations:

$$\begin{aligned} \mathbf{a}(t_k) &= \frac{\mathbf{F}}{m} \\ \mathbf{v}(t_{k+1}) &= \mathbf{v}(t_k) + \mathbf{a}(t_k)\Delta t \\ \mathbf{p}(t_{k+1}) &= \mathbf{p}(t_k) + \mathbf{v}(t_{k+1})\Delta t \\ \Delta t &= t_{k+1} - t_k, \end{aligned} \tag{3.19}$$

where $\mathbf{F}$ represents the sum of all the forces acting on a rigid body at time t_k.

Similar to the linear motion, we can calculate new angular velocity by calculating the total torque $\boldsymbol{\tau}$ acting on the body and numerically integrating it using the semi-implicit Euler method:

$$\begin{aligned} \boldsymbol{\alpha}(t_k) &= I^{-1}\boldsymbol{\tau} \\ \boldsymbol{\omega}(t_{k+1}) &= \boldsymbol{\omega}(t_k) + \boldsymbol{\alpha}(t_k)\Delta t. \end{aligned} \tag{3.20}$$

Since we are representing the rotation of a rigid body with quaternions, we can obtain the new rotation of a rigid body using the quaternion derivative equation 2.22:

$$q(t_{k+1}) = \frac{1}{2}\Delta t \omega(t_{k+1}) q(t_k). \tag{3.21}$$

The semi-implicit Euler is a first-order integrator, which means that it commits a global error of the order of Δt, which marks the importance of the size of a Δt.
Another benefit of this integration method is that it almost conserves energy making it far more accurate and stable than other methods.

4. Collision Detection

For now, we have discussed the simulation of unconstrained rigid body motion, meaning we can simulate the bodies correctly by applying forces and calculating new velocities and positions. However, in that discussion, objects did not interact with each other, so, right now, nothing prevents bodies from going through each other, which is undesirable in most cases. In order to realistically simulate rigid body motion, we have to detect collisions between them and respond to those collisions by applying additional forces or impulses which will move them apart.

In this section collision detection phase will be explained. It consists of finding body pairs that are colliding as well as contact points of those colliding pairs. In rigid body simulation, a collision occurs when two bodies intersect or when the distance between those bodies falls below a certain tolerance. Since we have to find all the colliding pairs, the complexity of detecting collisions with pairwise tests would be kN^2, where N represents the number of rigid bodies in the simulation and k is the complexity of detecting a collision between two shapes. Usually, there are many objects in the scene, so the complexity of this naive approach is too expensive for real-time simulations, and other methods have to be employed. In order to optimize the time complexity of the collision detection step, the collision phase is separated into two steps: the *broad phase* and the *narrow phase*.

The objective of the broad phase step is to find all the pairs that are potentially colliding, excluding all the pairs that are certainly not colliding. On the other hand, the narrow phase step operates on the resulting pairs and finds all the contact points between those pairs.

4.1. Broad Phase

As already mentioned, the broad phase step finds all potentially colliding pairs out of all simulated bodies. In order to optimize the complexity of the collision tests in this phase, the bodies are approximated with bounding volumes. If the intersection be-

tween those volumes is not found, we know that the actual shapes also do not intersect. However, if the intersection is found, then the bodies might intersect.

4.1.1. Bounding Volumes

Some popular bounding volumes typically used are *spheres*, *axis-aligned bounding boxes* (AABB), and *oriented bounding boxes* (OBB). The differences can be seen in the figure 4.1.

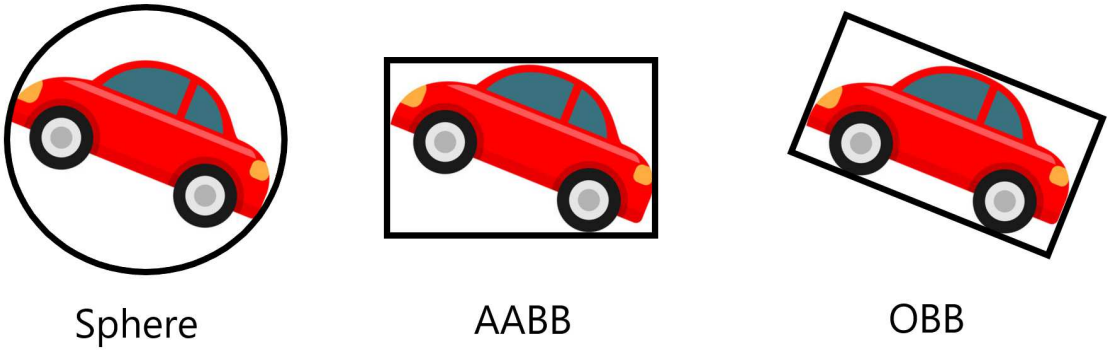

Figure 4.1: Bounding Volume Types

As one can observe, the most straightforward bounding volume is the *sphere*. The benefit is that the algorithm for detecting if two spheres are overlapping is simple and fast. At the same time, the tradeoff is that the approximation is not precise, meaning we will have larger space around the actual object wasted. On the other hand, OBB bounding volume type is the most complex out of the ones listed above, and the algorithm for detecting if two OBBs overlap is not trivial. However, it results in the best approximation of the actual shape.

In physics engine programming most commonly used bounding volumes are *axis-aligned bounding boxes* (AABB) because they are simple and offer good tradeoffs. In three-dimensional space, AABB can be represented with two three-dimensional vectors:

```
struct aabb {
    vec3 min;
    vec3 max;
}
```

The first vector represents the vertex of an AABB with the largest coordinates, while the second vector represents the vertex with the smallest coordinates.

The algorithm 1 is used to detect if two AABBs overlap.

Algorithm 1 AABB Overlap Test

```
1:  procedure TESTOVERLAP(a, b)
2:      if a.max.x < b.min.x or a.min.x > b.max.x then
3:          return false;
4:      else if a.max.y < b.min.y or a.min.y > b.max.y then
5:          return false;
6:      else if a.max.z < b.min.z or a.min.z > b.max.z then
7:          return false;
8:      end if
9:      return true;
10: end procedure
```

Even though the AABB overlap test is cheap, it will not help reduce the time complexity of pairwise tests $O(N^2)$. To minimize the number of AABB overlap tests, the broad phase step uses some kind of space partitioning, which has to be dynamically updated since most of our AABBs are not stationary during the simulation. There are many spatial partitioning data structures such as: *uniform grids*, *octrees* and *spatial hashing*. The data structure discussed in this paper is called *dynamic bounding volume tree*.

4.1.2. Dynamic Bounding Volume Tree

A *dynamic bounding volume tree* is a binary tree where each node bounds all of the AABBs of its children. It consists of internal nodes and leaf nodes. The leaf nodes are the collision objects, while the internal nodes are only used to accelerate the queries. The 2D example of the dynamic bounding volume tree can be seen in the figure 4.2.

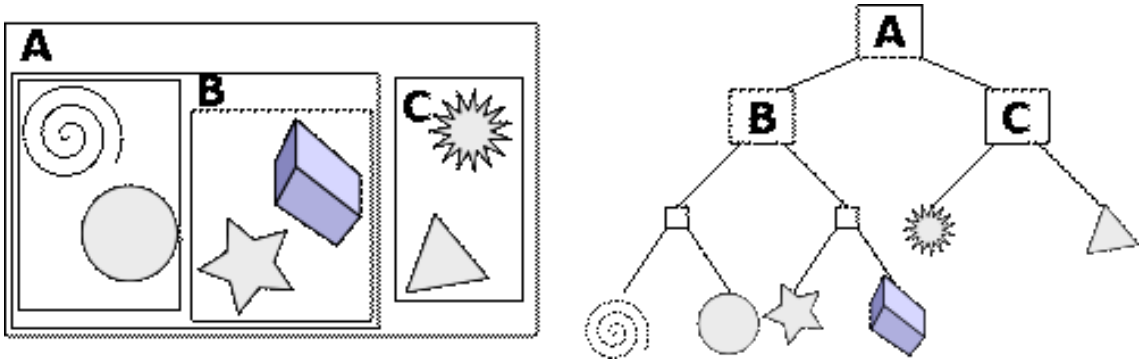

Figure 4.2: Dynamic Bounding Volume Tree

In order to understand the time complexity of the queries, we have to introduce the term *balanced tree*. *Balanced tree* is a binary tree where the difference between the heights of left and right sub-trees of each node never exceeds 1. It is an essential property because if we keep the tree balanced, we can preserve the logarithmic time complexity $O(\log N)$ for the queries.

The type of the self-balancing binary tree used in this paper is called AVL tree [6]. Every node has a balance factor which is defined by the height difference of its two child sub-trees. When the balance factor of a node becomes greater than 1, the tree is re-balanced using rotations [6]. As a result, all trivial operations such as insertion, deletion, and search have logarithmic time complexity $O(\log N)$.

Finally we can represent the tree with the following structures:

```
struct node {
    aabb aabb {};
    int parent;
    int left;
    int right;
    int height;
    bool is_leaf;
};

struct tree {
    node* nodes;
    int node_count;
    int root_index;
};
```

Insertion Algorithm

Insertion algorithm can be split into three stages:

- find the best sibling for the new leaf,
- create a new parent,
- walk back up the tree refitting AABBs.

The first stage is the most complex and the most important one. The method used and explained here for choosing the best sibling for a new node is called *surface area heuristic* (SAH). The idea is that the probability of our object overlapping with the AABB of the node is proportional to the surface area of the AABB. We can define the cost function of a tree as a sum of the surface areas of all AABBs:

$$C(T) = \sum_{i \in Nodes} SA(i). \tag{4.1}$$

The different BVH trees for the same rigid body simulation will only differ in inner nodes since the leaves, and the root node will have the same surface area. For this reason, we can adjust our cost function only to include the surface areas of the inner nodes. This gives us an objective way to compare the quality of the two trees.
Figure 4.3 shows an example tree where the new node is to be inserted.

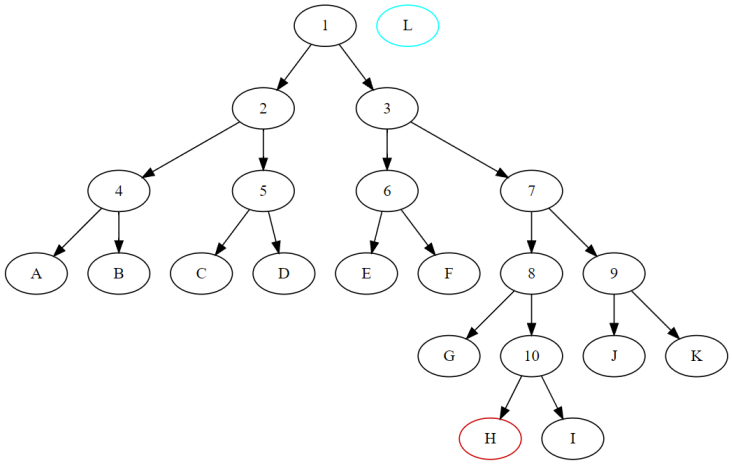

Figure 4.3: BVH Tree Before Insertion

Suppose we are inserting the new node L, and as the new sibling, we chose the node H. The added cost of choosing H as the sibling for L can be calculated as:

$$C_H = SA(11) + \Delta SA(10) + \Delta SA(8) + \Delta SA(7) + \Delta SA(3) + \Delta SA(1), \tag{4.2}$$

where

$$\Delta SA(node) = SA(node \cup L) - SA(node). \tag{4.3}$$

The resulting tree is shown in the figure 4.4.

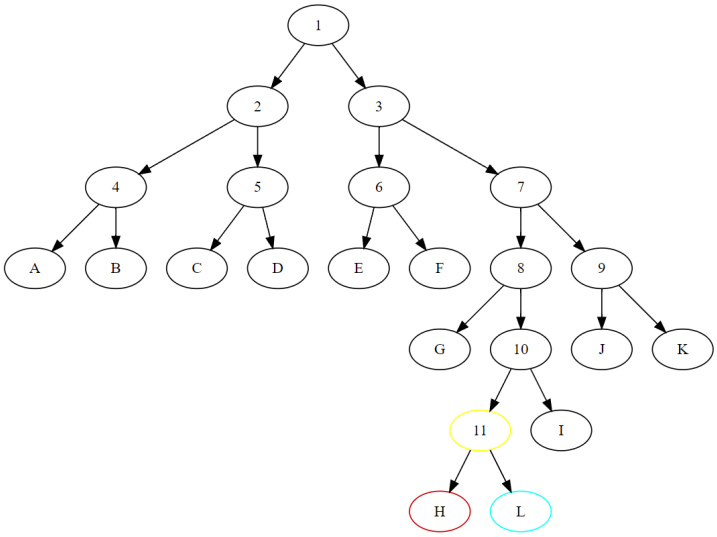

Figure 4.4: BVH Tree After Insertion

The naive approach of finding the best sibling by checking all the potential siblings and calculating the cost is too expensive, so the algorithm called *branch and bound* is used to improve the performance.
The main ideas behind the algorithm are:

- search through the tree recursively,
- skip sub-trees that cannot possibly be better.

In the example above, let us suppose we reach node 7 during the search. The algorithm then calculates the cost of choosing the node 7 as the new sibling using the equation:

$$C_7 = \Delta SA(7) + \Delta SA(3) + \Delta SA(1). \tag{4.4}$$

It compares this value with the costs of continuing the search through either left or the right child. If the cost of choosing node 7 as the new sibling is the smallest, we stop the algorithm, and node 7 becomes the sibling of the new node. Otherwise, depending on their corresponding cost values, we continue the search through either the left or the right sub-tree. When the new sibling is found, we have to rebalance the tree using already mentioned tree rotations and refit the AABBs of all the ancestors of a new node. Finally, the overall time complexity of the insertion algorithm is $O(\log N)$ where N is the number of nodes in the tree.

Query Algorithm

We have to perform the tree query when detecting all the potentially colliding pairs for a single object. It traverses the tree by visiting only nodes that overlap the AABB of the queried object. When the leaf is reached, the body pair, consisting of the object represented by the leaf and the queried body, is added to the list.

Rigid Body Movement

The tree structure might change when the body moves, and the optimal sibling for the body might also change. For this reason, the best approach for handling object movement is to remove and re-insert it back into the tree. To avoid frequent re-insertions for every object, AABBs are usually slightly larger to accommodate minor movements since the body will be re-inserted only when it moves outside the bounding volume.

When using the data structures explained above, the average time complexity of detecting all the potentially colliding pairs for every object is $O(N \log N)$, where N represents the number of bodies in the simulation.

4.2. Narrow Phase

Once we have found all the pairs as a result of the broad phase collision step, we have to find out if they are, in fact, actually colliding and calculate the contact points and collision normal in order to respond to the collision adequately. It is not trivial to determine if the arbitrary two shapes collide, and it is even harder to find the contact points between them. If the intersection is found, this phase results in a collision manifold, which is represented by the following data structure:

```
struct contact {
    vec3 rA; // Collision point within A's local space
    vec3 rB; // Collision point within B's local space
    float penetrationDepth;
}
struct manifold {
    int contactCount;
    contact* contacts;
    vec3 normal;
    vec3 tangents;
}
```

In this paper three collision algorithms are explained:

- sphere - sphere,
- sphere - box,
- box - box.

4.2.1. Sphere - Sphere Collision Algorithm

Sphere - sphere collision algorithm is the simplest and fastest. The spheres collide if the distance between them is less than the sum of their radii. The collision algorithm finds at most 1 contact point.

Algorithm 2 Sphere - Sphere Collision Algorithm

1: **procedure** SPHERESPHERE($sphereA, sphereB, outManifold$)
2: $normal \leftarrow sphereA.location - sphereB.location$;
3: $distance \leftarrow normal.length() - (sphereA.radius + sphereB.radius)$;
4: **if** $distance > 0$ **then**
5: **return** $false$;
6: **end if**
7: $outManifold.contactCount \leftarrow 1$;
8: $outManifold.contacts[0].penetrationDepth \leftarrow distance$;
9: $outManifold.contacts[0].rA \leftarrow sphereA.radius * normal$;
10: $outManifold.contacts[0].rB \leftarrow -sphereB.radius * normal$;
11: $outManifold.normal \leftarrow normal$;
12: **end procedure**

4.2.2. Sphere - Box Collision Algorithm

Sphere - box collision algorithm first finds the closest point on the box to the sphere and afterward checks if the distance between that point and the sphere is less than the sphere's radius. This algorithm also finds at most one contact point.

Algorithm 3 Sphere - Box Collision Algorithm

1: **procedure** SPHEREBOX($sphere, box, outManifold$)
2: $shereRelPos \leftarrow box.transform.inverseTransformLocation(sphere.location)$;
3: $closestPoint \leftarrow shereRelPos$;
4: $closestPoint.x \leftarrow min(box.halfExtents.x, closestPoint.x)$;
5: $closestPoint.x \leftarrow max(-box.halfExtents.x, closestPoint.x)$;
6: $closestPoint.y \leftarrow min(box.halfExtents.y, closestPoint.y)$;
7: $closestPoint.y \leftarrow max(-box.halfExtents.y, closestPoint.y)$;
8: $closestPoint.z \leftarrow min(box.halfExtents.z, closestPoint.z)$;
9: $closestPoint.z \leftarrow max(-box.halfExtents.z, closestPoint.z)$;
10: $closestPoint \leftarrow box.transform.transformLocation(closestPoint)$;
11: $normal \leftarrow closestPoint - sphere.location$;
12: $distance \leftarrow normal.length() - sphere.radius$;
13: **if** $distance > 0$ **then**
14: **return** $false$;
15: **end if**
16: $outManifold.contactCount \leftarrow 1$;
17: $outManifold.contacts[0].penetrationDepth \leftarrow distance$;
18: $outManifold.contacts[0].rA \leftarrow sphere.radius * normal$;
19: $outManifold.contacts[0].rB \leftarrow closestPoint - box.location$;
20: $outManifold.normal \leftarrow normal$;
21: **end procedure**

4.2.3. Box - Box Collision Algorithm

Box - box collision detection algorithm is the most complex of all of the above and will be explained in more detail. The algorithm consists of two parts:

- separating axis theorem (SAT) - used to detect a collision between two boxes,
- clipping - used to find contact points after the collision was detected.

Separating Axis Theorem

Separating axis theorem says that two convex objects do not overlap if there exists an axis onto which the two objects' projections do not overlap. In terms of boxes, it can be shown that we have to check 15 axes in total. If any of those axes are separating, we know that two boxes do not intersect.

The first six axes are called face axes and represent face normals. To simplify the problem of projecting box A onto the axes of box B, all computations can be done within the space of A. We can define the translation vector $\mathbf{t}$ as:

$$\mathbf{t} = r_a^T * (\mathbf{c_b} - \mathbf{c_a}), \tag{4.5}$$

where $\mathbf{c_i}$ denotes the center of a box, and r_a denoted the rotation of a box A. Vector $\mathbf{t}$ points from the $A's$ center to $B's$ center within the space of A. The relative rotation matrix to transform from $B's$ frame to $A's$ frame is defined as:

$$d = r_a^T * r_b. \tag{4.6}$$

Matrix d can be used to transform $B's$ local axes into the frame of A.
To calculate the separation, s along the local axis $\mathbf{l}$, the following equation can be used:

$$s = |\mathbf{t} \cdot \mathbf{l}| - (|\mathbf{e_a} \cdot \mathbf{l}| + |(d \cdot \mathbf{e_b}) \cdot \mathbf{l}|), \tag{4.7}$$

where vector $\mathbf{e_a}$ represents half extents of the box A, while vector $\mathbf{e_b}$ represents the half extents of the box B. Due to the symmetry, a projection's absolute value can be used to calculate the value representing a projection along a given direction. In our case, we are not interested in the sign of the projection and always want the projection toward the other box.

In 2D, face axes are the only axes we have to check. However, in 3D, a new topological feature arises on the surface of geometry: edges. Since edge to edge collision between two boxes cannot be detected using only face axes, we have to include edge

axes somehow. Given the two edges upon two boxes in 3D space, the vector perpendicular to both represents the possible separation axis and can be computed using the cross product. By calculating the cross product of all edge to edge axes combinations, we can obtain all possible axes of separation. Since there are 3 unique edge axes per box, the total count of additional axes we have to check is 9. Hence the total number of possible axes of separation is $6 + 9 = 15$.

Since matrix d is a linear operator and is composed of a concatenation between r_a^T and r_b, the columns of this matrix can be thought of as the local axes of B in a space of A. Let us derive the cross product between $A's$ axis x and $B's$ axis z, all within the space of A. The $A's$ $\mathbf{x}$ axis in the $A's$ local space can be written as $\{1,\ 0,\ 0\}$, while the $B's$ $\mathbf{z}$ axis using the matrix d can be written as $\{d_{02},\ d_{12},\ d_{22}\}$. The axis $\mathbf{l}$ is the cross product between these two axes and can be calculated directly without using the cross product formula as:

$$\mathbf{l} = \{0,\ -d_{22},\ d_{12}\}. \tag{4.8}$$

The same approach can be used to derive all 9 separating edge axes.
If an axis l separates the two boxes, the equation 4.7 will result in a negative value. Contact information can be generated once the axis of least penetration is found.

Contact Point Generation

Contact point generation is the second part of the collision detection algorithm. The contact points are used to resolve the collision during the physical simulation. There are only two cases of collision that should be handled: *edge to edge* and *something to face*.
The *edge to edge* case is resolved by calculating the *supporting edge* which is the edge most pointed to by a given axis of separation. Once the two *supporting edges* have been found, the algorithm determines the point where the edges overlap, which results in a single contact point.
Something to face case refers to either *face to face* or *edge to face* collision. It is a more complicated case and can result in any number of contact points between 1 and 8. In order to resolve this case, two faces from each box have to be calculated, which are by convention named reference and incident face. Once the faces are determined, the incident face is clipped to the side planes of the reference face. The process of clipping can be seen in the figure 4.5.

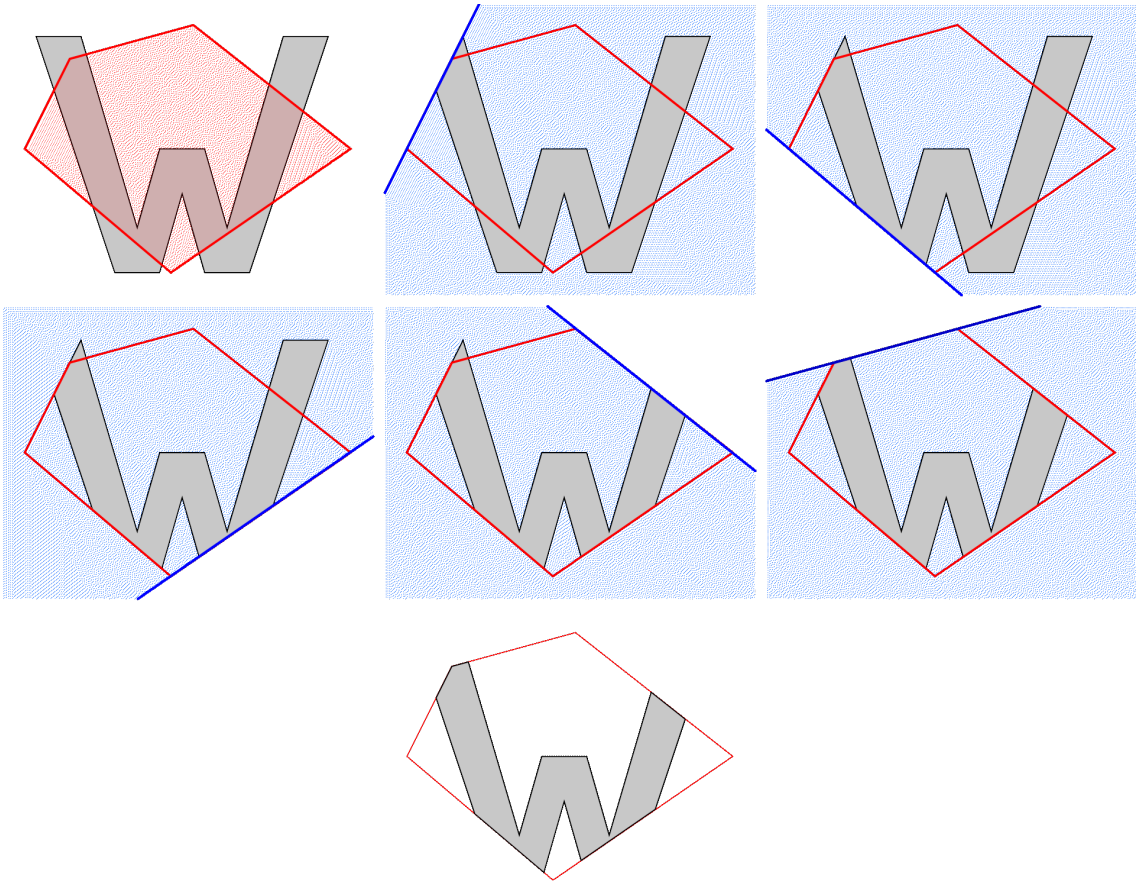

Figure 4.5: Clipping

The clipping is performed using the *Sutherland Hodgman* algorithm. This algorithm takes a list of vertices and planes as an input and outputs the new vertices that exist purely within the set of planes. The clipping results in at most 8 contact points.

The function $InFront$ returns $true$ if the point is on the same side of the plane as the plane normal, while the method $Intersect$ finds the intersection point of the given plane and line.

Algorithm 4 Sutherland Hodgman Clipping

```
1:  procedure CLIP(startingPolygon, clippingPlanes)
2:      output ← startingPolygon ;
3:      for each plane in clippingPlanes do
4:          input ← output;
5:          startingPoint ← input.last();
6:          for each endPoint in input do
7:              if INFRONT(startingPoint, plane) and INFRONT(endPoint, plane)
    then
8:                  output.push(endPoint);
9:              else if INFRONT(startingPoint, plane) then
10:                 output.push(INTERSECT(plane, startingPoint, endPoint));
11:             else if not INFRONT(endPoint, plane) then
12:                 output.push(INTERSECT(plane, startingPoint, endPoint));
13:                 output.push(endPoint);
14:             end if
15:             endPoint = startingPoint;
16:         end for
17:     end for
18:     return output;
19: end procedure
```

5. Constrained Rigid Body Simulation

Up to this point, we have discussed how to simulate rigid body motion and detect when the objects are colliding. However, we still don't have a way to respond to those collisions so that the interactions between the objects can be observed. In other words, we only discussed an unconstrained motion of rigid bodies. As a final step of realistically simulating rigid body motion, we have to introduce *constraints*. *Constraints* enforce a certain body behavior by defining a set of rules that must be satisfied during the simulation. Some real-world examples of constraints that might be part of our simulation are *hinge joints* and *ball joints*. However, the most important constraint, which prevents bodies from interpenetrating, used to resolve collisions is known as *non-penetration constraint*. In a physics engine, *constraint* is essentially a function that can take either acceleration, velocity, or position of a body as an input.

A simple example of a constraint in our simulation might be *the distance between two bodies should be exactly 4*. The equation with which we can define this constraint could look like this:

$$C_1(\mathbf{p_1}, \mathbf{p_2}) = ||\mathbf{p_1} - \mathbf{p_2}||^2 - 16, \tag{5.1}$$

where $\mathbf{p_1}$ represents the position of a first body, while p_2 represents the position of the second body. There are three cases we have to inspect in order to determine if the rule is satisfied or not:

- value of a function C_1 is greater than 0 - the distance between the bodies is greater than 4,
- value of a function C_1 is equal to 0 - the distance between the bodies is exactly 4,
- value of a function C_1 is less than 0 - the distance between the bodies is less than 4.

Looking back at our definition of a constraint, we can conclude that only the second

case satisfies the rule, so our constraint becomes:

$$C_1(\mathbf{p_1}, \mathbf{p_2}) = 0. \tag{5.2}$$

There are two categories of constraints:

- equality constraints - the only acceptable value of a constraint function is 0,
- inequality constraints - the constraint function may take on a broader range of values.

The constraint function in the example above belongs to the category of *equality constraints*. In order to satisfy the rule, we could directly set the positions of the bodies to appropriate values so that their distance is exactly 4. However, that method is not ideal because it may cause jittering and result in unrealistic behavior. There are two approaches to resolving constraints typically used by physics engines:

- force-based approach - the constraints are satisfied by applying corrective forces,
- impulse-based approach - the constraints are satisfied using impulses.

5.1. Force-Based Approach

Since in our simulation, we are moving rigid bodies by applying forces, we could use the same approach to satisfy the constraints. We know that only the forces acting in the direction parallel to the gradient of a constraint function can break the constraint [3]. Thus, the corrective forces must also be parallel to that gradient.
The gradient of the function C_1 can be calculated as:

$$\begin{aligned} \nabla C_1(\mathbf{p_1}, \mathbf{p_2}) &= \{\frac{\partial C_1}{\partial p_1}\dot{p_1}, \frac{\partial C_1}{\partial p_2}\dot{p_2}\} \\ &= \{2p_1(p_1 - p_2)\dot{p_1}, -2p_2(p_1 - p_2)\dot{p_2}\}. \end{aligned} \tag{5.3}$$

In order to ensure that the constraint function will be equal to 0, and remain unchanged throughout the simulation, the gradient has to be 0 as well [3]. Likewise, the gradient will not change if the second derivative of a constraint function is also 0. This is where our constraint forces will be applied, so we do not have to calculate any further derivatives.

5.1.1. System of Constraints

To generalize the equations above, we have to introduce the state vector, which contains the state of all rigid bodies in the system [3]. Let us suppose we are simulating N

rigid bodies. The state of a rigid body contains the position $\mathbf{p}$ and rotation represented with quaternion q. Thus, the state vector can be written as:

$$\mathbf{S} = \begin{bmatrix} \mathbf{p_1} \\ q_1 \\ \mathbf{p_2} \\ q_2 \\ \vdots \\ \mathbf{p_N} \\ q_N \end{bmatrix} = \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_N \end{bmatrix}. \tag{5.4}$$

Masses and moments of inertia for each rigid body can be defined with a following matrix:

$$\mathbf{M} = \begin{bmatrix} m_1 E_{3\times3} & & & & & & \\ & I_1 & & & & & \\ & & m_2 E_{3\times3} & & & & \\ & & & I_2 & & & \\ & & & & \ddots & & \\ & & & & & m_N E_{3\times3} & \\ & & & & & & I_N \end{bmatrix}, \tag{5.5}$$

where $E_{3\times3}$ denotes 3 by 3 indentity matrix, while m_i and I_i represent mass and inertia tensor of the i-th body. Lastly, let's define the force vector, which contains forces and torques acting on each rigid body:

$$\mathbf{F} = \begin{bmatrix} \mathbf{f_1} \\ \boldsymbol{\tau_1} \\ \mathbf{f_2} \\ \boldsymbol{\tau_2} \\ \vdots \\ \mathbf{f_N} \\ \boldsymbol{\tau_N} \end{bmatrix}. \tag{5.6}$$

Each force in the vector is the sum of external and constraint forces, thus, the force vector can be split into two parts:

$$\mathbf{F} = \mathbf{F_{ext}} + \mathbf{F_c}, \tag{5.7}$$

where F_{ext} denotes external forces acting on the rigid bodies, while F_c represents constraint forces.

Now the Newton's second law applied to the whole system can be written as:

$$\mathbf{F} = M\ddot{\mathbf{S}}. \tag{5.8}$$

Finally, let's introduce M constraints in our system. We can group them into a single function that takes the state vector $\mathbf{S}$ as an input:

$$\mathbf{C}(\mathbf{S}) = \begin{bmatrix} C_1(\mathbf{S}) \\ C_2(\mathbf{S}) \\ \vdots \\ C_M(\mathbf{S}) \end{bmatrix}. \tag{5.9}$$

As already mentioned, to keep the constraint functions as close to 0, we have to calculate the first and second derivatives with respect to time. The first derivative can be calculated as:

$$\dot{\mathbf{C}} = \frac{\partial \mathbf{C}}{\partial \mathbf{S}}\dot{\mathbf{S}}. \tag{5.10}$$

We can simplify the equation by introducing the *Jacobian matrix* J. *Jacobian matrix* is a generalization of a gradient and contains all first order partial derivatives of a function [8]:

$$J = \frac{\partial \mathbf{C}}{\partial \mathbf{S}} = \begin{bmatrix} \frac{\partial C_1}{\partial s_1} & \frac{\partial C_1}{\partial s_2} & \cdots \frac{\partial C_1}{\partial s_N} \\ \frac{\partial C_2}{\partial s_1} & \frac{\partial C_2}{\partial s_2} & \cdots \frac{\partial C_2}{\partial s_N} \\ \vdots & \vdots & \ddots \vdots \\ \frac{\partial C_M}{\partial s_1} & \frac{\partial C_M}{\partial s_2} & \cdots \frac{\partial C_M}{\partial s_N} \end{bmatrix}. \tag{5.11}$$

Now equation 5.10 becomes:

$$\dot{\mathbf{C}} = J\dot{\mathbf{S}}. \tag{5.12}$$

The derivative of a state vector $\dot{\mathbf{S}}$ can be thought of as velocity vector representing linear and angular velocities of all bodies in the system:

$$\dot{\mathbf{S}} = \mathbf{V} = \begin{bmatrix} \mathbf{v_1} \\ \boldsymbol{\omega_1} \\ \mathbf{v_2} \\ \boldsymbol{\omega_2} \\ \vdots \\ \mathbf{v_N} \\ \boldsymbol{\omega_N} \end{bmatrix}. \tag{5.13}$$

The second derivative of a constraint function $\mathbf{C}$ can be calculated using the multiplication rule:

$$\ddot{\mathbf{C}} = \dot{J}\dot{\mathbf{S}} + J\ddot{\mathbf{S}}, \tag{5.14}$$

where

$$\dot{\mathbf{J}} = \frac{\partial \dot{\mathbf{C}}}{\partial \mathbf{S}}. \tag{5.15}$$

Combining the equations 5.8 and 5.14 we can derive the expression for the second derivative of a constraint function:

$$\ddot{\mathbf{C}} = \dot{J}\dot{\mathbf{S}} + JM^{-1}(\mathbf{F_{ext}} + \mathbf{F_c}). \tag{5.16}$$

Since we want the second derivative to be 0, we can set $\ddot{C}$ to 0 and rearrange the equation:

$$JM^{-1}\mathbf{F_c} = -\dot{J}\dot{\mathbf{S}} - JM^{-1}\mathbf{F_{ext}}. \tag{5.17}$$

The only unknown here is the constraint force vector F_c, which we are trying to determine. Since we know that our constraint force has to be parallel to the gradient, we can write it as a multiple of J:

$$\mathbf{F_c} = J^T\boldsymbol{\lambda} = \nabla C_1\lambda_1 + \nabla C_2\lambda_2 + \cdots + \nabla C_M\lambda_M. \tag{5.18}$$

Now, we can substitute F_c in the original equation 5.17 with the expression above:

$$JM^{-1}J^T\boldsymbol{\lambda} = -\dot{J}\dot{\mathbf{S}} - JM^{-1}\mathbf{F_{ext}}. \tag{5.19}$$

This is the system of linear equations with M unknowns. There are plenty of well-known methods which can be used to solve the system, such as: *Gauss Seidel*, *Gauss Jacobi* etc.

However, this approach is not practical and is too expensive for most applications. Since in our simulations, we will most likely have a lot of simulated objects, the matrices we have to work with will grow rapidly. Hence, the space and time complexity of this approach are not suited for real-time applications. One of the solutions to this problem is to only use pairwise constraints, which significantly simplifies the matrices and calculations. Another commonly used simplification is *impulse based approach*, which is discussed in the next section. An impulse is the integral of a force with respect to time and represents the change in momentum during that time:

$$\mathbf{P} = \int \mathbf{F}dt. \tag{5.20}$$

Since impulses are applied at the velocity level, we do not have to calculate the second derivative of a constraint function. Thus, our equations are much easier to work with.

5.2. Impulse-Based Approach

With *impulse based approach* we can directly use the equation 5.12 to calculate the final impulses we have to apply in order to satisfy the constraints [3]. This approach can be divided into 4 steps:

1. compute all external forces,
2. apply external forces and compute the resulting velocities,
3. calculate corrective impulses to satisfy constraints,
4. apply constraint impulses and calculate new velocities.

Since here we are acting at velocity level, we can write an approximation of acceleration as the ratio between the velocity change and delta time for the current time step:

$$\ddot{\mathbf{S}} \approx \frac{\dot{\mathbf{S}}(t + \Delta t) - \dot{\mathbf{S}}(t)}{\Delta t}. \tag{5.21}$$

If we again use the Newton's second law:

$$M\ddot{\mathbf{S}} = \mathbf{F_c} + \mathbf{F_{ext}}, \tag{5.22}$$

and substitute $\ddot{\mathbf{S}}$ with the expression 5.21, while $\mathbf{F_c} = J^T\boldsymbol{\lambda}$, we get the following equation:

$$M(\dot{\mathbf{S}}(t + \Delta t) - \dot{\mathbf{S}}(t)) = \Delta t(J^T\boldsymbol{\lambda} + \mathbf{F_{ext}}). \tag{5.23}$$

The only unknown is again vector $\boldsymbol{\lambda}$, which has M components. Since we know that the final velocity $\dot{\mathbf{S}}(t + \Delta t)$ has to be perpendicular to the gradient (we want $\dot{\mathbf{C}}$ to be 0), we can write $J\dot{\mathbf{S}}(t + \Delta t) = 0$. With that knowledge, we can rearrange the equation and get the expression for $\boldsymbol{\lambda}$:

$$JM^{-1}J^T\boldsymbol{\lambda} = -J(\frac{1}{\Delta t}\dot{\mathbf{S}} + M^{-1}\mathbf{F_{ext}}). \tag{5.24}$$

This is again the system of linear equations, but slightly different from what we got using the *force-based approach*. When dealing with equality constraints, the value of $\boldsymbol{\lambda}$ is not restricted, while for inequality constraints, we may want to clamp the final impulse to a certain range $\{\lambda^-, \lambda^+\}$. Technically speaking, we are solving a *Mixed Linear Complementary Problem* (MLCP). In this paper, the iterative approach is used to solve the MLCP, known as *Projected Gauss Seidel*.

5.2.1. Gauss Seidel

Gauss Seidel algorithm is used to solve a n-dimensional generic linear equation $Ax = b$. It is an iterative algorithm that initially guesses the result x_0, and afterward iterates through the rows, adjusting the elements of x corresponding to the diagonal element of A. The criteria that is used to stop the algorithm is:

- fixed number of iterations is reached,
- $||Ax - b||$ falls below a certain tolerance,
- Δx_i falls below a certain tolerance.

Algorithm 5 Gauss Seidel

```
1: procedure GAUSSSEIDEL(iterations)
2:     x ← x_0 ;
3:     for iter ← 1 to iterations do
4:         for i ← 1 to n do
5:             Δx_i ← (b_i − Σ_{j=1}^{n} A_ij x_j) / A_ii ;
6:             x_i ← x_i + Δx_i;
7:         end for
8:     end for
9: end procedure
```

5.2.2. Projected Gauss Seidel

Projected Gauss Seidel is an extension of a basic *Gauss Seidel* algorithm that handles bounds of the unknowns. In our case, there are bounds on λ for inequality constraints, while for equality constraints, those bounds can be simply set to $\{-\infty, \infty\}$ [1].
Now we can express the problem defined by equation 5.24 as an MLCP. We can substitute $M^{-1}J^T$ with matrix B and $\eta = -J(\frac{1}{\Delta t}\dot{\mathbf{S}} + M^{-1}\mathbf{F_{ext}})$, so our problem becomes:

$$\begin{aligned} \mathbf{w} &= JB\boldsymbol{\lambda} - \boldsymbol{\eta}, \\ \lambda^- \leq &\lambda_i \leq \lambda^+, \forall i, \\ w_i = 0 &\leftrightarrow \lambda^- \leq \lambda_i \leq \lambda^+, \forall i, \\ \lambda_i = \lambda^- &\leftrightarrow w_i \geq 0, \forall i, \\ \lambda_i = \lambda^+ &\leftrightarrow w_i \leq 0, \forall i. \end{aligned} \tag{5.25}$$

Since we are considering only pairwise constraints, matrix J can be significantly simplified. In general Jacobian matrix J is M $\times 6N$ for M constraints and N bodies.

By considering only pairwise constraints, each row will have at most two non-zero blocks of length 6 [1]. Thus, we can represent the matrix J with $M \times 12$ matrix J_{sp}:

$$J_{sp} = \begin{bmatrix} \mathbf{J_{11}} & \mathbf{J_{12}} \\ \mathbf{J_{21}} & \mathbf{J_{22}} \\ \vdots & \vdots \\ \mathbf{J_{M1}} & \mathbf{J_{M2}} \end{bmatrix}. \tag{5.26}$$

However, we still have to store the information about the bodies that each row of the matrix J represents, so an additional matrix J_{map} is required. It has a size of $M \times 2$ where each row stores two indices of the bodies represented by the corresponding row in the matrix J_{sp}:

$$J_{map} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ \vdots & \vdots \\ b_{M1} & b_{M2} \end{bmatrix}. \tag{5.27}$$

We can finally introduce the pseudo-code of the *Projected Gauss Seidel* algorithm using the matrices defined above.

Algorithm 6 Projected Gauss Seidel

procedure PROJECTEDGAUSSSEIDEL($iterations$)
  $\boldsymbol{\lambda} \leftarrow \boldsymbol{\lambda^0}$;
  $\mathbf{a} \leftarrow B\boldsymbol{\lambda}$;
  for $i \leftarrow 1$ **to** M **do**
    $d_i \leftarrow J_{sp}(i,1)B(1,i) + J_{sp}(i,2)B_{sp}(2,i)$;
  end for
  for $iter \leftarrow 1$ **to** $iterations$ **do**
    for $i \leftarrow 1$ **to** M **do**
      $b_1 \leftarrow J_{map}(i,1)$;
      $b_2 \leftarrow J_{map}(i,2)$;
      $\Delta\lambda_i \leftarrow \frac{\eta_i - J_{sp}(i,1)a(b_1) - J_{sp}(i,2)a(b_2)}{d_i}$;
      $\lambda_i^0 \leftarrow \lambda_i$;
      $\lambda_i^0 \leftarrow max(\lambda_i^-, min(\lambda_i^+, \lambda_i^0 + \Delta\lambda_i))$;
      $\Delta\lambda_i \leftarrow \lambda_i - \lambda_i^0$;
      $a(b_1) \leftarrow a(b_1) + \Delta\lambda_i B(1,i)$;
      $a(b_2) \leftarrow a(b_2) + \Delta\lambda_i B(2,i)$;
    end for
  end for
end procedure

The only parameter where the multiple constraints differ is *Jacobian matrix*, so that is the only thing we have to calculate per constraint. Now that we know how to solve the MLCP defined by constraints, we can discuss how to resolve collisions and apply friction.

5.3. Contact Constraint

Each colliding pair represents a new *contact constraint* in our system. It's function measures the object separation and can be written as:

$$C_n = (\mathbf{p_b} + \mathbf{r_b} - \mathbf{p_a} - \mathbf{r_a}) \cdot \mathbf{n}, \tag{5.28}$$

where vector $\mathbf{n}$ represents collision normal pointing from body A to body B [1]. In order to resolve the constraint we have to calculate the *Jacobian matrix* by differentiating C_n with respect to time:

$$\dot{C_n} = (\mathbf{v_b} + \boldsymbol{\omega_b} \times \mathbf{r_b} - \mathbf{v_a} - \boldsymbol{\omega_a} \times \mathbf{r_a}) \cdot \mathbf{n} + (\mathbf{p_b} + \mathbf{r_b} - \mathbf{p_a} - \mathbf{r_a}) \cdot \omega_a \times \mathbf{n}. \tag{5.29}$$

Since penetration is usually very small, the second term in the equation can be ignored. The *Jacobian matrix* can be determined by separating the velocities from the other terms:

$$J_n\mathbf{V} = \begin{bmatrix} -\mathbf{n}^T & -(\mathbf{r_a} \times \mathbf{n})^T & \mathbf{n} & (\mathbf{r_b} \times \mathbf{n})^T \end{bmatrix} \begin{bmatrix} \mathbf{v_a} \\ \boldsymbol{\omega}_a \\ \mathbf{v_b} \\ \boldsymbol{\omega}_b \end{bmatrix}. \tag{5.30}$$

This is an example of the inequality constraint, and the final impulse is bounded since we want to push the bodies apart but not pull them together [1]:

$$0 \leq \lambda_n < \infty. \tag{5.31}$$

5.3.1. Handling Penetration

Since we are using discrete collision detection, contact is not recognized until the bodies suddenly overlap. The differential equations derived above are usually not enough to prevent bodies from drifting into each other, so we need additional impulse which will move the bodies apart. It has to be proportional to the penetration depth and can be defined as [1]:

$$bias = -\beta C_n. \tag{5.32}$$

The scalar β is tunable and represents the speed of penetration resolution. If it is too small, it will not be able to resolve the penetration, while too large values can cause bounciness. We can add the bias parameter to our original equation for *impulse based approach* 5.24, so we end up with the final equation for resolving contact constraints:

$$JM^{-1}J^T\boldsymbol{\lambda} = -\frac{1}{\Delta t}\beta C_n - J(\frac{1}{\Delta t}\dot{\mathbf{S}} + M^{-1}\mathbf{F_{ext}}). \tag{5.33}$$

5.4. Optimizations

There are several optimization techniques commonly used by physics engines when dealing with constraints:

- warm-starting - used to speed up the convergence of *Projected Gauss Seidel* algorithm,
- body islands - divides bodies into smaller systems that will converge faster,
- sleeping - avoids useless work and calculations for bodies that are at rest.

5.4.1. Warm Starting

Since the bodies don't move much from step to step, we can use the information generated in the previous simulation step to speed up the convergence of the next step. The contacts and generated impulses are cached and reused as a starting point of the *Gauss Siedel algorithm* in the next simulation cycle. This is called *warm starting*, since we "warm started" the contact by using the previously calculated λ as initial guess for the next value of λ [1]. As a result, in majority of the cases the algorithm has to only adjust the initial λ, instead of starting from completely random value, which speeds up the convergence by a significant amount, and makes the simulation more stable.

5.4.2. Body Islands

During the simulation, one can observe that there may be distinct groups of bodies, where only the bodies belonging to the same group impact each other movement. The example can be seen in the figure 5.1. Since the blue platform represents the static body, we can clearly observe two groups of bodies coloured green and red. Those

Figure 5.1: Body Islands

groups can be obtained by running a *Depth first search* algorithm where edges are represented by contacts between bodies, while bodies itself represent the nodes of the graph. Once we divide the bodies into groups, it allows us to simulate each group separately, instead of running the simulation for the whole system at once. It improves the convergence and simplifies the calculations, reducing both space and time complexities.

5.4.3. Sleeping

By running the simulation for a longer period, we may end up in a situation where some islands are at rest, meaning the bodies move by a negligible distance at each step. In that scenario, we are wasting resources by calculating all the necessary parameters of the simulation for the bodies belonging to that island. In order to save time and resources, we can put the entire group to sleep when the linear and angular velocities of all the bodies in that group stay below a certain tolerance. When the island is at sleep, it is ignored during the simulation phase, and the bodies stay at the same location until they are awakened. Once the new contacts or forces are introduced that impact certain bodies inside the group, the whole group has to be awakened.

6. Results

As part of this paper, the algorithms and data structures discussed earlier are implemented in software using C++ programming language and OpenGL graphics programming interface. The library used for window creation is GLFW, while OpenGL context was created with the help of GLUT. In order to compare the implementation with some popular physics engines, *Bullet* and *PhysX* were integrated into the same graphics engine.

The GPU used for testing the implementation was 2070 super, while the CPU was i9-9900k. The resolution at which the application was rendered was 1920 x 1080.

Most common test for physics engines is box stacking. It consists of multiple boxes stacked right on top of each other. This test case introduces a lot of contacts between the bodies and each contact impacts the movement of all of the boxes in the stack, directly or indirectly. If the physics engine is stable and the parameters are chosen correctly, the structure should not fall apart, and the cubes should remain at the initial locations. The table below shows the parameters used for the simulation.

β	0.05
interations	10
gravity scale	1
mass	10 kg
delta time	0.016
dimensions	1m x 1m x 1m

The result can be seen on the figure below. The red points represent contact points in the simulation.

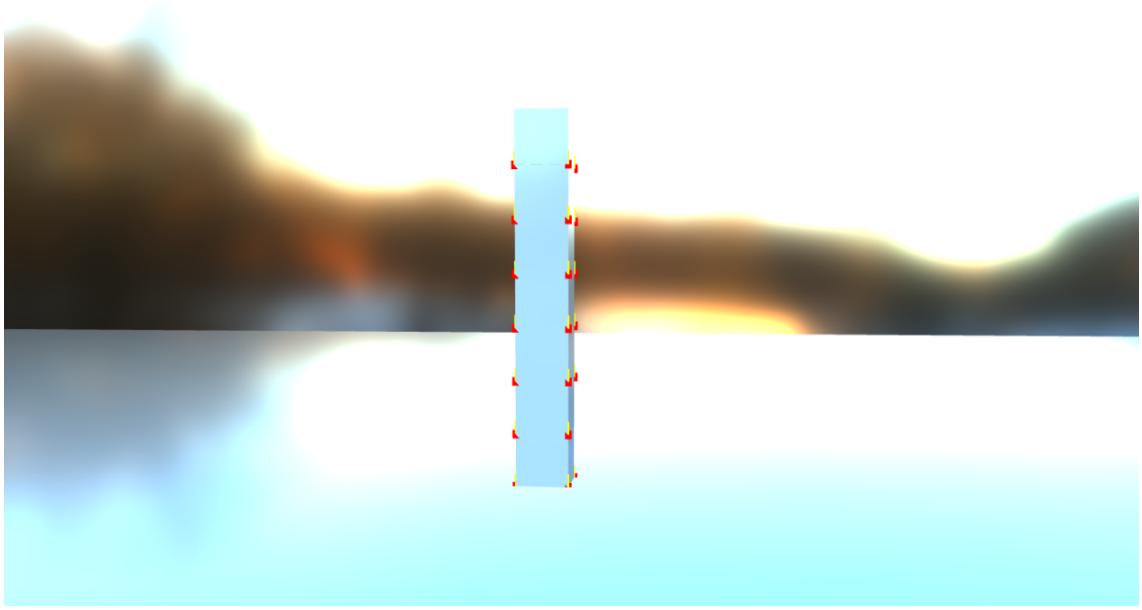

Figure 6.1: Box Stacking

The figure 6.2 shows the engine's performance in terms of frames per second, simulating a different number of bodies in the scene. The performance of the custom-made physics engine is roughly the same as compared to popular physics engines *Bullet* and *PhysX*.

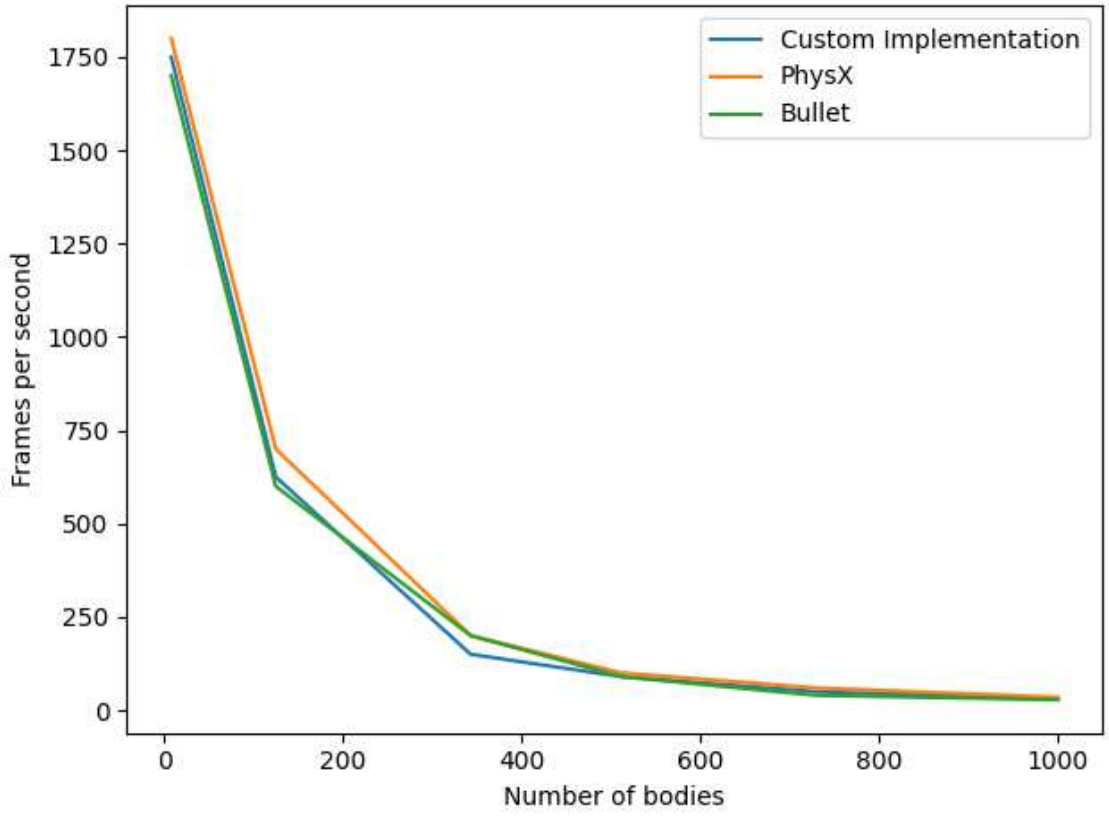

Figure 6.2: Performance Comparison

7. Conclusion

We have discussed how physics can be simulated within the game or any other application which requires realistic simulation of physical phenomena. This thesis is focused on rigid body motion, collision detection, and modeling the interaction between the bodies.

The physics engine is the core software component of all popular game engines. The quality of the simulation greatly impacts the player experience and immersion. Thanks to continuous development over the years, many clever simplifications and improvements have been made, especially regarding constrained dynamics, collision detection, and MLCP resolution. The techniques discussed in this paper are widely used in popular physics engines such as *Box2D*, *Bullet Physics*, *PhysX* etc. Other than games, physics simulations have a great impact on scientific research. One of the examples is computational fluid dynamics modeling, where particles are assigned force vectors combined to show circulation.

With the rising importance of physics simulations in many areas, new specialized hardware was developed to speed up the computations. *Physics processing unit* is a good example of hardware acceleration for physics engines. It is a dedicated microprocessor designed to handle the common calculations of physics. One of the more recent approaches to physics calculations is exploiting the GPU advantages in a highly parallelized environment. With the introduction of *Nvidia RTX* graphics cards, the broad-phase collision detection step can be completely offloaded to the GPU.

BIBLIOGRAPHY

[1] E. Catto. *Iterative Dynamics with Temporal Coherence*. Crystal Dynamics, 2005.

[2] B. U. Division of Engineering. *Dynamics and Vibrations*. https://www.brown.edu/Departments/Engineering/Courses/En4/notes_old/RigidKinematics/rigkin.htm#:~:text=A%20rigid%20body%20is%20an,of%20motions%20of%20the%20body.

[3] N. Souto. *Video Game Physics Tutorial, 2015*. https://www.toptal.com/game/video-game-physics-part-i-an-introduction-to-rigid-body-dynamics.

[4] Wikipedia, the free encyclopedia. *Kronecker Delta*, February 2022. https://en.wikipedia.org/wiki/Kronecker_delta.

[5] Wikipedia, the free encyclopedia. *Quaternion*, January 2022. https://en.wikipedia.org/wiki/Quaternion.

[6] Wikipedia, the free encyclopedia. *AVL Tree*, June 2022. https://en.wikipedia.org/wiki/AVL_tree.

[7] Wikipedia, the free encyclopedia. *Center of mass*, June 2022. https://en.wikipedia.org/wiki/Center_of_mass.

[8] Wikipedia, the free encyclopedia. *Jacobian matrix and determinant*, June 2022. https://en.wikipedia.org/wiki/Jacobian_matrix_and_determinant.

[9] Wikipedia, the free encyclopedia. *Moment of Inertia*, June 2022. https://en.wikipedia.org/wiki/Moment_of_inertia.

LIST OF FIGURES

List of Algorithms

3D Physics Engine

Abstract

This thesis explores all the fundamental concepts implemented in modern physics engines. It covers rigid body dynamics, collision detection and constraint solvers. Optimization techniques and special data structures necessary for real-time simulations are discussed. The algorithms and data structures presented are implemented in software using the C++ programming language and OpenGL graphics programming interface. The final results are presented and compared to popular physics engines in terms of stability and performance.

Keywords: physics engine, rigid body, collision detection, dynamic bounding volume tree, constraints, contraint solver, Gauss-Seidel algorithm.

Fizikalni pogon za 3D objekte

Sažetak

U sklopu ovog rada obrađeni su temeljni koncepti koji su sastavni dio modernih fizikalnih pogona. Objašnjeni su algoritmi vezani uz dinamiku krutog tijela, detekciju kolizije i rješavače ograničenja. Predstavljene su optimizacijske tehnike koje omogućavaju izvođenje fizikalnih simulacija u stvarnom vremenu. Obrađeni algoritmi i strukture podataka implementirani su koristeći programski jezik C++ i grafičko programsko sučelje OpenGL. Prezentirani su rezultati u pogledu stabilnosti i performansi, te je dana usporedba s drugim popularnim fizikalnim pogonima.

Ključne riječi: fizikalni pogon, kruto tijelo, detekcija kolizije, dinamičko stablo omeđujućih volumena, ograničenja, rješavač ograničenja, Gauss-Seidel algoritam.