A NOVEL BROAD-PHASE CONTINUOUS-TIME COLLISION DETECTION ALGORITHM

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

TARIK KAYA

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
GAME TECHNOLOGIES

JANUARY 2016

Approval of the thesis:

A NOVEL BROAD-PHASE CONTINUOUS-TIME COLLISION DETECTION ALGORITHM

submitted by **TARIK KAYA** in partial fulfillment of the requirements for the degree of **Master of Science in Game Technologies Department, Middle East Technical University** by,

Prof. Dr. Nazife Baykal _____
Director, **Graduate School of Informatics Institute, METU**

Assoc. Prof. Dr. Hüseyin Hacıhabiboğlu _____
Head of Department, **Modelling and Simulation, METU**

Assoc. Prof. Dr. Hüseyin Hacıhabiboğlu _____
Supervisor, **Modelling and Simulation, METU**

Examining Committee Members:

Prof. Dr. Veysi İşler
Computer Engineering Department, METU _____

Assoc. Prof. Dr. Hüseyin Hacıhabiboğlu
Modeling and Simulation, METU _____

Assoc. Prof. Dr. Ahmet Oğuz Akyüz
Computer Engineering Department, METU _____

Prof. Dr. Haşmet Gürçay
Computer Engineering Department, Hacettepe University _____

Assoc. Prof. Dr. Alptekin Temizel
Modeling and Simulation, METU _____

Date: **26 January 2016**

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: TARIK KAYA

Signature :

ABSTRACT

A NOVEL BROAD-PHASE CONTINUOUS-TIME COLLISION DETECTION ALGORITHM

Kaya, Tarık

M.S., Department of Game Technologies

Supervisor : Assoc. Prof. Dr. Hüseyin Hacıhabiboğlu

January 2016, 49 pages

Today's game development tools rely on realistic physics simulation more than ever. Physics simulation is a highly sophisticated subject, which can be approached from various angles, because of the impossibility of exact simulation. The impossibility of exact simulation for real world physics comes from the requirements of infinite precision, resolution and therefore infinite computational power. Hence the main aim of game physics simulations is making the game-world seem as physically realistic as feasible within the computational power limitations. One can always increment the physical precision and the number of active physical objects at the cost of performance and vice versa. This makes performance a critical issue. In our work we optimize the continuous-time collision detection algorithm by adding a novel broad-phase step to improve performance without any loss of realism, i.e., the output of our optimized method is exactly the same as the old method.

Keywords: Physics Simulation, Collision Detection, Continuous-time, Broad-phase, Optimization

ÖZ

YENİ BİR GENİŞ-AŞAMA SÜREKLİ-ZAMANLI ÇARPIŞMA TESPİT ALGORİTMASI

Kaya, Tarık

Yüksek Lisans, Oyun Teknolojileri Bölümü

Tez Yöneticisi : Doç. Dr. Hüseyin Hacıhabiboğlu

Ocak 2016 , 49 sayfa

Geçmişe kıyasla günümüzde oyun geliştirme araçları gerçekçi fizik simülasyonuna çok daha fazla önem vermektedir. Fizik simülasyonu, kavraması zor ve çok farklı açılardan yaklaşılabilen bir konudur, çünkü mükemmel fizik simülasyonu imkansızdır. Mükemmel fizik simülasyonun imkansızlığının nedeni, sonsuz çözünürlük, kesinlik ve dolayısıyla sonsuz bilgisayar gücü gereksinimidir. Bu yüzden oyun fiziği simülasyonunun amacı, bilgisayar gücünün mümkün kıldığı kadarıyla oyun dünyasını gerçekçi göstermeye çalışmaktır. Fizik simülasyonunun kesinliğini ve doğruluk oranını azaltarak daha hızlı bir simülasyon elde etmek ve tersi her zaman mümkündür. Bu yüzden simülasyon verimliliği büyük önem taşır. Biz işbu çalışmamızda sürekli-zaman çarpışma tespiti algoritmasına kendi geniş-aşama algoritmamızı ekleyerek fizik simülasyonunun verimliliğini arttırdık. Bunu algoritma çıktısını hiç değiştirmeden, yani gerçekçilikten ödün vermeden başardık.

Anahtar Kelimeler: Fizik Simülasyonu, Çarpışma Tespiti, Sürekli-zaman, Geniş-aşama, Optimizasyon

To my dear wife, Ayfer.

ACKNOWLEDGMENTS

I would like to express my gratitude to my supervisor, Assoc. Prof. Dr. Hüseyin Hacıhabiboğlu for invaluable guidance and support.

I would also like to thank Deniz Uğurca and Serkan Pekçetin for their incredible technical insights.

TABLE OF CONTENTS

ABSTRACT iv

ÖZ v

ACKNOWLEDGMENTS vii

TABLE OF CONTENTS viii

LIST OF TABLES xi

LIST OF FIGURES xii

LIST OF ABBREVIATIONS xiii

CHAPTERS

1 INTRODUCTION 1

1.1 Motivation 1

1.2 Contributions 3

1.3 Structure 4

2 BACKGROUND 5

2.1 General Physics Simulation 5

2.2 Game Physics 6

2.3 Categorization of Game Physics 6

2.3.1 Spatial Categories 6

2.3.1.1 Discrete Space 7

2.3.1.2 Continuous Space 7

2.3.2 Temporal Categories 8

2.3.2.1 Discrete Time 8

2.3.2.2 Continuous Time 9

2.4 Simulation Phases 10

2.4.1 Integration 10
2.4.1.1 Explicit Euler 11
2.4.1.2 Implicit Euler 11
2.4.1.3 Runge Kutta 12
2.4.1.4 Verlet 12
2.4.2 Collision Detection 12
2.4.3 Collision Resolution 13
2.4.3.1 Force Based 13
2.4.3.2 Impulse Based 14
2.5 Previous Work 14
2.5.1 Priority Queue Based Event Simulation 14
3 COLLISION DETECTION 17
3.1 Narrow Phase 17
3.1.1 Separating Axis Theorem 18
3.1.2 Gilbert–Johnson–Keerthi 18
3.1.3 Voronoi Clipping 19
3.1.4 Static Circle Pair Collision 19
3.1.5 Dynamic Circle Pair Collision 20
3.2 Broad Phase 21
3.2.1 Bounding Volumes 22
3.2.1.1 Axis Aligned Bounding Box 23
3.2.1.2 Oriented Bounding Box 24
3.2.1.3 Discrete Orientation Polytopes . . . 24
3.2.1.4 Spheres 25
3.2.1.5 Sphere Swept Volumes 25
3.2.1.6 Inner Sphere Trees 25
3.2.2 Spatial Subdivision 26
3.2.3 Bounding Volume Hierarchies 26
3.2.4 Sort and Sweep 29
3.3 Comparison of Discrete and Continuous Time 29

4 PROPOSED METHOD 33
4.1 Broad-Phase for Continuous-Time Collision Detection . 33
4.2 Axis Aligned Bounding Boxes 34
4.3 Bounding Volume Hierarchy 34
4.4 Adaptive Time Step 35
4.5 Top-down Divide by Median Tree Construction 36
4.6 Partitioning an array 36
4.7 Pseudocode 36
5 RESULTS AND DISCUSSION 39
5.1 Complexity Analysis 39
5.2 Running Times 40
5.3 Comparison with Discrete-Time 42
6 CONCLUSION AND FUTURE WORK 45
REFERENCES 47
APPENDICES

LIST OF TABLES

Table 5.1 Running times in seconds (average of 100 runs) 41
Table 5.2 Comparison with discrete-time 43

LIST OF FIGURES

Figure 1.1 Tetris uses a discrete space simulation 2
Figure 1.2 Lunar Lander uses a continuous space simulation 3

Figure 3.1 Different types of bounding volumes (blue), applied on the same object (green) in order: AABB, OBB, DOP, Sphere, Sphere-Swept, Inner Sphere Tree 23
Figure 3.2 A simple AABB-hierarchy. The objects are black. The AABB levels from top to bottom are: red, green, orange, blue. 27

Figure 5.1 Running time measurements compared to sample functions in scaled log-log plot, where x axis represents number of objects and y axis represents running times 42
Figure 5.2 A screen-shot from our visualization tool 44
Figure 5.3 A screen-shot from our visualization tool displaying some AABBs 44

LIST OF ABBREVIATIONS

CPU	Central Processing Unit
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
SIMD	Single Instruction Multiple Data
V-Clip	Voronoi Clipping
SAT	Separating Axis Theorem
GJK	Gilbert–Johnson–Keerthi
AABB	Axis Aligned Bounding Box
DOP	Discrete Orientation Polytopes
SDL	Simple DirectMedia Layer
LoD	Level of Detail

CHAPTER 1

INTRODUCTION

Almost all digital games use some form of physical simulation to offer its players a more relatable virtual world. The game of "Lunar Lander" (See Figure: 1.2) is an example for realistic and continuous physical simulation. However, the physical simulation does not have to be realistic. It can even be abstract and discrete such as in the game of "Tetris" (See Figure: 1.1). Although the complexities and the requirements of these different physics simulations can vary significantly, they share a common feature: they are one of the most computationally expensive parts of a game engine. Hence the optimization of the physical simulation becomes essential.

One of the most time consuming parts of a physical simulation is the collision detection step, where we detect all of the colliding pairs of objects. In this work, collision detection optimization is the main focus.

Our optimization is based on a certain approach to physics simulation, namely; continuous-time collision detection, where the simulator works with events, not time steps, as opposed to discrete-time collision detection. For discrete-time collision detection there are many known and practically applied optimization techniques, known as broad-phase algorithms. However, little research has been done for the broad-phase of continuous-time collision detection. This work aims to fill this gap.

To summarize; we add a broad-phase optimization step to continuous-time collision detection in order to achieve sub-quadratic time complexity, which has been $O(n^2)$ before.

1.1 Motivation

Graphics rendering and physics simulation technologies have a lot in common. In graphics rendering, we would like to be able to render as many triangles as possible to improve the smoothness and the overall visual quality of graphics. Although we have known how to render triangles for a long time, we are still in the progress of optimizing the rendering pipeline. That is why a tremendous amount of work has been dedicated to graphics research, such as the invention of special hardware called GPU's. The situation in physics

Figure 1.1: Tetris uses a discrete space simulation

simulation is very similar, in the sense that we are very much interested in increasing the amount and the level of detail of our simulated physical objects in our games. One example would be: the number of particle effects in a game is rarely bounded by the game design, but instead it is bounded by the underlying hardware and software capabilities.

In game physics simulations there is always a trade-off between accuracy and performance. In order to improve the performance of a physical simulation, one can always decrease the amount of detail in physical objects, increase the length of the time step used, or use numerical systems with less precision. But all of these methods have their disadvantages. The effects of these methods range from less immersive virtual environments to game crashing bugs, making the whole system unplayable.

On the other hand, some optimization techniques do not depend on trade-offs. Some optimization techniques give us exactly the same output as the non-optimized techniques with significantly reduced performance requirements. In this work, we will present such an algorithm.

Although the discrete-time collision detection methods are more widely used and investigated, this work explains why continuous-time collision detection is an important subject, that is strictly needed in some situations such as the billiard games.

It is our sincere hope that through the presented optimization process, the continuous-time collision detection will be a more feasible option for game physics simulations.

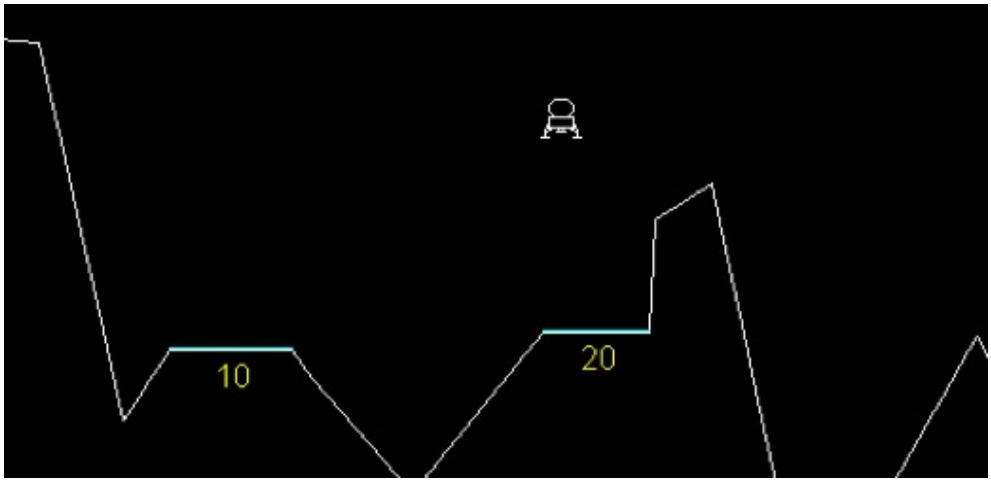

Figure 1.2: Lunar Lander uses a continuous space simulation

1.2 Contributions

The main contribution of this work is the optimization of the continuous-time collision detection algorithm by the addition of our novel broad-phase step to eliminate most of the potential collision checks, which leads to a significant performance increase.

We have explained the significance of continuous-time collision detection along with its disadvantages. (See section: 3.3)

To demonstrate the work that has been done, not only the collision detection algorithm has been implemented, but the whole physical simulation pipeline (See section: 2.4) has been, namely:

1. **Spatial integration**: We implemented the simplest method of spatial integration, which is Explicit Euler Integration. (See section: 2.4.1.1)
2. **Collision detection**: We implemented our own novel collision detection method, which will be explained in detail. (See section: 4)
3. **Collision resolution**: We implemented the simple impulse based elastic collision resolution for this step. (See section: 2.4.3.2)

For visualization purposes a simple software-renderer has also been implemented. It utilizes the SDL2.0 (Simple DirectMedia Layer) with C++. [1] The visualization software has the capabilities to render all physical objects, to change speed of time and to draw AABB's of our novel broad-phase algorithm.

The physical simulation software and the visualization software are kept in separate projects. They communicate via files. The physical simulation software simulates the given physical environment and then writes the results to a file. The visualization software reads that file and renders the results. This

separation is a desired property in order to keep the running time measurements accurate.

A complexity analysis of the proposed simulation is also given. Although the worst case time complexity of our method is still quadratic, the average time complexity is sub-quadratic. This is illustrated by the practical time measurements of simple quadratic method and our novel method.

1.3 Structure

In this chapter the motivation and the contribution of our work are presented.

In chapter 2 background for physics simulation is given. Starting with general physics simulation and its applications, the chapter focuses on game physics simulation. Game physics simulations are categorized according to the representations of time and space. Two out of the three phases of physics simulation are explained in detail, leaving out collision detection, which has its dedicated chapter 3. Previous work on the topic of continuous-time collision detection is also given.

Chapter 3 is dedicated to the second phase of the physical simulation, which is collision detection, because collision detection is the major topic of this work. The collision detection algorithms are divided into two categories:

1. **Narrow Phase** algorithms detect collisions between pairs of physical objects.
2. **Broad Phase** algorithms optimize the collision detection between more than two physical objects. This is the topic of our contribution with this work.

Chapter 3 also includes a comparison between discrete-time and continuous-time collision detection methods.

Chapter 4 explains the details of our novel method, which is a broad-phase algorithm for continuous-time collision detection. This chapter also includes the entire pseudocode of our novel method.

Chapter 5 gives a complexity analysis of the proposed method along with practical running time measurements.

The conclusion and possible future work can be found at chapter 6.

CHAPTER 2

BACKGROUND

In this chapter, background for physics simulation is given. The topic of collision detection is left for the next chapter.

2.1 General Physics Simulation

Although the focus of this work is on game physics, the world of physics simulation is vastly broader than digital games. Physics simulation has found uses in a diverse set of areas from robotics to cosmology. Here are a few examples:

- **Robotics:** Lin describes a novel collision detection algorithm which is useful for robotics. The narrow-phase collision detection between convex polyhedra is optimized via their novel incremental method, thus allowing robots to plan their navigation at real time. [2]
- **Flow simulation:** Physics simulation can be used to predict incompressible fluid flows, for example by using vortex methods. Ploumhans et al. use the vortex sheet algorithm to achieve no-slip boundary condition in order to simulate accurate redistribution of flow particles. [3]
- **Cosmological simulation:** Some researchers try to explain the universe through physical simulation. Viel and Haehnelt present the results of a highly sophisticated cosmological simulation. [4]
- **Civil engineering:** Especially the finite element method proves to be very useful for civil engineering. Finite element method is the state of art method for simulating multidimensional physical setups. It is a result of advanced linear algebra and differential calculus combined. It can also be thought of as a generalization of the simple finite difference methods. [5]
- There are many more examples for the usage of physical simulation. [6] [7] [8]
- **Game physics** is the main focus of this work, which is just a subsection of general physics simulation topic.

2.2 Game Physics

Different areas of physics simulation have different requirements. In civil engineering, simulating a bridge construction requires extremely high numerical accuracy and precision to be considered useful. In flow simulation, higher accuracy can be achieved by increasing the amount of graph vertices or number of particles based on the chosen method. Hence they need to be as efficient as possible to overcome this computational burden.

When it comes to game physics, the main goal is to make virtual worlds look realistic or at least coherent in its own rules. Even when the aim is to achieve a certain level of realism, the simulation itself does not need to be actually realistic at all, as long as it is successful at making the players think that it is. This is another property that physics simulations share with graphics rendering, as well as the need for optimization.

For example, in the early Tomb Raider games, the main character was physically simulated as a simple capsule, which was common at early 3D games, because of hardware and software performance limitations. This method has been successful at making players perceive realism, even though it was only a rough approximation.[9]

Another example is LoD (Level of Detail) technique used in NVidia Apex [10]. If a cloth mesh is far away from the players' perspective, it is simulated less accurately, because it does not substantially contribute to the perception of realism.

2.3 Categorization of Game Physics

As explained earlier, the aims of general physics simulation and game physics simulation are different. General physics simulation aims for the realism, whereas game physics simulation merely needs to be consistent and coherent. This leads to freedom in the types of physical simulations one can perform on game engines.

We can categorize game physics in different ways. Some of the important aspects for our work are presented here:

1. **Spatial**; refers to how the space of physics simulation is represented.
2. **Temporal**; refers to how the time of physics simulation is represented.

2.3.1 Spatial Categories

Different game mechanics simulate space in different ways. We will categorize space in games into two:

1. **Discrete** space means that there are fixed possible positions and objects can not be in-between them.
2. **Continuous** space means the absence of fixed possible positions and objects can be at any position.

2.3.1.1 Discrete Space

Some games simulate space discretely. The best known example for this is the game of Tetris, where players try to place blocks in such a way that they form full rows. The blocks move in a grid discretely. They are never half way between grid cells. Instead they appear on their next position instantaneously.

There are many different games that share this property. The common point of these games is that they utilize a grid of cells. It can be a rectangular grid such as in the case of Tetris, or it can be a hexagonal grid such as in the case of many strategic games like Civilization. There can even be three dimensional grids.

These types of games generally discretize the orientation (rotation) of objects as well as their positions.

Since discrete space in games presents problems with trivial solutions in general, our focus will not be on this type of physics.

2.3.1.2 Continuous Space

Compared to discrete space, continuous space simulation in games are more realistic, because real life physics work continuously. In these kinds of simulations, objects do not disappear and then reappear at their next location. Even if this is the case, objects do these jumps in such small time-steps and relatively small space-steps, that they seem to be moving continuously to players.

Although it is a theoretical discussion whether the real life physics work in an extremely small partitioned discrete way or in an absolute continuous way[11], for the context of game physics we will consider seemingly very small spatial steps to be continuous, because the way how current computers work limits us to do so.

There is an increasing trend towards continuous physics, partly due to their recent availability. Typically they require floating point arithmetic and more computational capabilities. It was nearly impossible to simulate continuous space in games in 1970. Nowadays it is very common, even for simple games with low budget.

An example for this type of physical simulation is classical platformer games, such as Super Mario Brothers.

Our work focuses on continuous space simulation.

2.3.2 Temporal Categories

Game physics simulations also differ on the basis of how they handle time. There are similarly two main categories here:

1. **Discrete** time refers to fixed increments in time, regardless of how many events the time increment contains.
2. **Continuous** time refers to adjusting increments in time, which change according to the physical events in simulation.

A simple approach is discrete time steps, where we integrate object velocities altogether and then check for pairwise collisions. This means that there will be overlapping (not only touching) objects, which require special care. When applicable, this approach is more common due to its simplicity and low performance requirement.

A different approach would be to simulate time continuously. Although we know from computation theory[12] that our current computers work exclusively discretely, it is possible to simulate continuous time if the "events" of our simulation occur finitely and discretely. Theoretically this approach is much more realistic and exact compared to discrete time solutions. However one should note that this method may not be applicable under all circumstances.

Our work is focused on continuous-time simulation.

2.3.2.1 Discrete Time

Discrete time simulation means that we simulate certain time lengths as steps, one after another. In each step, we first move our objects regardless of any collisions. Then the collision checking and resolution are performed.

This implies that we simulate the physics in time steps, regardless of how many events occur inside a time step. These events may be collisions, force changes, impulses and many others. Even if these events that occur in a single time step are inter-dependent or overlapping we treat them as independent in these kinds of simulations. Note that it is also possible to miss some events in this scheme.

Another important problem with the discrete time simulation is that objects do not actually come into touching positions, but instead they overlap. In order to look realistic, we need to prevent objects from overlapping. To handle this we apply artificial separating procedures. These procedures vary in complexity. The simplest and widely used approach would be to move them away from each other in the axis that contains the centers of colliding pair of objects. This would be a rough approximation. On the complex and computationally expensive side, we can try to approximate the time of collision and

rewind the simulation back to the colliding time, so that we can achieve more exact positions and velocities during the collision.

After this separation step, we can resolve collisions by setting new velocities to the pairs of colliding objects, by applying either impulses or forces.

One way to overcome these possible short-comings in physics simulation is to use as small time-steps as computationally feasible. Halving the time step means we need to execute twice as many simulations than a full time step. This leads to a trade off between accuracy and performance.

One argument for the discrete time simulation for games is that the human eye is not very good at spotting the physical artifacts in crowded scenes. As long as objects do not overlap and do bounce off each other, the precision of these events generally do not play a huge factor in perception of realism.

The main reasons for the usage of discrete time simulation is ease of implementation and performance.

2.3.2.2 Continuous Time

If real world physics work in discrete time steps, it seems to work with infinitely small time steps, which is not computationally possible with our current hardware. Despite this fact; if we need more accurate physics simulation than discrete time simulation, we have another choice, which is continuous time physics simulation.

If we try to make discrete time simulation more accurate, we will need to make the time step shorter and shorter. This operation will result in many time steps, in which we do not compute any useful information, because there will not be many events in each time step. This means that if we had kept the time step long, we would get the same output, because of the lack of events in the small time steps. How can we differentiate between useful and useless time steps? We can achieve this by using continuous time simulation.

The main idea of this concept is; instead of simulating in discrete time steps, we can work in discrete events. We can always keep track of the next event that will occur and set the time step to simulate up to the time of the next event. After we simulate the next event, we can calculate the next nearest event.

As promising as this method may sound, there are some issues to take into consideration. The most notable one is that the computation of the next event can be computationally very expensive or even impossible. Special care must be given to resting contacts, such as a book resting on a table, under gravitational forces. Another one is that simulating discrete time steps means that we move objects as if none of them collide while moving and then calculating the overlapping pairs can be achieved by algorithms of sub-quadratic complexity. But calculating the next event is in its nature a quadratic problem, because

we need to take each pair of objects into consideration. Our contribution with this work is exactly about this issue. We have decreased the computational complexity of the calculation of the next event.

One should note that in circumstances where missing some collisions can not be tolerated, continuous-time physics simulation is a necessity.

2.4 Simulation Phases

Regardless of the different categories of physics simulations, they share some common characteristics. One of them is that they work in phases. common phases of physics simulations can be enumerated as follows:

1. Integration
2. Collision Detection
3. Collision Resolution

Each of them in detail shall be examined.

2.4.1 Integration

Establishing some terminology is necessary for explaining the integration step.

We define the *position* of an object as the coordinates of the center of mass of an object. We regard the position as a vector. We will denote position with $\vec{p}$.

We define the *velocity* of an object as the change of position with respect to time. Therefore the velocity is also a vector. We shall use linear velocity and velocity interchangeably. We shall make clear when the subject is angular velocity instead of linear velocity. We will denote velocity with $\vec{v}$.

We define the *acceleration* of an object as the change of velocity with respect to time. Therefore acceleration is also a vector. We shall treat the angular acceleration separately just as in the case of velocity. We will denote acceleration with $\vec{a}$.

One should note that the velocity is the first order derivative of position and the acceleration is the second order derivative of position with respect to time.

As in most studies on physics simulation we will not be focusing on the third order derivative of position, which is called *jerk*.

To summarize:

$$\vec{a} = \frac{d\vec{v}}{dt} = \frac{d^2\vec{p}}{dt^2} \tag{2.1}$$

2.4.1.1 Explicit Euler

By Newton's laws of motion we can calculate the position of an object at any time, assuming it has constant acceleration: [9]

$$\vec{p}(t+dt) = \vec{p}(t) + \vec{v}(t)dt + \frac{1}{2}\vec{a}(t)dt^2 \tag{2.2}$$

Assuming constant acceleration means that the objects are under only constant forces such as gravity, do not collide and are not effected by certain models of friction and drag.

These assumptions are true only for trivial physical simulations. Therefore we need to use small time steps in order to reduce error of this false assumption.

Using a small time step means that the term dt^2 becomes negligible. So instead we can use this heuristic in practice:

$$\vec{p}(t+dt) = \vec{p}(t) + \vec{v}(t)dt \tag{2.3}$$

Similarly, we can calculate the new velocity according to:

$$\vec{v}(t+dt) = \vec{v}(t) + \vec{a}(t)dt \tag{2.4}$$

For simulation, this means updating positions and velocities of objects according to velocities and accelerations they had in a previous time. This is the simplest and computationally least expensive method for integration and it is used very widely for game physics.

Our work uses this approach for integration.

2.4.1.2 Implicit Euler

The assumption of constant acceleration made by explicit Euler method does not hold in some cases, such as in spring systems. Although one can always increase accuracy by shortening the time step, there is an alternative method called *implicit Euler method* [13]

Explicit Euler uses the current velocity and acceleration to predict the next po-

sition and velocity, whereas implicit Euler uses the next velocity to predict the next position and the next acceleration to predict the next velocity. Namely:

$$\vec{p}(t+dt) = \vec{p}(t) + \vec{v}(t+dt)dt \tag{2.5}$$

$$\vec{v}(t+dt) = \vec{v}(t) + \vec{a}(t+dt)dt \tag{2.6}$$

One shall notice that the unknowns in these equations are dependent on the other unknowns, since acceleration may be dependent on position, such as in spring systems. This makes the solution of this set of equations a lot more complex than explicit Euler. Implicit Euler system is non-linear, which requires computationally expensive methods such as "Newton-Raphson Solver" which are out of scope for our work. [14]

2.4.1.3 Runge Kutta

Another approach of integration is dividing the integration into sub-steps without completely decreasing the time step altogether. Runge Kutta is an integration method where we can adjust the accuracy according to our precision requirements and performance limits. The number of sub-steps determine the order of accuracy. [14]

2.4.1.4 Verlet

Another approach for integration is keeping track of previous positions instead of velocities. This leads to verlet integration: [15]

$$\vec{p}(t+dt) = \vec{p}(t) + (\vec{p}(t) - \vec{p}(t-dt)) \tag{2.7}$$

Which can be simplified as:

$$\vec{p}(t+dt) = 2\vec{p}(t) - \vec{p}(t-dt) \tag{2.8}$$

Verlet integration can be proven to be of fourth order accuracy. [14]

2.4.2 Collision Detection

In physics simulation, we need to find all pairs of colliding objects in order to resolve the collisions. This phase is called collision detection. Collision detection is computationally the most expensive phase in most cases. That is why our work is built upon optimizing this phase.

Since our work is concentrated in this subject, the next chapter will go into details of this phase.

Right now it is worth mentioning that the collision detection is performed after integration phase and before collision resolution phase.

2.4.3 Collision Resolution

The final step of physics simulation is the resolution of collisions. The simulation detects the collisions in the previous step, in order to resolve them in this step.

In real life a collision is not an instantaneous event. Real physical objects are not rigid, despite the general assumption of physical simulations. Even the most rigid objects in real life have some elasticity. Hence their behavior are spring-like near the surface. So a collision in real life takes some time to resolve, with opposite forces applied on pairs of colliding objects. [9]

On the other hand this time of collision resolution is so short most of the time, that we can simulate it as instantaneous impulses.

This topic leads to the major categorization of collision resolution. Some physics simulations try to simulate real life more realistically, using only forces to resolve collisions instead of impulses. They are called "Force Based Simulations". This is a complicated topic and unrelated to our work on collision detection. Therefore we will not go into details.

Another simplified approach is to treat collisions as instantaneous impulses, that separate colliding objects. This is the approach we will follow mainly due to its simplicity.

It is worth noting that different elasticity values are easy to simulate in both approaches.

2.4.3.1 Force Based

Although more complicated and computationally more costly, these types of simulations can achieve the realism that impulse based approaches are not able to, due to their fundamental assumptions.

Perhaps the best known example for this approach is the "Open Dynamics Engine", which is mainly used for realistic engineering simulations, but also in some games. [16]

2.4.3.2 Impulse Based

Due to its simplicity and computational performance, this is the preferred model of collision resolution in many game physics simulations. Examples include Box2D [17] which is the most widely used two dimensional game physics engine by far.

Since our work is not about collision resolution but about collision detection, the details of this topic will be left out. It suffices to say that we use this approach for our physics simulation at this work, due to its simplicity.

2.5 Previous Work

It is our observation that most of the work on game physics simulation comes from the game industry, not from the academia. It is also a known fact that the majority of the industry uses discrete time collision detection instead of continuous time collision detection. This makes the research on continuous time collision detection very sparse. However, most popular game physics engines are open source. [17] [18] [16] This enables us to explore the commonly used techniques, such as discrete time collision detection and methods for overcoming tunneling.

Although there has not been much research on our specific subject, which is continuous time broad phase collision detection, there has been much research on discrete time broad phase collision detection, which is explained in detail. (See section: 3.2) Our aim is to bridge the gap with this work.

2.5.1 Priority Queue Based Event Simulation

Sedgewick [19] uses a physics simulation similar to our simulation in order to demonstrate the priority queues. In Sedgewick's work all of the upcoming collisions are calculated and put into a priority queue. Their priority is inversely proportional to their collision time. These events are processed one by one. It is important to note that a collision may result in other collisions and may invalidate some others. Therefore we need to validate the collisions before processing them.

This is a valid approach for continuous time physics simulation. However the complexity analysis of this approach shows a $O(n^2)$ memory complexity, which is what we have been trying to avoid in our work.

A simple observation is enough to prove that this algorithm has quadratic memory complexity. At any point of time in the simulation the objects may be moving towards a common point. This leads to $O(n^2)$ number of upcoming events, since every pair of objects will collide.

Sedgewick's approach is also not numerically stable. Every event time is cal-

culated much before occurring, compared to our work. This leads to numerical instabilities in floating point arithmetic because the magnitudes of represented numbers are much bigger.

The sole purpose of our work is to avoid quadratic complexity. Therefore we see our approach as an improvement over this method.

CHAPTER 3

COLLISION DETECTION

The contribution of this work is about collision detection. As previously described, collision detection is one of the three phases of physical simulation. It is worth noting that collision detection is almost always the most performance critical phase of the three phases, because naive approaches to integration and collision resolution lead to linear time algorithms, whereas a naive collision detection implementation has quadratic time complexity. This makes the optimization of collision detection essential. Hence a huge body of work has been done to achieve this goal. [20]

Because of the popularity of discrete time collision detection in practice, all of the methods described below have been aimed towards discrete time collision detection. The aim of this work is to make a working set of these collision detection methods available for continuous time collision detection.

The task of collision detection is categorized into two phases:

1. **Narrow Phase** algorithms detect collisions between two physical objects.
2. **Broad Phase** optimize the collision detection between more than two bodies.

3.1 Narrow Phase

Narrow phase of collision detection refers to the detection of collision between a pair of objects. The objects in hand can be arbitrarily complex.

All of the methods described below work for convex objects in any dimensions. The reason for that is collision detection for convex objects leads to much more optimized approaches than general objects which can also be concave. Also, any concave object can be divided into convex objects and then tested against each other separately. In order to avoid quadratic complexity in this case, broad-phase collision detection methods are used, which will be explained later. There are also some special methods aimed towards non-convex bodies. [21]

Since the methods that will be described below have been aimed towards discrete time collision detection, they work only for static objects, in the meaning that they do not take velocities into account. They take objects as snapshots in time, with their corresponding positions and rotations.

3.1.1 Separating Axis Theorem

The main idea of separating axis theorem is simple: If the one dimensional projections of two bodies do not overlap, they cannot possibly overlap in their native dimensions. The objects are usually two or three dimensional. Separating Axis Theorem is a popular method in practice, because it is relatively easy to make intuitive sense. However, the theoretical exposition which leads to SAT is not further discussed. The interested reader is referred to [20] for theoretical details.

In two dimensions, this method has linear time complexity with respect to total vertex count of possibly colliding objects. This comes from the proven fact, that we only need to check for edge normals as separating axes.[20]

For three dimensional objects, one needs to check for both surface normals and cross products of edge directions as separating axes. By taking Euler's formula into account, it can be shown that this leads to a quadratic time complexity with respect to number of vertices of both objects. [22]

Euler's formula(3.1) states that number of edges is linearly correlated to the total number of vertices and faces.

$$\#edge = \#face + \#vertex - 2 \tag{3.1}$$

Despite its quadratic time complexity SAT is a useful method for three dimensional objects with low number of vertices, such as rectangular prisms, which are commonly used [23]

This work does not make use of SAT.

3.1.2 Gilbert–Johnson–Keerthi

In order to bring down the time complexity of SAT, one can utilize the idea of Minkovski Sums.[24]

The main idea is: Two convex objects collide if and only if the Minkowski sum of the origin-reflected object with the possibly colliding object contains the origin. This can also be named as a Minkowski difference.

The naive implementation of this idea leads to a yet another algorithm with quadratic time complexity, which runs slower than SAT in practice. But with some clever optimization, GJK is brought down to sub-quadratic complexity.

[25]

This work does not make use of GJK.

3.1.3 Voronoi Clipping

In almost every physical simulation, one can notice temporal coherence. Temporal coherence refers to objects not changing their positions instantaneously and arbitrarily, similar to a teleportation. Instead they change their positions smoothly, given realistic velocities.

One can utilize this fact for optimizing collision detection. A simple application of this relates to SAT method. One can store the separating axis for a pair of objects from previous calculations. There is a high probability that it is still a separating axis. Storing those axes at each calculation can lead to great gain in efficiency, with increased need of memory.

This idea can be further developed into incremental approaches for collision detection, which rely highly on temporal coherence. One of the most developed algorithms in this fashion utilizes the Voronoi division of space and is called V-Clipping [26]

Although this method has theoretically high complexity, it runs fast in practice due to the temporal coherence.

This work does not make use of V-Clip.

3.1.4 Static Circle Pair Collision

Compared to general convex body collision detection algorithms described thus far, we now describe a simpler specific algorithm for detecting collisions between static circles, meaning we treat them as stationary objects, with no velocity.

Two static circles collide if the distance between them are less than or equal to the total of their radii.

Circles i and j collide if and only if

$$||\vec{p_i} - \vec{p_j}|| <= (r_i + r_j) \tag{3.2}$$

where $\vec{p_i}$ stands for the position of the center of circle i and r_i stands for the radius of circle i. The same notation applies to circle j. The notation $||\vec{v}||$ stands for the scalar length of a vector. Throughout this work, multiplication of two vectors refer to the inner product of vectors.

Since calculating the length of a vector involves a square root and square roots are costly to compute, this method can be optimized by squaring both sides

of the inequality. One should note that this is possible because both sides are always non-negative. The resulting equation is more performant in todays hardware:

$$(\vec{p_i} - \vec{p_j})^2 <= (r_i + r_j)^2 \tag{3.3}$$

where squaring a vector means calculating the dot product with itself.

In summary, this equation leads to a simple and efficient method for testing whether two static circles collide.

3.1.5 Dynamic Circle Pair Collision

Since the focus of this work is on continuous time collision detection, the bodies used in collision detection are not static but dynamic. It means that velocities must be taken into consideration for collision detection algorithms.

We need a collision detection algorithm which determines the time of collision, instead of whether the circles are currently colliding.

We know that the position of dynamic circle is a function of time. Here we will assume that the linear velocity is constant, not a function of time. Therefore we can safely use Explicit Euler Method(2.2). Constant velocity leads to this simplified formula:

$$\vec{p}(t + dt) = \vec{p}(t) + \vec{v}dt \tag{3.4}$$

Assuming we know the value $\vec{p}(0)$ we get the formula:

$$\vec{p}(t) = \vec{p}(0) + \vec{v}t \tag{3.5}$$

This is a first order equation with time variable.

We want to know the exact time of collision, which is the time where the distance between the circles is equal to the total of their radii. Replacing the position values of static circle collision formula(3.3) with these new position functions we get:

$$(\vec{p_i}(t) - \vec{p_j}(t))^2 = (r_i + r_j)^2 \tag{3.6}$$

Note that squaring a vector means taking the inner product with itself.

We need to solve this equation for time:

$$((\vec{p_i}(0) + \vec{v_i}t) - (\vec{p_j}(0) + \vec{v_j}t))^2 = (r_i + r_j)^2 \tag{3.7}$$

$$((\vec{v_i} - \vec{v_j})t + \vec{p_i}(0) - \vec{p_j}(0))^2 = (r_i + r_j)^2 \tag{3.8}$$

$$(\vec{v_i} - \vec{v_j})^2 t^2 + 2((\vec{v_i} - \vec{v_j}) \cdot (\vec{p_i}(0) - \vec{p_j}(0)))t + (\vec{p_i}(0) - \vec{p_j}(0))^2 - (r_i + r_j)^2 = 0 \tag{3.9}$$

This formula may seem complicated but it is actually a simple quadratic formula, that can be solved efficiently, using only a square root function call and basic arithmetic.

$$at^2 + bt + c = 0 \tag{3.10}$$

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{3.11}$$

This is the end result:

$$t = \frac{-\vec{v_{ij}} \cdot \vec{p_{ij0}} \pm \sqrt{(\vec{v_{ij}} \cdot \vec{p_{ij0}})^2 - \vec{v_{ij}}^2(\vec{p_{ij0}}^2 - (r_i + r_j)^2)}}{\vec{v_{ij}}^2} \tag{3.12}$$

Where:

$$\vec{v_{ij}} = \vec{v_i} - \vec{v_j} \tag{3.13}$$

$$\vec{p_{ij0}} = \vec{p_i}(0) - \vec{p_j}(0) \tag{3.14}$$

As can be understood from $\pm$ sign, this formula gives two possible solutions. As it is common in algebraic solutions, this formula also accepts possible negative or even complex values of time. In this case we regard complex results as nonexistent and negative values as past times. We can safely disregard both cases, as we are only interested in future collisions. In the case of two positive solutions, we take the minimum one, since that event will take place first, and we are interested in the resolution of the first upcoming collision event.

This work uses this approach.

3.2 Broad Phase

Narrow phase detects only the possible collision between a pair of objects. Physical simulations are rarely required to deal with only two objects. Instead some algorithms are needed for handling arbitrary amount of objects, of which any pair can collide.

A simple but nonetheless valid idea is to do an all-pairs check. This is implemented by checking the possible collision of every object with all other objects. This obviously leads to an $O(n^2)$ algorithm where n stands for the number of objects.

Although for small numbers of n this all-pairs-check method can be a valid candidate, it becomes quickly impractical as we increase the n. It turns out that we can achieve sub-quadratic algorithms utilizing the following idea:

Some sets of objects are obviously not colliding. For example; consider a big scene with lots of objects. Do the collision checks between the objects in the far left upper corner and the objects in the far right lower corner really need to be performed? Eliminating similar cases will achieve sub-quadratic algorithms, as we will explore in this section.

3.2.1 Bounding Volumes

For achieving sub-quadratic broad-phase methods, a significant optimization method needs to be explained. Although the method of bounding volumes is not a broad-phase algorithm, it is used throughout the broad-phase algorithms, especially in bounding volume hierarchies.

As we have seen in the narrow-phase collision detection section, checking collisions between complex bodies can be computationally expensive. An important method for eliminating not colliding pairs as early and cheaply as possible is the bounding volumes.

The idea is: we can wrap the arbitrary physical bodies with simple shapes such as spheres or boxes, such that the bodies are completely inside those simple shapes. Then we can be sure that if the simple wrapping shapes do not overlap, the underlying complex shapes can not overlap. This eliminates many complex checks by doing only simple shapes collision checks. This method usually improves performance dramatically and it is used in almost all physics engines. [17]

There are many different choices for bounding volumes. They all have their own advantages and disadvantages, such as elimination ratio and computational expense of collision checks.

Since the purpose of a bounding volume is to increase efficiency, we want the bounding volumes to have efficient collision detection methods.

Bounding volumes are desired to be tight fitting, minimizing the volume that is only covered by the bounding volume and not the physical object itself. This is a desired property because it decreases the number of narrow phase collision checks performed.

Almost always, these two properties are at odds with each other. The task of a physical simulation engineer is to find a balance between them.

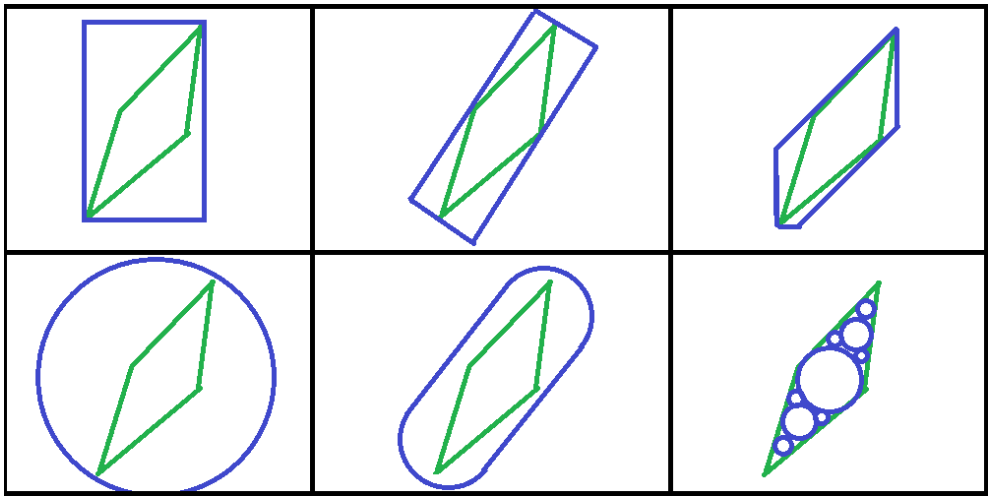

Figure 3.1: Different types of bounding volumes (blue), applied on the same object (green) in order: AABB, OBB, DOP, Sphere, Sphere-Swept, Inner Sphere Tree

3.2.1.1 Axis Aligned Bounding Box

One of the simplest shapes that we can use is the axis aligned bounding box. An axis aligned bounding box is a rectangular prism which has all its sides parallel to coordinate axes.

The practical implementation of AABB needs to store only two scalars for each dimension. If we work in two dimensions, AABB is a collection of 4 scalars. For each dimension AABB keeps track of the minimum and the maximum extend for that dimension. This makes the collision detection very easy and performant. (See algorithm: 1)

Algorithm 1 AABB collision check

```
procedure AABB COLLISION CHECK(AABB a, b)
    for each dimension x in space do
        if a.x.max < b.x.min or b.x.max < a.x.min then
            return false
        end if
    end for
    return true
end procedure
```

Although the collision detection method is fast, an axis aligned bounding box may not be tight fitting in many circumstances.(See fingure: 3.1) The next two bounding volumes will aim to improve to tightness of fitting.

AABB is the method of choice for this work, mainly due to its simplicity.

3.2.1.2 Oriented Bounding Box

There are many situations where an AABB does not fit tightly but the freedom to rotate the box results in a much tighter fitting. This idea is named oriented bounding boxes, where the boxes do not have to be parallel to coordinate axes, but can be rotated freely.

Although OBB results in a tighter fitting almost always, it has the cost of more expensive collision detection method. Instead of the simple AABB collision check, we need to use SAT (See section: 3.1.1), so that we can take orientations into account.

There are many successful implementations of this technique in use nowadays. [27]

3.2.1.3 Discrete Orientation Polytopes

AABB has limited its sides in certain directions, whereas OBB has eliminated this limitation.

Another idea is to keep the side directions set to certain angles, but increase the number of directions. This idea leads to k-DOP's where k stands for the number of allowed directions, such as 4-DOP, 6-DOP.

Having the directions fixed lets us avoid the costly computation of SAT but still gives us opportunity for tight fitting.

The cost of collision detection for DOP increases linearly with the number of allowed directions.

This leads to a direct trade-off between tight fitting and collision detection performance, as it is the main pattern of bounding volumes.

The collision check between two k-DOP's is almost the same as AABB check. (See algorithm: 2)

Algorithm 2 DOP collision check

```
procedure DOP COLLISION CHECK(DOP a, b)
    for each direction x in DOP do
        if a.x.max < b.x.min or b.x.max < a.x.min then
            return false
        end if
    end for
    return true
end procedure
```

3.2.1.4 Spheres

Another approach to bounding volumes is to use spheres. A major benefit of using spheres is their performant and simple collision detection method. (See section: 3.1.4) Depending on the circumstances, circles may be tight fitting as well. So one should always keep this option in mind when implementing collision detection systems.

A draw-back of using spheres as bounding volumes is that calculating the tight fitting sphere for arbitrary objects is costly. This is the major reason that spheres are not very common in practical collision detection methods.

3.2.1.5 Sphere Swept Volumes

A sphere swept volume is the Minkowski sum of a sphere at the origin and any arbitrary object.

A sphere swept volume can also be defined this way: A sphere of a certain radius is carried through every point of a certain object. The points that are touched by the sphere belong to the sphere swept volume. For example:

- If we sphere sweep a point, we get a sphere
- If we sphere sweep a line, we get a capsule
- If we sphere sweep a box, we get a larger box with rounded edges and corners.

The most general way to do collision detection between two sphere swept volumes is to calculate the distance between the underlying objects. If the distance is less than the total of their sphere radii, then the objects are colliding.

Note that any object is a sphere swept volume of itself with zero radius. This enables the collision detection between sphere swept volumes and any arbitrary objects.

Sphere swept volumes work well with fault tolerating circumstances and noisy data, since we can adjust the sphere radius easily. This includes the numerical precision problems in implementations using floating point arithmetic. [20]

3.2.1.6 Inner Sphere Trees

Inner sphere trees are different than other bounding volumes. They have not been designed to bound the object, but instead to fill its interior. But they can easily be adjusted to completely bound the object simply by increasing the

sphere radii. Even without completely bounding the object they are useful in fault-tolerated environments such as haptics. [28] [29]

In summary; utilizing bounding volumes leads to a significant performance increase but it still does not achieve sub-quadratic time complexity. In order to achieve sub-quadratic complexity, one needs to use spatial data structures.

There are three main categories for such spatial data structures:

1. Spatial Subdivision
2. Bounding Volume Hierarchies
3. Sort and Sweep

Data structure of choice for this work is bounding volume hierarchies.

3.2.2 Spatial Subdivision

The main idea of spatial subdivision is to divide the simulation space into sections and to associate objects with any section if they share some volume. Therefore two objects can not be colliding if they are not associated with a common section.

The choice of this division leads to different algorithms:

The simplest approach is to divide the whole space into regular rectangular spaces. This method is called "uniform grid". A rule of thumb is to keep the subdivision size a little larger then the biggest object in the scene. This method usually results in many empty grid cells, therefore one may utilize hashing. [20]

Another approach is a hierarchical tree structure. We keep dividing any subspace as long as it contains more than a certain number of objects. This leads to quadtree's or octree's depending on the dimensions of the application.

Hierarchical data structures are more adaptive than uniform ones, but they have a higher constant factor for performance, due to their more complicated implementations. This is a trade-off and the right choice depends on the application.

3.2.3 Bounding Volume Hierarchies

Instead of dividing the space of simulation one can also divide the physical objects into sets of objects and their corresponding bounding volumes.

The difference between spatial subdivision and bounding volume hierarchies is this:

- In spatial subdivision we generally do not allow the subdivisions to overlap. Any physical object can overlap with many subdivisions.
- In bounding volume hierarchies we allow subdivisions to overlap. Any object is fully bounded by a bounding volume.

The construction of bounding volumes can be listed as:

1. Top-Down
2. Bottom-Up
3. Incremental

Top-down construction is the method of choice for this work, because it provides balanced hierarchy trees.

In order to produce a hierarchy we can use this top-down approach: First let us choose a criterion, for example; any coordinate axis value. Let us take all the physical objects that we wish to simulate, and divide them into two sets according to our chosen criterion. Then recursively, let us divide these two sets into two, until we have sets of single objects. This will form a tree-like hierarchy. Let us create bounding volumes for each set in this hierarchy. This will result in a bounding volume hierarchy. For an illustration see figure: 3.2

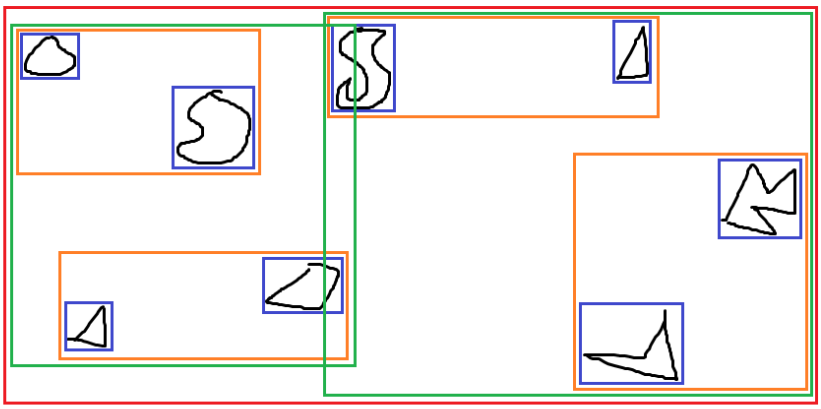

Figure 3.2: A simple AABB-hierarchy. The objects are black. The AABB levels from top to bottom are: red, green, orange, blue.

After the construction we can use the recursive algorithm to detect all pairs collisions in algorithm 3.

The complexity of algorithm 3 is $O(n^2)$ in the worst case. However, the actual performance of the algorithm depends on how many bounding volumes at each level in the hierarchy are actually intersecting, which depends on the states of the objects and on the production method of BVH. Assuming that

Algorithm 3 BVH collision check

```
procedure BVH COLLISION CHECK(BVHnode)
    if BVHnode is body then
        return
    end if
    BVH collision check pair(BVHnode.left, BVHnode.right)
    BVH collision check(BVHnode.left)
    BVH collision check(BVHnode.right)
end procedure

procedure BVH COLLISION CHECK PAIR(BVHnode1, BVHnode2)
    if BVHnode1 does not collide BVHnode2 then
        return
    end if
    if BVHnode1 is body and BVHnode2 is body then
        Narrow phase(BVHnode1, BVHnode2)
    else if BVHnode1 is body then
        BVH collision check pair(BVHnode1, BVHnode2.left)
        BVH collision check pair(BVHnode1, BVHnode2.right)
    else if BVHnode2 is body then
        BVH collision check pair(BVHnode1.left, BVHnode2)
        BVH collision check pair(BVHnode1.right, BVHnode2)
    else
        BVH collision check pair(BVHnode1.left, BVHnode2.left)
        BVH collision check pair(BVHnode1.left, BVHnode2.right)
        BVH collision check pair(BVHnode1.right, BVHnode2.left)
        BVH collision check pair(BVHnode1.right, BVHnode2.right)
    end if
end procedure
```

no bounding volumes at each level overlap, this algorithm runs in $O(n)$. It is trivially easy to produce low quality BVH's which lead to poor performance. While producing the BVH, one needs to keep the bounding volumes at each level as tight fitting as possible to eliminate useless checks.

As we will show, this method suits the needs of continuous-time collision detection very well. Therefore we will use this broad phase technique in our work.

3.2.4 Sort and Sweep

Another broad phase approach is called "sort and sweep" or "sweep and prune". The main idea is to keep sorted one dimensional projections of physical objects in many directions. Similar to the idea of separating axis theorem, a pair of objects may collide only if all of their one dimensional projections collide.

In order to achieve this check efficiently, we keep sorted starting and ending points on these one dimensional projections. Because of the temporal coherence, this sorted list does not change much usually. Applying insertion sort on nearly sorted lists are nearly linear-time efficient. So we use insertion sort to keep our lists sorted from frame to frame. On these sorted lists, it is very easy to do linear time checks of one dimensional intersections. [30]

Since this work does not make use of this technique, details are left out.

3.3 Comparison of Discrete and Continuous Time

Discrete time collision detection is a much more popular option in practice. The main reason for this is performance. Simply ignoring or postponing the resolution of some collision events within the time steps leads to less computation. However, it also leads to inaccuracies in the physics simulation. The level of inaccuracy depends on the choice of time step length and whether some additional methods are used in order to reduce the error. Regardless of how short the time step is chosen, or whether any additional error avoidance methods are used, it is clear that discrete time collision detection can not perform completely accurate simulations.

For some physical situations, the inaccuracy of discrete time collision detection leads to unacceptable simulation artifacts. For example, in a shooter game, game characters usually throw objects at each other, whether they are bullets, bombs, laser beams or any other game object. Some of these objects, such as bullets, can be extremely fast moving. Since many games use discrete time collision detection, it can be easy to miss a collision between a bullet and an object. The probability of missing a collision grows as the objects are smaller and faster. Missing a collision between a bullet and an enemy hinders the game enjoyment greatly. Therefore physics simulation programmers

have come up with ways to avoid this issue. Almost all of the solutions involve creating a temporary large object, containing the fast moving object at its two consecutive positions between time steps. If this large object intersects with any other object, they are assumed to be colliding between time steps. Even though it is a commonly used method, it has its problems. One of them is that this method does not handle rotations, because the orientations in-between the frames are not captured by the temporary object.[17] Assuming that the method catches a valid collision, the physics simulation handles the collision too late, leading to physical inaccuracies.

The order of collisions are not preserved under discrete time collision detection. Although some methods are used in order to estimate the collision time, [9] they are only heuristics. Sometimes the order of collisions can be important because the collisions can be interdependent. The resolution of a collision may result in another collision or prevent some other collision from happening.

The best example for the problems with discrete time collision detection is the simulation of a billiard game. For this type of a physical simulation it is vital to handle the collisions at their exact times and positions. Missing the slightest collision leads to completely different game states. Also interdependent collisions happen very frequently. Therefore we have developed a physical simulation similar to billiard for our work.

If done correctly, continuous time collision detection eliminates all these problems, because the collisions are detected exactly when they happen, so that they can be resolved immediately and therefore accurately.

One may wonder why continuous time collision detection is so rarely used in practice, even though it solves so many problems and always leads to more accurate physical simulation. The main reason for the lack of popularity of continuous time collision detection is performance. Even from the narrow phase collision detection code of circles that we have described earlier, it is obvious that the handling of dynamic objects are much more complex than the static ones, both in terms of performance and implementation. Another aspect for the performance issue is this: It is obvious that there can be at most $O(n^2)$ collisions at any time step in the case of discrete time collision detection, whereas the number of collisions for continuous time collision detection is unbounded.

Another limitation of continuous time collision detection involves the rotation. Continuous-time collision detection between arbitraryly shaped, rotating objects leads to complex equations involving trigonometric terms. [9] Even such a simple equation(3.15) may not have a solution with constant time complexity.

$$\cos(x) = x \tag{3.15}$$

When we take rotation into account for collision detection, many trigonometric terms get involved in the equations. Few of them have algebraic solutions,

which are fast to calculate. Almost all of them require numeric approximation methods, which are not completely accurate and precise. It is always possible to miss some results due to local extremes in the equation. They are also computationally expensive.

For games physics simulation, the most important question is: "Is the error noticeable?". For many games, such as action platformers, the errors due to discretization is rarely noticeable. For some games where missed collisions pose a great threat, we can use preventive methods for very fast objects. These methods are good enough for almost all first person shooter games. But for some games, such as billiard, the accuracy provided by continuous time collision detection is necessary.

It is worth noting that continuous-time physics simulation is a necessity in the case that one can not tolerate any missed collisions, such as scientific computations.

CHAPTER 4

PROPOSED METHOD

4.1 Broad-Phase for Continuous-Time Collision Detection

During research one can find that the usage of broad-phase methods for discrete-time collision detection is well established and practically applied in almost all physics engines. Although there are many choices for discrete-time broad-phase algorithms (See 3.2), we have not found any broad-phase algorithms for continuous-time collision detection, which has sub-quadratic space complexity. Without any broad-phase methods, the only available collision detection method is the naive all-pairs check, which results in $O(n^2)$ time complexity. We wish to bridge this gap by introducing a novel continuous-time broad-phase collision detection algorithm.

The proposed work successfully optimizes the continuous time collision detection algorithm by adding a novel broad phase step. The naive all pairs check method has $O(n^2)$ time complexity and $O(n)$ space complexity. The proposed method keeps the space complexity at $O(n)$ but achieves an average time complexity of $O(n \log(n))$. Although the worst case time complexity is still $O(n^2)$, practical running time measurements clearly show that the proposed method has sub-quadratic time complexity.

As demonstration, the proposed method and a visualization for it have been implemented. The physical simulation code and visualization code are kept separate, in order to keep the running time measurements more accurate. Two programs communicate via files. The main purpose of the visualization tool has been to ensure that the physical simulation gives correct outputs by not missing or miscalculating any collisions.

The implementation is two dimensional for the ease of visualization and debugging. All of the methods we propose are trivial to convert to three dimensions. Adding another dimension does not complicate the sphere representation, which is identical to circle representation. Calculating the collision time of a pair of spheres can again be simplified to a quadratic equation, as in the case of circles. Axis aligned bounding boxes have already been invented originally for three dimensional objects and do not require any more sophistication when converted from two dimensions. Similarly bounding volume hierarchy algorithm stays identical. The only extra work arises in the visu-

alization of three dimensional physical simulation for the proposed method, which is not the focus of this work. So three dimensional version of the proposed method has been regarded as a trivial exercise.

As shown earlier (See equation: 3.15), rotation in continuous time collision detection is a complicated task for arbitrary objects. It is also unrelated to broad phase, since all broad phase methods, including this method, work with bounding volumes, making the complexities of the objects irrelevant. Therefore only circles and static lines have been used for the implementation. Working with circles eliminated the problem of rotations. Although this may seem to limit the usefulness of the proposed work, the circles or spheres can be used as a bounding volume for any object. It is also trivial to convert dynamic circles to dynamic AABB's. Therefore the proposed approach can be utilized under any circumstances.

The only way that the rotation of the dynamic circular objects could have any impact on the simulation output would be to apply friction on the collision response algorithms. We have also disregarded the collision friction altogether, since the collision responses do not effect collision detection in any way, because the focus of this work is on the optimization of collision detection.

The linear drag of the simulation, also known as air friction, is not implemented and left for future work. The reason for this is that there are numerous different implementations to simulate linear drag, and the narrow-phase continuous time collision detection algorithm changes according to the model of linear drag applied. Another reason for this choice is that the linear drag does not change the broad-phase algorithms, which is the main focus of this work.

4.2 Axis Aligned Bounding Boxes

Axis aligned bounding boxes have been chosen for the implementation, but the proposed approach is suitable for any bounding volumes. The main reason for this choice has been the ease of implementation, compared to methods such as oriented bounding boxes.

4.3 Bounding Volume Hierarchy

Bounding volume hierarchies have been the broad phase technique of choice, because the proposed method is an algorithm to apply them on continuous-time collision detection efficiently, whereas other broad-phase methods for discrete-time collision detection would require radically different ideas to be applicable to continuous-time collision detection.

The continuous-time applications of the other discrete-time broad-phase methods such as "spatial subdivision" and "sort and sweep" have the potential to

be the topic of further research.

4.4 Adaptive Time Step

Adaptive time step is the crucial idea of the proposed method to be adaptable for continuous-time collision detection. In discrete-time collision detection, only the overlapping object pairs are detected without taking their velocities into consideration. Therefore the bounding volumes need to cover only the objects at their position at any given time. In continuous-time physics, the simulation does not let objects to overlap, instead it calculates the times of first contacts and resolves those events in order. Therefore covering only the objects does not let a bounding volume hierarchy algorithm to detect future collisions, since they do not take velocities into account at all.

In order to adapt bounding volume hierarchies for continuous-time collision detection, a method is needed to let bounding volumes collide, in the case that the underlying objects will be colliding in a time interval that is interested in. If this time interval is chosen as infinite, the bounding volumes will have infinite size, which will result in a correct algorithm, but with quadratic time complexity.

The main idea of the proposed method is to keep an adaptive time step, and make the bounding volumes cover the object at any given time and the object after the chosen time step. The optimal choice of the time step is the time required for the first collision in the simulation. However, since this value is what needs to be calculated in a continuous-time collision detection algorithm, binary search can be used to determine this value, in order to achieve logarithmic time complexity.

This is the reason why a time step parameter is still needed for the proposed method, although continuous time collision detection is performed. This parameter will be self adjusting, so there is no choice of time step as in the case of discrete time collision detection.

The only need for the time step length is to determine the sizes of the AABB's that contain only one object. The rest of the AABB's will be dependent on these bottom AABB's as they will be determined in a bottom-up fashion.

If the next collision event within the current time step length cannot be detected, the time step will be increased by multiplying with a constant. The time step length will be increased until the next event is detected. After the next event is found, the time step will be decreased, so that it stays adaptive, not strictly increasing. We decrease by multiplying it with another constant.

Increasing the time step length will increase the sizes of the bottom AABB's so the next event that was previously not detected, will be detected after a certain number of increases.

Since any constant will yield a logarithmic time complexity, the increasing

constant has been chosen as 2 and the decreasing constant as $1/2$ arbitrarily. Although different choices of these constants will not change the time complexity of the algorithm, one can most probably find better constant values, which yield better running times.

4.5 Top-down Divide by Median Tree Construction

Top-down hierarchy construction has been chosen, mainly due to ease of balanced tree creation. The construction of the bounding volume hierarchy tree is almost the same as the classical discrete time collision detection case. The illustration of the technique can be fount at 3.2.

The axis aligned bounding boxes are built for the bodies as they will include the body and the body after the current time step. So AABB will take velocities into account. After initializing the bottom AABBs, the tree is constructed in a top-down fashion. Objects are divided into two equal sets with respect to a coordinate axis in each turn. The axis of choice depends on the level of the tree.

This procedure will result in a balanced tree, which can be easily represented as an array, avoiding memory fragmentation and pointer chasing.

After determining the hierarchy, the upper AABB's are determined by the lower AABB's as they will enclose their descendants.

4.6 Partitioning an array

During top-down construction of the hierarchy, the AABB's will need to be partitioned so that the median is in the middle of the array, all elements that are less than the median are in the first half of the array and the greater values are in the second half of the array. Equalities are broken arbitrarily.

It is a proven fact that we can determine the median of an array in linear time. [31] After determining the median, the array can be divided into two parts in linear time as well. In the proposed implementation C++/STL "nth_element" function is used for this task, which is documented to have linear time complexity. [32]

4.7 Pseudocode

Algorithm 4 describes the full version of the proposed method.

Algorithm 4 Continuous Collision Detection

```
procedure GET NEXT EVENT(objects)
    Construct hierarchy (objects)
    Next Event = none
    BVH collision check (root of hierarchy)
    while Next Event is none do
        time step = time step * increasing constant
        Update Hierarchy
        BVH collision check (root of hierarchy)
    end while
    time step = time step * decreasing constant
    return Next Event
end procedure

procedure CONSTRUCT HIERARCHY(objects)
    Construct hierarchy recursive(objects, 0)
end procedure

procedure CONSTRUCT HIERARCHY RECURSIVE(objects, level)
    if objects has single object then
        return AABB enclosing object(now), object(now + timestep)
    end if
    axis = axis of level
    median = median of objects in axis
    half1 = objects before than median
    half2 = objects after than median + median
    AABB1 = Construct hierarchy recursive(half1, level + 1)
    AABB2 = Construct hierarchy recursive(half2, level + 1)
    return AABB enclosing AABB1, AABB2
end procedure

procedure UPDATE HIERARCHY
    for BVHnode from leaves to root do
        if BVHnode contains single object then
            Update BVHnode to cover object(now), object(now + timestep)
        else
            Update BVHnode to cover both children
        end if
    end for
end procedure
```

```
procedure BVH COLLISION CHECK(BVHnode)
    if BVHnode is body then
        return
    end if
    BVH collision check pair(BVHnode.left, BVHnode.right)
    BVH collision check(BVHnode.left)
    BVH collision check(BVHnode.right)
end procedure

procedure BVH COLLISION CHECK PAIR(BVHnode1, BVHnode2)
    if BVHnode1 does not collide BVHnode2 then
        return
    end if
    if BVHnode1 is body and BVHnode2 is body then
        Narrow phase(BVHnode1, BVHnode2)
    else if BVHnode1 is body then
        BVH collision check pair(BVHnode1, BVHnode2.left)
        BVH collision check pair(BVHnode1, BVHnode2.right)
    else if BVHnode2 is body then
        BVH collision check pair(BVHnode1.left, BVHnode2)
        BVH collision check pair(BVHnode1.right, BVHnode2)
    else
        BVH collision check pair(BVHnode1.left, BVHnode2.left)
        BVH collision check pair(BVHnode1.left, BVHnode2.right)
        BVH collision check pair(BVHnode1.right, BVHnode2.left)
        BVH collision check pair(BVHnode1.right, BVHnode2.right)
    end if
end procedure

procedure NARROW PHASE(circle1, circle2)
    collision time = calculate collision time(circle1, circle2)
    if collision time > time step then
        return
    end if
    if collision time < global minimum collision time then
        global next event = collision of circle1, circle2
    end if
end procedure
```

CHAPTER 5

RESULTS AND DISCUSSION

The proposed method has been implemented in C++ and the output is visualized in order to make sure that the results are correct. Both the naive check-for-all-pairs method and the proposed novel broad phase algorithm are implemented. It is empirically shown that the proposed algorithm runs asymptotically faster than the naive approach.

Sedgewick's [19] priority-queue based approach has not been implemented for comparison, as it quickly becomes infeasible as the number of objects grows, because of the quadratic memory complexity. For example, the naive method and the proposed approach can theoretically work with a million objects but the memory required by Sedgewick's method for a million object is over a terabyte. This amount of memory may be feasible for scientific computation but for game simulations it is out of reach with today's hardware.

5.1 Complexity Analysis

As in all broad phase collision detection methods, it is impossible to achieve a sub-quadratic asymptotic time complexity bound in the worst case, because at any simulation $O(n^2)$ collisions may happen simultaneously. But in practice broad phase algorithms achieve a significant performance boost and they are used in almost all physics engines.

Although we can not prove sub-quadratic time complexity in the worst case, we shall examine each part of the algorithm separately.

- **Constructing the hierarchy** is $O(n \log n)$. The analysis is almost the same as merge-sort. Since we divide the array from the median, we divide into two equal sized parts. The depth of the recursion is $\log n$, where each level works in linear time with respect to amount of all objects.

$$T(1) = 1 \tag{5.1}$$

$$T(n) = 2T(n/2) + n = O(n \log(n)) \tag{5.2}$$

- **Updating the hierarchy** is $O(n)$. A constant time is needed to update each bounding volume and there are $2n - 1$ many bounding volumes in total. There are two cases for a bounding volume:

  1. The bounding volume may be a leaf in the hierarchy tree. To update this bounding volume, we only need to calculate the current and after time step positions of the underlying object and cover them. This is done in constant time.
  2. The bounding volume may be a non-leaf node in the hierarchy tree. To update this bounding volume, we only need to update the bounds of this bounding volume to cover both of its children.

- **Narrow phase of collision detection** is $O(1)$ assuming that a square-root operation takes constant time.

- **Checking collisions through the bounding volume hierarchy** is $O(n^2)$ in worst case. The practical running time depends on the quality of the generated hierarchy and the states of the objects. In practice this step behaves more closely to $O(n \log n)$ as can be seen at the running time measurements.

- **Number of time step increases** is logarithmic with respect to the time of next collision, since we multiply the time step with a constant in each iteration. In practice this step behaves more closely to constant time, but we shall consider this step as $O(\log t)$, where t stands for the next time of collision.

- **In total:** Assuming that checking the bounding volume hierarchy behaves $O(n \log n)$ as it does in practice, our algorithm finds the next event in $O(n \log n + n \log t)$, where n stands for the number of objects and t stands for the time of next collision. We consider this complexity sub-quadratic as the practical running time measurements confirm.

5.2 Running Times

We have implemented the naive all-pairs-check algorithm and our novel broad phase method in order to compare performance. We have chosen C++ as the programming language, because it seems to be the dominant choice for game physics development.

In order to make sure that the physical simulation works correctly, we have developed a software visualization tool written in SDL2.0 framework.

Here are the environment specifications:

- CPU: Intel i7-2630QM @2.00GHz
- RAM: 8.00GB DDR3

Table 5.1: Running times in seconds (average of 100 runs)

Number of objects	Number of collisions	All pairs check	Our broad phase
8	**52**.28	**0**.000193307	**0**.000337453
16	**213**.15	**0**.001134925	**0**.000512313
32	**628**.46	**0**.009495239	**0**.005448446
64	**1700**.78	**0**.091004501	**0**.032925012
128	**4670**.91	**1**.009229282	**0**.172442404
256	**12138**.45	**9**.716522560	**1**.101287737
512	**33332**.82	**105**.5300373	**6**.773387463
1024	**89060**.59	**1273**.946865	**46**.33965052
2048	**246636**.2	**13085**.64345	**264**.8951511

- OS: Windows7 Professional SP1
- Compiler: Visual Studio 2013 in default release options
- Floating point precision: double

We have created randomized scenes to test performance with constant object density and relatively constant velocities, regardless of the scene size. The scene is similar to a billiard table, where 4 lines cover a rectangular area and prevent the circles from going outside. The velocities are uniformly randomized in an interval, where the maximum velocity can take an object from a corner of the scene to the opposite corner in one second, regardless of the size of the scene. The sizes of the circles are also uniformly randomized. The scene size changes according to the number of circles, in order to keep the object density constant. The benchmark simulates 5 seconds of physics for each test.

In order to reduce noise in the running time measurement data, the test cases have been carried out a hundred times and the averages are taken, except for the last test case with 2048 objects. The last test case has been executed 10 times, because of the long time requirement.

We have also tested custom scenes such as the beginning formation of a billiard table. We have soon realized that the simulation goes into the chaotic randomized state extremely quickly, which made the initial formations of objects irrelevant.

One should note that the scene setups and the outputs of both all-pairs-check and our novel collision detection methods are exactly the same. They have the same space complexity, but they have radically different time complexities, which is confirmed by the running time measurements at table 5.1.

The running time measurements have also been plotted in a scaled log-log chart in figure 5.1. The y-axis represents running times in logarithmic scale.

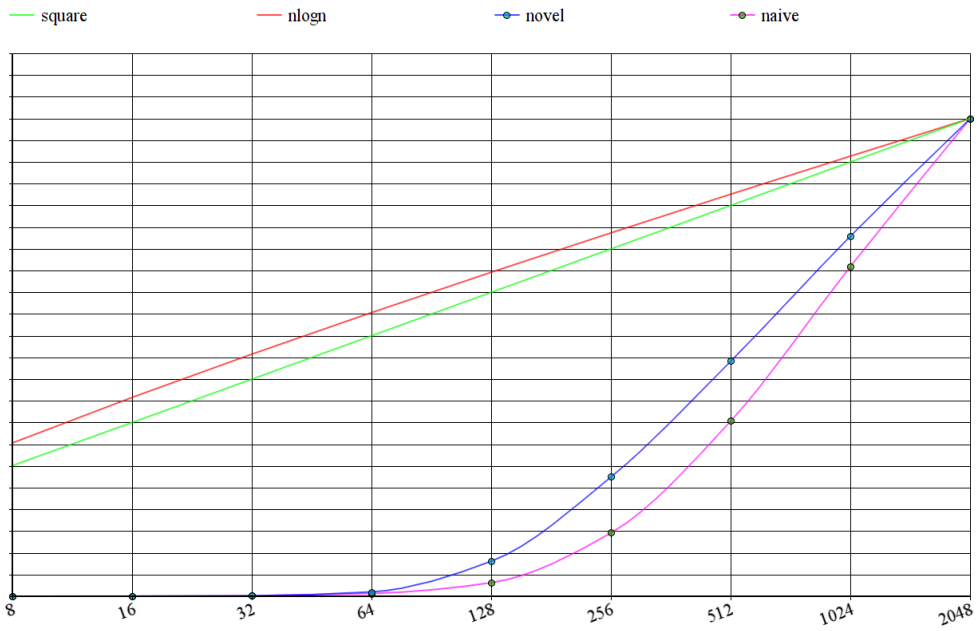

Figure 5.1: Running time measurements compared to sample functions in scaled log-log plot, where x axis represents number of objects and y axis represents running times

The x-axis represents running times in logarithmic scale. An $O(n^2)$ and an $O(n \log n)$ function have been plotted for reference. The running times of our approach and the naive approach have also been plotted. All four lines have been scaled to have the same maximum value. The reason why this chart lacks to show that our approach runs closer to $O(n \log n)$ compared to $O(n^2)$ is suspected to be the low number of objects in the scene. The constant factors play significant role in the running times and the data is noisy. Increasing the running times is not practical due to long time requirements for measurements.

It is also worth mentioning that these are not per frame time requirements, but simulation durations of 5 seconds, which would correspond to 300 frames in a realistic setup. The objects in the simulations have very high velocities, enough for any object to travel from one corner of the scene to the opposite corner in one second. Therefore the collision event counts are much higher than a realistic game setup. One should expect shorter running times in a realistic game scenario.

5.3 Comparison with Discrete-Time

A comparison between our approach and the discrete-time bounding volume hierarchy approach has also been carried out. The physical setup is the same as the previous setup with 1024 objects. The tests have been executed a hundred times and taken averages to reduce noise. Although the discrete-time physics simulation provides better running times in general, it misses some

collisions as expected. The number of misses depends on the chosen time step. The results are shown in table 5.2.

Table 5.2: Comparison with discrete-time

Algorithm	Running-Time (seconds)	Number of Collision Misses
Continuous-Time	**46**.33965052	0
Discrete-Time (dt = 0.001)	**236**.7274949	**0**.17
Discrete-Time (dt = 0.005)	**57**.51156817	**1**.93
Discrete-Time (dt = 0.01)	**29**.52776001	**32**.87
Discrete-Time (dt = 0.03)	**13**.82739977	**156**.06
Discrete-Time (dt = 0.1)	**5**.480592384	**3416**.14
Discrete-Time (dt = 1)	**1**.893161469	**87281**.33

The results clearly indicates a trade-off between performance and accuracy for different algorithms. There is a direct and simple trade-off between performance and accuracy for discrete-time physics simulation with different choices of time steps, whereas the continuous-time physics simulation is always completely accurate but its performance is very much dependent on the physical setup.

Different physical scenarios will provide significantly different performance outputs. Therefore one should make use of appropriate profiling tools before deciding on the physical simulation algorithm.

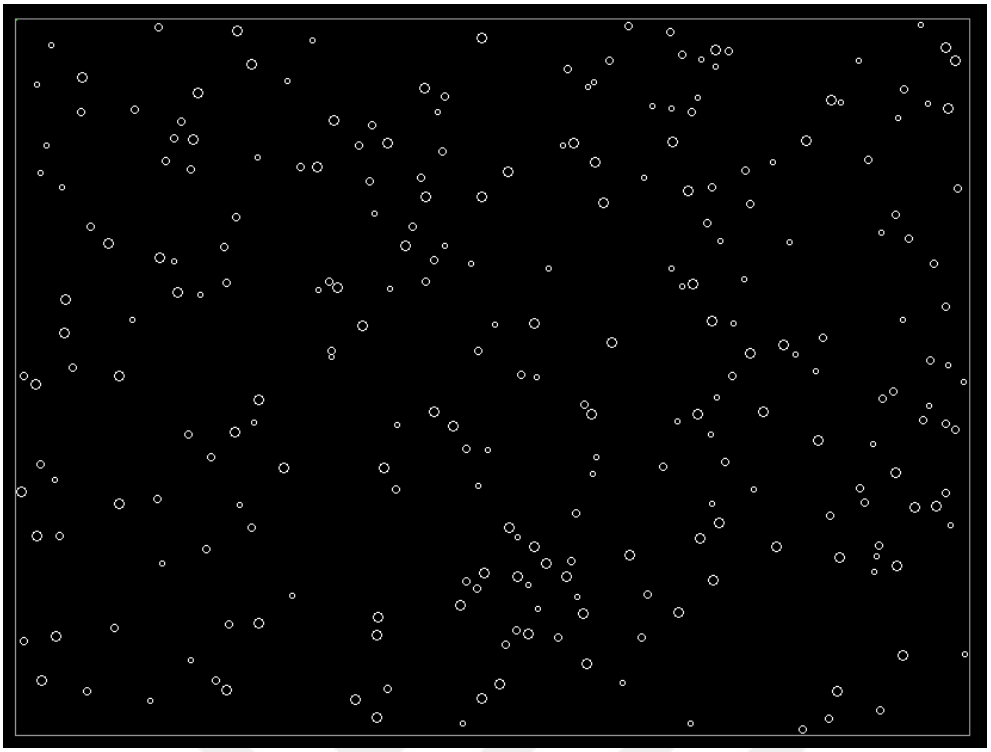

Figure 5.2: A screen-shot from our visualization tool

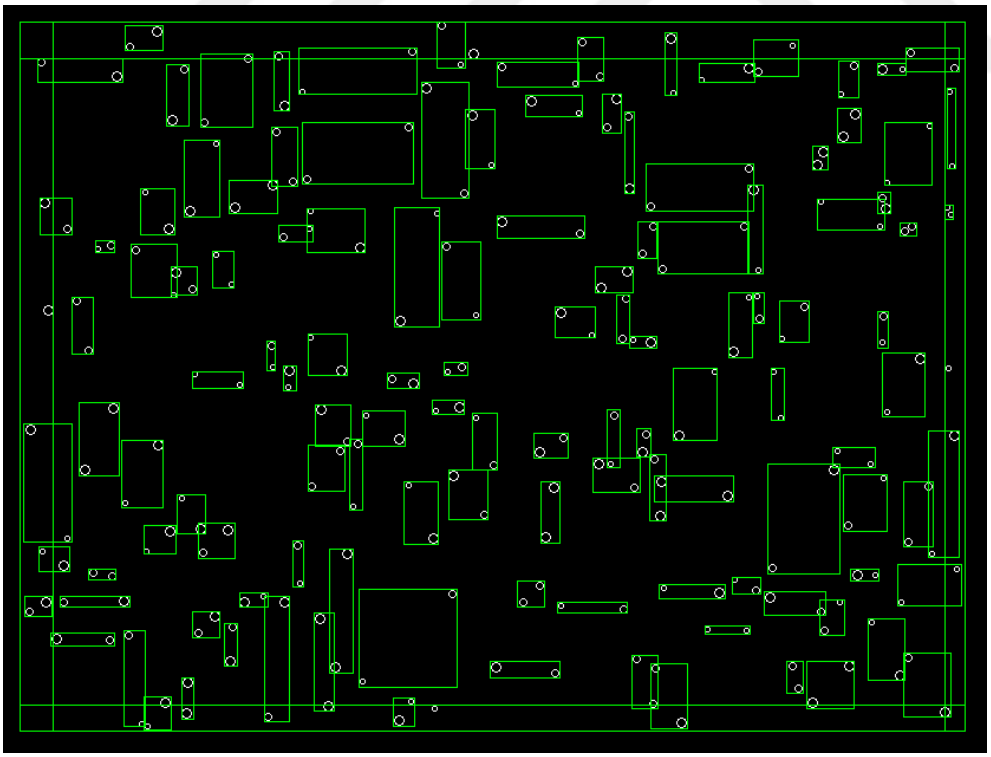

Figure 5.3: A screen-shot from our visualization tool displaying some AABBs

CHAPTER 6

CONCLUSION AND FUTURE WORK

A novel broad phase method that optimizes physical simulations which use continuous time collision detection has been presented. The proposed method can solve the broad phase needs of almost all continuous time physical simulations. The detailed narrow phase continuous time collision detection has not been concerned with. Only circles have been used for simulation to avoid rotations.

Although the proposed method still has a time complexity of $O(n^2)$ for the worst case and this does not constitute a theoretical improvement, empirical measurements of the running time confirm that it provides a significant improvement for practical cases.

Bounding volume hierarchies have been chosen as the broad phase method. Bounding volume hierarchies have been well developed and understood for discrete time collision detection. The contribution of this work has been adapting them for continuous time collision detection. Some future work may involve adapting other broad phase methods for continuous time collision detection, such as "spatial subdivision" and "sort and sweep".

This work can be seen as a proof of concept. The proposed method and implementation have not optimized to the furthest. Even in this state, it is clear from the results that the proposed approach is valid. Some implementation details may yield great gains in performance such as:

- AABB has been used for the ease of implementation. It is suspected that OBB may be better suited for continuous time collision detection.
- Top-down construction methods have been used for bounding volume hierarchies by dividing from median. There are other known hierarchy construction methods, which can be experimented with.
- The bounding volume scaling constants are chosen as 2 and $1/2$ arbitrarily. These values can be optimized further, which will have different optimal values according to setup of physical objects.
- Parallel computing has not been utilized for the proposed approach. Multi-thread programming, CPU-SIMD and GPGPU can result in great performance increase.

Another future work for the improvement of the proposed work may be to include drag to physical simulation. Throughout the work, constant velocities and no acceleration have been assumed, besides the collision impulses. Drag has been neglected, because it does not complicate the broad phase methods, which has been the main focus for this work. Adding drag may complicate the narrow phase collision detection, depending on the model of drag chosen. For some models of drag, the solution of the narrow phase collision detection will stay a quadratic equation. But some other models of drag may complicate the solution. A through investigation of this issue may be needed for practical uses of the prop work.

REFERENCES

[1] Sam Lantinga. Simple DirectMedia Layer. `https://www.libsdl.org/`, 2015. Online; accessed 13-November-2015.

[2] Ming Chieh Lin. *Efficient Collision Detection for Animation and Robotics*. PhD thesis, 1993. AAI9430587.

[3] Paul Ploumhans, Grégoire Winckelmans, John Salmon, Anthony Leonard, and Michael Warren. Vortex methods for direct numerical simulation of three-dimensional bluff body flows: Application to the sphere at re = 300, 500, and 1000. *J. Comput. Phys.*, 178(2):427–463, May 2002.

[4] Matteo Viel and Martin G Haehnelt. Cosmological and astrophysical parameters from the sloan digital sky survey flux power spectrum and hydrodynamical simulations of the lyman α forest. *Monthly Notices of the Royal Astronomical Society*, 365(1):231–244, 2006.

[5] Erdogan Madenci and Ibrahim Guven. *The Finite Element Method and Applications in Engineering Using ANSYS*. Springer Publishing Company, Incorporated, 2nd edition, 2015.

[6] Rubin Landau, Jose Paez, and Cristian Bordeianu. *A Survey of Computational Physics: Introductory Computational Science*. Princeton University Press, Princeton, NJ, USA, har/cdr edition, 2008.

[7] Atul Thakur, Ashis Gopal Banerjee, and Satyandra Gupta. A survey of cad model simplification techniques for physics-based simulation applications. *Comput. Aided Des.*, 41(2):65–80, February 2009.

[8] Masanobu Shinozuka and C.-M. Jan. Digital simulation of random processes and its applications. *Journal of Sound Vibration*, 25:111–128, November 1972.

[9] Ian Millington. *Game Physics Engine Development, Second Edition: How to Build a Robust Commercial-Grade Physics Engine for Your Game*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2010.

[10] NVIDIA. APEX Clothing Module. `http://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/apexsdk/APEX_Clothing/Clothing_Module_Doc.html`, 2015. Online; accessed 13-November-2015.

[11] Edward Witten. String theory dynamics in various dimensions. *Nucl. Phys. B*, page 9503124, 1995.

[12] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.

[13] Tiantian Liu, Adam Bargteil, James O'Brien, and Ladislav Kavan. Fast simulation of mass-spring systems. *ACM Trans. Graph.*, 32(6):214:1–214:7, November 2013.

[14] Matthias Müller, Jos Stam, Doug James, and Nils Thürey. Real time physics: Class notes. In *ACM SIGGRAPH 2008 Classes*, SIGGRAPH '08, pages 88:1–88:90, New York, NY, USA, 2008. ACM.

[15] Ernst Hairer, Christian Lubich, and Gerhard Wanner. Geometric numerical integration illustrated by the störmer–verlet method. *Acta Numerica*, 12:399–450, 5 2003.

[16] Russell Smith. Open Dynamics Engine. `http://www.ode.org/`, 2007. Online; accessed 13-November-2015.

[17] Erin Catto. Box2D. `http://box2d.org/`, 2015. Online; accessed 13-November-2015.

[18] Scott Lembcke. Chipmunk2d. `https://chipmunk-physics.net`, 2015. Online; accessed 13-November-2015.

[19] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley Professional, 4th edition, 2011.

[20] Christer Ericson. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology) (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[21] Eran Guendelman, Robert Bridson, and Ronald Fedkiw. Nonconvex rigid bodies with stacking. *ACM Trans. Graph.*, 22(3):871–878, July 2003.

[22] Stefan Aric Gottschalk. *Collision Queries Using Oriented Bounding Boxes*. PhD thesis, 2000. AAI9993311.

[23] Gino Van Den Bergen and Gino Johannes Bergen. *Collision Detection*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1 edition, 2003.

[24] Henk Bekker and J. B. T. M. Roerdink. An efficient algorithm to calculate the minkowski sum of convex 3d polyhedra. In *Proceedings of the International Conference on Computational Sciences-Part I*, ICCS '01, pages 619–628, London, UK, UK, 2001. Springer-Verlag.

[25] Gino Van den Bergen. A fast and robust gjk implementation for collision detection of convex objects. *J. Graph. Tools*, 4(2):7–25, March 1999.

[26] Brian Mirtich. V-clip: Fast and robust polyhedral collision detection. *ACM Trans. Graph.*, 17(3):177–208, July 1998.

[27] Thomas Hudson, Ming Lin, Jonathan Cohen, Stefan Gottschalk, and Dinesh Manocha. V-collide: Accelerated collision detection for vrml. In *Proceedings of the Second Symposium on Virtual Reality Modeling Language*, VRML '97, pages 117–ff., New York, NY, USA, 1997. ACM.

[28] Ren Weller. *New Geometric Data Structures for Collision Detection and Haptics*. Springer Publishing Company, Incorporated, 2013.

[29] Carol O'Sullivan and John Dingliana. Real-time collision detection and response using sphere-trees, 1999.

[30] Daniel Tracy, Samuel Buss, and Bryan Woods. Efficient large-scale sweep and prune methods with aabb insertion and removal. In *Proceedings of the 2009 IEEE Virtual Reality Conference*, VR '09, pages 191–198, Washington, DC, USA, 2009. IEEE Computer Society.

[31] Thomas Cormen, Clifford Stein, Ronald Rivest, and Charles Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[32] Microsoft. STL documentation. `https://msdn.microsoft.com/en-us/library/7s2yb954.aspx`, 2015. Online; accessed 13-November-2015.