San Jose State University
SJSU ScholarWorks

Master's Projects | Master's Theses and Graduate Research

Spring 2014

Parallelized Rigid Body Dynamics

John Calvin Linford
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the Computer Sciences Commons

Recommended Citation
Linford, John Calvin, "Parallelized Rigid Body Dynamics" (2014). *Master's Projects*. 343.
DOI: https://doi.org/10.31979/etd.w7c5-2h52
https://scholarworks.sjsu.edu/etd_projects/343

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

Parallelized Rigid Body Dynamics

A Writing Project
Presented to
The Faculty of the Department of Computer Science
San Jose State University

In Partial Fulfillment
Of the Requirements for the Degree
Masters of Science

By
John Calvin Linford
December 2013

©2013

John Calvin Linford

ALL RIGHTS RESERVED

The Designated Project Committee Approves the Project Titled

Parallelized Rigid Body Dynamics

By

John Calvin Linford

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

DECEMBER 2013

Dr. Teng Moh	Department of Computer Science
Dr. Robert Chun	Department of Computer Science
Dr. Chris Pollett	Department of Computer Science

// ABSTRACT

Parallelized Rigid Body Dynamics

by John Calvin Linford

Physics engines are collections of API-like software designed for video games, movies and scientific simulations. While physics engines often come in many shapes and designs, all engines can benefit from an increase in speed via parallelization. However, despite this need for increased speed, it is uncommon to encounter a parallelized physics engine today. Many engines are long-standing projects and changing them to support parallelization is too costly to consider as a practical matter. Parallelization needs to be considered from the design stages through completion to ensure adequate implementation.

In this project we develop a realistic approach to simulate physics in a parallel environment. Utilizing many techniques we establish a practical approach to significantly reduce the run-time on a standard physics engine.

ACKNOWLEDGMENTS

I would like to thank Dr. Teng Moh for his guidance throughout this project. His insight, advice and guidance throughout the project was invaluable.

I thank Dr. Soon Tee Teoh for his guidance during the preparation of this project. I also thank Dr. Robert Chun and Dr. Chris Pollett for serving on my defense committee.

TABLE OF CONTENTS

CHAPTER

1. Introduction **1**

2. Physics Engine Basics **4**

2.1. Specialized Engines 4

2.1.1. Dimensions 4

2.1.2. Rigid Body vs. Soft Body 4

2.1.3. Fluid Dynamics 5

2.1.4. Electromagnetism 5

2.2. Terminology 5

2.2.1. The World 5

2.2.2. Contact 6

2.3. Rigid Body Features 6

2.3.1. Collision Detection 7

2.3.1.1. Rough Collision Detection 7

2.3.1.2. Fine Collision Detection 7

2.3.2. Collision Resolution 9

2.3.2.1. Contact Generation 9

2.3.2.2. Contact Resolution 9

2.3.3. Integration 10

2.3.4. Cleanup 10

3. Previous Work **11**

3.1. Box2D 11

3.2. Pe 12

3.3. Nvidia's PhysX 15

4. Design and Implementation **17**

4.1. Rigid Body Dynamics 17

4.1.1. Mass 18

4.1.2. Vertices 18

4.1.3. State 19

4.1.4. Linear Velocity 19
4.1.5. Angular Velocity 20
4.1.6. Linear Acceleration 20
4.1.7. Angular Acceleration 20
4.1.8. Moment of Inertia 20
4.1.9. Linear Momentum 21
4.1.10. Angular Momentum 21
4.2. Collision Detection 21
4.2.1. Discrete Collision Detection 21
4.2.2. Broad Phase 22
4.2.2.1. Quad-Tree 23
4.2.2.2. Bounding Box 24
4.2.3. Fine Phase 25
4.2.3.1. Circle vs. Circle 25
4.2.3.2. Circle vs. Polygon 26
4.2.3.3. Polygon vs. Polygon 27
4.3. Collision Response 28
4.3.1. Penetration Resolution 29
4.3.2. Contact Resolution 29
4.4. Integration 32
4.5. Parallelism 32
4.5.1. Task Partitioning 32
4.5.2. Data Partitioning 33
4.5.3. Pitfalls 33
4.5.4. Parallel Implementation 35
4.5.4.1. Parallel Quad-Trees 35
4.5.4.2. Parallel Contact Resolution 36
4.6. Software Engineering 37
5. Results 39
5.1. Circle Stress 40
5.2. Polygon Stress 42
5.3. Mobile Application 42

5.4. Conclusions 43

6. Further Work and Conclusion 44

6.1. Continuous Collision 44

6.2. Sleep 45

6.3. Warm Startup 45

6.4. Memory Pooling 46

6.5. Conclusion 46

LIST OF FIGURES

1. Box2D portrayed through “Angry Birds” 2
2. Cinema4D portraying soft body dynamics 5
3. Bounding boxes 8
4. Process view for pe 13
5. pe simulating 500,000 rigid bodies 15
6. PhysX performance gains 16
7. Two objects penetrating one another 22
8. A regional quad-tree 23
9. Visual Minkowski Difference 27
10. Data partitioning used to parallelize tasks 33
11. Quad-tree modified to support four threads 35
12. Diagram outlining basic class relationship 38
13. Circle stress screenshot 41
14. Circle stress test results 41
15. Polygon stress test results 42
16. Mobile Application Example 43

CHAPTER 1

Introduction to Physics Engines

Physics engines generally come in two forms: high-precision and real-time engines. High-precision engines attempt to arrive at the most physically accurate answer possible. Very few, if any, shortcuts are taken designing and building these engines and as a result they are often quite slow. Industries that use these engines often have physical accuracy as their utmost priority and so they regretfully accept the slow downside that these engines bring. Real-time engines are on the opposite side of the spectrum. They look to maximize speed even at the cost of physical accuracy. Industries that use these real-time engines have run-time as their utmost concern.

The most common use of a physics engine is in video games. Physics engines used for video games are nearly always real-time engines. In order to provide a smooth experience for gamers, video games must maintain high frame rates. The frame rate a game is outputting is directly correlated to how quickly the game code, including the physics, can complete its cycles. Games that have processes hogging too much CPU time will become slow and unplayable.

Video games frequently have quite a few resources competing for CPU time, from the drawing of sprites and graphics to playing sound to checking for user input. If the video game is physically intensive, computing physics is an additional CPU-intensive task that will be competing for CPU usage against these other tasks. As a result it is extremely important for efficient video games that the physics engine is not over utilizing CPU time.

To obtain the desired end results, many video game physics engines will use estimations and approximations. The concept behind these engines is not to get the most physically correct answer, but to get a believable answer as quickly as possible. Some popular real-time physics engines for games include Box2D [1] and Nvidia’s PhysX [2].

Figure 1. Popular video game "Angry Birds" relies heavily on physics engine Box2D.

Alternatively, high precision engines will not take the shortcuts that real-time engines do. This leads to much more accurate results, but there is a large cost in terms of time. These high-precision engines also often are simulating more physical phenomena as well. While phenomena such as gaseous effects or electromagnetism are not as important in most games, many movies and scientific simulations will require these more complicated simulations. Unlike real-time engines, the main goal of high precision physics engines is to create very accurate results with little concern as to the run-time impacts such precision brings.

Movies are often using these high-precision engines. Obviously computer animated movies rely heavily on physics to create believable environment and action, but real-action movies utilize physics to create realistic special effects as well. In this environment, time is not nearly as much of an issue as it is with the gaming industry. Animators pre-render the movie and effects so engines are able to take more time computing the results. These high-precision engines will take a long time to do the initial calculations, but the results are that recording and playback will be instantaneous at the theatre.

Not surprisingly, the scientific community also relies on physics engines and computer simulations. There are many experiments that scientists cannot reproduce in their labs, but can successfully

reproduce in a virtual lab with the help of a physics engine. Even more so than the engines used for movies, these scientific engines strive to get the best accuracy at all costs and operate to a very high precision [3]. These engines are often very slow and extremely specialized in order to obtain the best possible results. These engines often serve to solve real-life problems such as weather forecasting, air resistance analysis or the performance of tire treads under different weather conditions.

High-precision engines are not often bounded by time, but a decrease in run-time could still be significantly beneficial to employees and scientists who spend part of their day waiting for the results. Real-time engines however, are severely bounded by time and can benefit greatly from a speed boost. By parallelizing the physics engine we are blurring the boundary between high-precision engines and real-time engines. As the run-time decreases, real-time engines can increase their precision and high-precision engines will approach the run-times of their quicker counterparts.

However, due to the complexity and cost issues involved nearly all physics engines are single-threaded so this boundary is still a strong line between the two primary types of physics engines. While this is fine for now, the ever-increasing demand for more realistic physics and the continuing trend of increasing the number of available cores will create higher demand for parallel physics in the near future.

Chapter 2

Physics Engine Basics

Before discussing how we can parallelize an engine, a strong understanding of physics engine basics must be had. There are too many variations on many of these techniques to go into too much detail here, so we just describe the basic ideas behind each subject listed below. A detailed look at how we implement each subject can be seen in chapter 4.

2.1 Specialized Engines

Physics in general is an extremely large and complex subject area and nearly everything we see, interact with or experience is related to physics in some form. Therefore, creating a completely general physics engine is often unrealistic, as the work is near endless [4]. Physics engines are often targeted towards a certain group or simulation. There is a near limitless amount of physics an engine can support, but some of the common ones include the following subjects.

2.1.1 Dimensions

Most physics engines limit themselves to a certain dimension (i.e. 2-D or 3-D). While it may sound easy, adding support for a different dimension often requires a completely different software approach. There are many shortcuts that are necessary depending on the dimension that is being simulated and not specializing the engine for a specific dimension will often yield a slow experience.

2.1.2 Rigid Body vs. Soft Body

Rigid body dynamics assumes that all bodies are infinitely rigid, meaning bodies will not deform when external forces affect them. This greatly reduces the complexity of the engine and creates a faster run time environment. However, this approach based on such an assumption is not realistic, as even the hardest and most rigid materials will deform under sufficient forces. A more realistic alternative would be the soft body engine, which supports the simulation of soft-bodies as well (shown in the simulation of a flexible fabric falling between pillars in **Figure 2**).

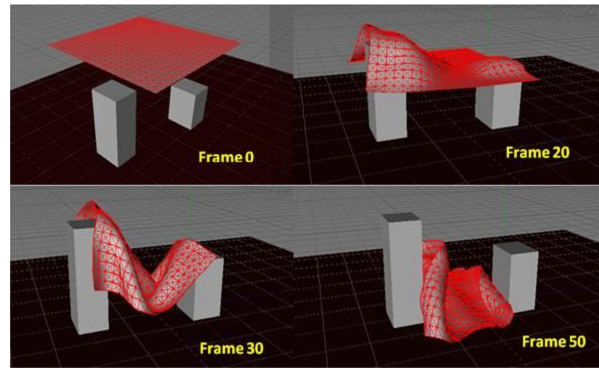

Figure 2. Physics engine and modeling software Cinema4D displaying soft body dynamics.

2.1.3 Fluid dynamics

Some engines will also support fluid and/or gas dynamics. These engines are most commonly used for scientific simulation where fluids and gases are involved, or movies and video games where realistic water or other liquid or gaseous effects are needed.

2.1.4 Electromagnetism

Some engines may support electromagnetic effects. These engines are often seen in video games and movies as light and the physics associated with light and reflections. Basic lighting does not often need an entire physics engine but more complex variations (such as lighting through water, glass, magnifying images, etc) often benefit from a physical simulation.

2.2 Terminology

Some terminology that is specific to physics engines may not be well known. In subsequent sections we will define and refer to these terms to simplify our explanations for the purposes of this paper.

2.2.1 The World

In physics engines, the world refers to the environment that is being simulated. Much like our real physical world, the simulated world is essentially a virtual environment where physical action and reaction explained by the laws of physics occur and take place. The world naturally includes all of the

bodies contained within it and any special properties associated with the world (i.e. acceleration fields such as gravity, etc).

2.2.2 Contact

A contact is generally an object that will fully describe a collision between two bodies. The contact contains all necessary information that is needed to resolve this collision. Depending on the type of collision this can change, but generally a contact will contact two bodies (and all of their attributes such as mass, velocity, etc), a contact point and a contact normal.

2.3 Rigid Body Features

With all of the possible combinations interactions between physical bodies and the properties that act upon and influence them in the real world as explained through the discipline and science of physics, the reality is that the scope of a physics engine that seeks to emulate the real world is nearly limitless. For the sake of simplicity alone, practical decisions must be made to limit the scope of the physics engine design project early on. A physics engine designed for real-time situations (i.e. video games) which assumes that all bodies are infinitely rigid will be assumed from this point forward for the purposes of this paper.

The foregoing assumption significantly simplifies the physics of the engine under examination and study here and correlates with the majority of physics studies and principles that most students are traditionally exposed to in middle school and high school. This includes forces and impulses, drag, gravity, angular and linear velocity, angular and linear acceleration, collision detection and collision response. The details of the actual physics are lengthy and relatively complicated subjects in their own right, so here we will assume a basic knowledge of physics is known and we only point out some special unexpected cases. More information can be found regarding these materials in John Taylor's *Classical Mechanics*[5].

In general, a physics engine has a few phases that it will loop through continuously while performing calculations. Prior to this loop, a physics engine will initialize the simulated world. This usually includes allocating memory that will be used by the simulated world and related objects, setting global variables

and constants such as gravity, and adding any preliminary rigid bodies that will be used immediately upon creation (usually some scenery in the level, such as a floor, walls, etc).

Once initialization is completely the physics loop can begin. What a physics engine does in the loop varies from engine to engine, but in general there is a collision detection phase, collision response phase, integration phase, and cleanup phase. For a detailed list of what such engines do please see chapter 5.

2.3.1 Collision Detection

Collision detection phase consists of determining whether any rigid bodies are colliding with other bodies during this game loop. Collision detection is usually further subdivided into a rough collision phase (also known as broad collision, shown in **Figure 3**) and fine collision phase.

2.3.1.1 Rough Collision Detection

Rough collision detection performs quick checks on rigid bodies to quickly prune the data of bodies that are not colliding. Note that this phase may return a list of objects that are not actually colliding, but will never leave out a body that is colliding (in other words there may be false positives for a collision, but never false negatives). The main goal here is to be as quick as possible to reduce the potential collision list - even if we are not being very accurate.

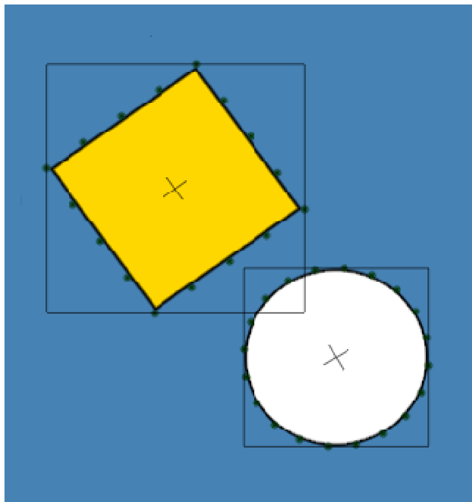

Figure 3. Bounding boxes is common technique used in rough (broad) collision detection. Note this scenario would return true for broad collision despite the objects not actually touching.

2.3.1.2 Fine Collision Detection

After the list is pruned to a more reasonable size, fine collision detection takes place where the engine looks more closely to return a list of guaranteed colliding rigid bodies. This phase obviously has a much larger runtime than rough collision, but it is necessary to determine which bodies need to be resolved in the next phases.

Unlike rough collision detection, the fine collision detection takes into account details such as body shape to guarantee that the two bodies are touching. In general, we are trying to determine whether a certain body contacts a point of another body.

There are many different ways to perform both of these tasks and the details can be quite complex. A full list of potential methods can be found in Christer Ericson's *Real-Time Collision Detection* [7]. How the physics engine is doing this is not entirely important right now. The important thing is that an engine must first do a quick sweep of the bodies to check for collisions, and then a fine check on the bodies. At the end of the collision detection phase we will have a list of bodies that are colliding with one or more other bodies.

2.3.2 Collision Resolution

Once we know which bodies are colliding with one another we can take measures to resolve the collisions. This phase is also generally subdivided into two phases: contact generation and contact resolution.

2.3.2.1 Contact Generation

Contact generation takes our rigid bodies and creates a “contact” out of the collision. A contact is used to fully describe a collision so that it may be resolved at a later time. It is important that we collect all contacts first before resolving them since a body could be involved in multiple collisions in one loop. If we resolve this body too soon, it will incorrectly affect the next contacts. For 2-Dimensions we must know both bodies (and all their attributes such as mass and velocity) in contact, the contact point and the contact normal and any physical coefficients associated with the collision such as friction or restitution.

Contacts will be kept and stored to be resolved at a later time. Resolving a contact is often more complicated than the naïve approach of resolving all contacts in the list one by one. There is an order that contacts should be resolved in and we will not know this order until we have all contacts generated.

2.3.2.2 Contact Resolution

After the contact generation phase we will have a complete list of contacts that fully-describe all collisions in our system. The contact resolution phase will run through each contact that has been generated and resolve this contact. Depending on the type of bodies involved in the collision this could result in a change in velocity and/or angular velocity. In most rigid body collisions, basic Newtonian physics is used, and we determine final velocities through Newtonian concepts such as conservation of momentum, which guarantees that the momentum of a closed system before a collision is equal to the momentum of the system after a collision.

A naïve approach to contact resolution consists of a simple loop that resolves each contact without prejudice. However, this can cause problems if a body has two or more simultaneous contacts. Resolving one contact may affect the other contact. This can create unnecessary work with the resulting impact on

the time required to complete the resolution. To resolve this problem, if we always resolve the most "severe" contact first (the contact with the highest velocities) then we will avoid this issue.

2.3.3 Integration

At this point in the loop, all collisions have been captured and the associated bodies have been assigned new linear and angular velocities. The bodies have not actually moved anywhere on the screen however. We need to now "step" forward in time and integrate our engine. The time we move forward is either a set constant time or a derived time (such as how long it took us to complete the above steps). We then loop through every rigid body in our system and apply the appropriate updates to that body. This includes moving the linear and angular positions of the body (found by multiplying velocity and the time step). Other updates to bodies are done here as well, including updating velocities if that body is in an acceleration field (such as gravity).

2.3.4 Cleanup

Our engine has now completed a complete physics loop. Naïve engines could continue the loop from here, and repeat the above steps indefinitely. However, most engines will usually go through some form of cleanup to ensure efficiency. This cleanup stage can consist of a lot of tricks to speed up the engine. Some examples include putting rigid bodies to sleep (and thus ignoring them in future iterations) if the body has not moved in a while, or removing rigid bodies from the world if they go outside the world's bounds.

CHAPTER 3

Previous Work

Since parallelism in physics engines is still in its infancy there are not a lot of examples of commercial quality parallel engines. Most attempts at parallelizing physics are academic papers consisting of theory or specialized examples. There is much to learn from these academic approaches that will surely be used more often in coming years, but there are also some pitfalls where theory does not always translate to a practical approach. It is also useful to consider current single-threaded engines because there are techniques that can be utilized in both types of engines. Each engine mentioned below features a unique approach to solve the challenges we faced in implementing our own engine.

3.1 Box2D

Box2D is an open source physics engine, specializing in simulating rigid bodies in 2-dimensions [1]. It was originally developed by Erin Catto in C++, but has since been expanded to multiple platforms and languages. Box2D does many things fantastically, however the main takeaway I will focus on is Box2D's simplification of differential equations.

Box2D uses a numerical approximation to reduce the complexity of integration. While this method is not entirely physically accurate it will yield results that are good enough to be undetectable by human eyes during video game play. Collision resolution and integration introduce differential equations into our physics engine that must be solved. Physics engines are often required to solve hundreds of these equations per cycle, which can lead to massive slowdowns without approximations. These non-linear and multi-variable problems are the best candidates for approximation via numerical integration.

Numerical integration is achieved by starting with the first order form of our equation. Consider a baseball traveling through the air. In this scenario we have our velocity equation $dx/dt = f(t, x)$ and our initial velocity $x(0) = x0$. Assuming that $f(t,x)$ is some non-linear function, we can approximate the velocity with a linear function $[x(t + h) - x(t)] / h$. Rearranging terms, we have the approximation: $x(t + h) = x(t) + h\, f(t, x(t))$.

This formula allows us to obtain a reasonable approximation for some time step *h*, and get the position *x* at any time, with a much smaller CPU load. Of course the larger h is, the larger our uncertainties are. This is not a significant concern however, because Box2D is recalculating *x(t+h)* every game loop so h is relatively small.

Although Box2D is a single-threaded solution many of the techniques utilized can be utilized in a parallel engine. Box2D is an expansive project and contains far more information than what is outlined above. However our engine will differ in most other aspects so we will keep the description brief.

3.2 *pe*

Klaus Iglberger and Ulrich Rude of the University of Erlangen-Nuremberg in Germany arguably demonstrate the best academic approach to parallel physics. The paper, entitled **Massively Parallel Rigid Multi-Body Dynamics**, takes the idea of parallel physics to a massive scale [12]. Iglberger et al, develop their own physics engine (called *pe*) to be used for massive parallelization of rigid bodies. *pe* is a framework written in C++ and has been under development since 2006.

As previously mentioned, Iglberger-Rude also point out that the current implementations of physics engines come in two limitations. This includes engines that strive for accuracy but are using polynomial time. These engines will require massive computing power with additional objects and are limited to a few thousands concurrent objects. The other variety, used mostly for games, forgoes accuracy for speed, but these are also limited to a few thousand concurrent objects due to time restraints in games. Iglberger-Rude attempt to solve this problem and they have developed a quick but precise engine through parallelization. *pe* is designed for massive cluster super-computers that utilize MPI (Message Passing Interface) to communicate between CPUs.

The Iglberger-Rude algorithm first will assign each rigid body to a specific processor based on the body's location. Because rigid bodies are not generally point masses it is important to keep the entire rigid body on the same MPI for the entire time to avoid discrepancies; since they partition their processes in space (a rigid body could be on the boundary and scale two processes). To avoid this, the center of mass was used as a pinpoint on the location of the body (this guarantees each body belongs to one and only one process). Each process checks the center of mass and if it resides inside that process's bounds then it will

be responsible for carrying out actions on that body. Each process has no knowledge of any other objects that are outside its boundary. If a body is overlapping two processes, both processes must work with that body (via MPI) because there could be forces acting on that body from both processes.

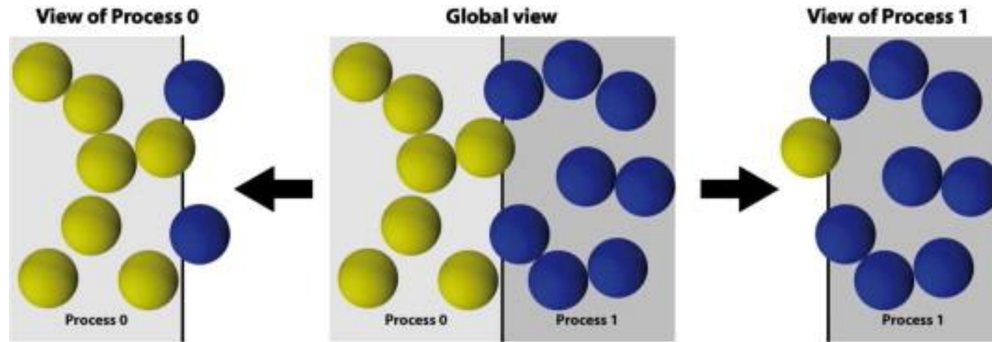

Figure 4. Process view for *pe.* Each process is only aware of bodies that belong to that process and bodies that are overlapping its bounds.

MPI processes send data to each other to stay in-sync. Obviously, for performance and complexity reasons, the number of MPI messages sent and received should be kept to a minimum, which can be problematic for a large scale rigid body simulation. Therefore, for efficient parallelization *pe* sends a single MPI message to a remote process in every communication step. Since this requires multiple data structures to be sent in each MPI message, they send raw bytes and have each process reconstruct the structures from the byte stream. A process cannot recognize beforehand if an incoming message is expected, even if no data is needed to be sent, so they still need to send an empty message as the process must be waiting every step for this to work properly.

The parallel algorithm is based on the “fast frictional dynamics” solver that was presented in a previous paper by Kaufman et al [16]. This FFD solver is a fast real time collision response algorithm. The benefit of this FFD algorithm is the fact that collisions are treated locally – that is post-collision velocities for colliding rigid bodies are only calculated based off the states of the colliding rigid bodies. This means that no LCPs (linear complementarily problems) that many physics engines would need to solve are used. The FFD runs in O(n + c) time, where n is the number of rigid bodies and c is the contact points between bodies.

FFD consists of three main steps. Step 1 (after MPI force synchronization) has every rigid body take a half step of its velocity. A simple for loop of all rigid bodies that belong to each Parallel process is run to perform this step.

Step 2 (after a MPI update of remote and notification of new rigid bodies) first checks all contact points for all rigid bodies. Another for loop for each process checks all rigid bodies that belong to said process for contact points. If a contact point is found, rotation and constraints are calculated.

Step 3 (after a MPI exchanging constraints on rigid bodies) takes a further look at constraints violated in step 2. For all rigid bodies that had violations, post-collision velocity is calculated, friction response is taken into account and the resulting new velocity is added to the rigid body position. If no constraints were violated, then the original velocity is simply added again in another half step to complete the full step velocity. Finally, another MPI is sent updating remote and notification of new rigid bodies before the process repeats at the force sync and step 1.

Of course one anticipated problem is that a collision could take place on a boundary. Since each process only recognizes rigid bodies which have their center of mass within the process's boundaries, it is possible that a collision is taking place without the local process recognizing it (consider a ball, whose center of mass is within the bounds, but the outer radius has a contact point outside of the bounds with a rigid body that is fully contained in the remote process bounds). The remote process will recognize both rigid bodies, since the ball has points within its bounds, so the remote process must make the constraint calculations. But since the remote process does not control the ball, the constraints will then be sent via MPI to the local process for step 3.

Additionally, when a rigid body leaves the boundaries of the local process the local process must inform the remote process that it crossed into the event. This works as follows: When the rigid body is fully contained in the local process, the remote process knows nothing about it. When the overlap begins, the local process will send an MPI to the remote process of its state. When the center of mass crosses the bounds then the remote process will take ownership of the rigid body (but local process will still retain knowledge of its state). Finally, when the rigid body fully exits the local bounds, the local process will cease recognizing the rigid body.

This algorithm was designed for massive cluster computing systems. It was tested on the HLRB-II supercomputer, a computer consisting of 9,728 dual core processors, which could simulate approximately 1.4 billion rigid bodies, something completely unachievable with a single core processor.

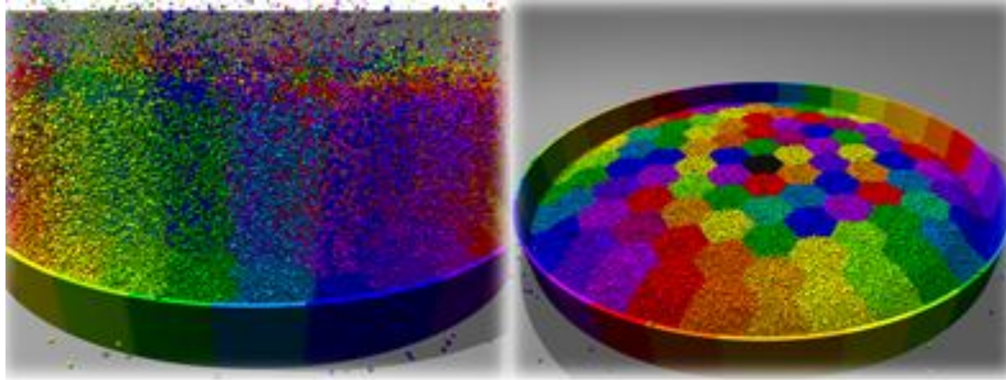

Figure 5. *pe* simulating 500,000 rigid bodies without trouble. Each color is a separate MPI process.

Obviously, with the use of such a powerful computer is not a practical approach for video games, but the same techniques can be scaled down to more modest platforms.

3.3 Nvidia's PhysX

Nvidia has arguably developed the most sophisticated parallel physics engine in use today [2]. PhysX supports both 2d and 3d environments, GPU accelerated physics, and works on multiple platforms. Unfortunately, since it is a closed source, we must rely on information released by Nvidia and cannot really look into the details of many of its features.

PhysX is designed for a large number of processing cores. While most practical setups are generally limited to about 2-4 cores on the CPU, PhysX bypasses this restriction by offloading most of the work to the GPU, which tend to have hundreds of processing cores that PhysX can take advantage of. PhysX will run on the CPU if no GPU is available, but because of its highly scalable design PhysX greatly benefits from GPU calculations.

How exactly Nvidia does this is not entirely clear. On some GPUs with a dedicated Nvidia Physics accelerator, Nvidia utilizes partitioned fundamental tasks in hardware, with a control processor, a processor for data movement (moving data in and out of memory) and a processor for floating point calculations. This partitioning is implicit hardware acceleration and would be relatively invisible to software.

However, in recent years Nvidia has moved away from the physics accelerator and instead works more with the standard GPU itself. The details of this are unknown, but software techniques as described above are almost assuredly being used. Regardless of how Nvidia performs the task, Nvidia is dividing the work between multiple processors to distribute and even out the load (via data partitioning for the most part to get the most out of the numerous amount of cores a GPU offers).

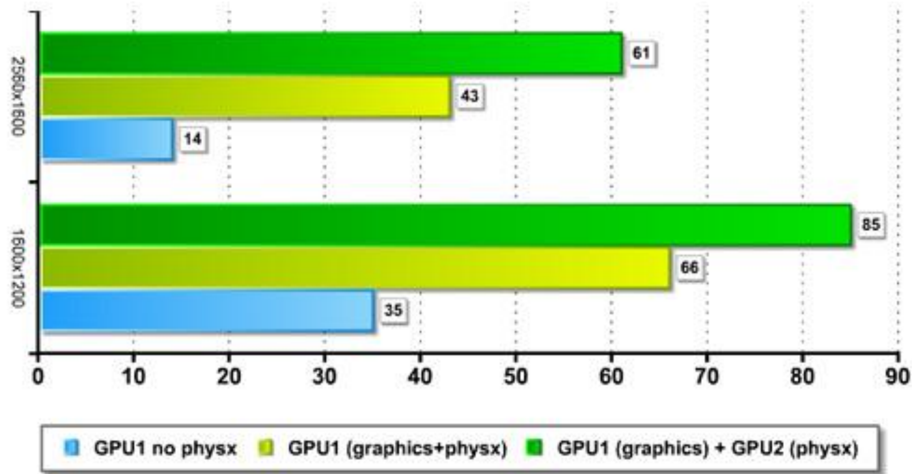

Figure 6. Nvidia PhysX displaying roughly double frame rate gains over non-PhysX environments and even further gains with a dedicated GPU for physics.

CHAPTER 4

Design and Implementation

With a firm understanding of physics engine basics in hand, we can now begin to examine how we created our own engine. Unlike some of the above-described implementations, the main goal of our engine as created and described below is to provide not only a parallel approach but also a practical one.

In brief we have implemented a parallel rigid-body dynamics physics engine. We wrote the project in Java, from scratch, so that it may be used in android environments. The solution below is intended for devices that are hardware limited, so no GPU acceleration is used as many phones do not have this luxury. Instead we develop the engine to only use four threads so that it is optimized for realistic CPUs seen on phones, which mostly consist of dual and quad-core chips.

Since the engine is a generic approach that does not actually solve anything by itself, we also created multiple test scenarios to test and stress the engine. Additionally we created two mobile games that showcase the engine to give readers some potential ideas of what our engine can be used for.

4.1 Rigid Body Dynamics

Rigid bodies are objects that we will assume for purposes of this project do not compress under forces. Strictly speaking, all rigid bodies will compress and deform if subjected to sufficient compressional forces. For instance, a steel ball will compress under forces, but the compression is so slight that the added precision we would attain from calculating this compression does not outweigh the extra computation time required to do this. Materials like rubber will compress under forces, but the de-compression is generally too fast for human eyes to detect when the rubber body is undergoing numerous collisions (for example, a bouncing a rubber ball). Objects like this also have a strong tendency to return to their original position. Because of this tendency and unnoticeable effects of compression we can approximate collisions involving even elastic, squishy materials like rubber. Bodies that do not have this resilience tendency to return to their original shapes and positions (like a ball of putty) are not good candidates for rigid body modeling because even small forces will permanently affect and alter the shape of the body.

The assumption above helps us reduce the computation time required during collisions. We can now use basic laws of physics and not worry about the more CPU intensive task of calculating body shapes after collisions have occurred.

Listed below are some important attributes associated with rigid bodies that we must update throughout the physics loop. In software engineering terms this means that each rigid body object we create will require some memory allocation to store the relevant information and we must update that information if and when it changes throughout the physics loop.

4.1.1 Mass

Mass is defined as the property of a body that gives rise to the phenomena of the body's resistance to being accelerated by a force and the strength of gravitational attraction between bodies. Our engine will ignore the latter as the effects are negligible for all but extremely massive bodies. However, a body's mass does play a large role in determining the body's dynamics due to the resistance to acceleration attribute of the body. Generally speaking, the more massive an object is, the more resistant it is to velocity changes.

Note that in software it is often more convenient to denote the mass of an object as the inverse mass (1 / mass). This makes it much easier when dealing with infinite mass bodies, as the inverse mass of such large bodies is zero. This simplifies rigid body dynamic equations and makes it easier to store a single digit integer rather than an infinitely large number. Setting the mass of an object to infinity will result in an immovable object.

4.1.2 Vertices

A vertex is a point that lies on the boundary of our object. We use vertices to describe the shape of an object by specifying a location at each corner of the polygon. Vertices are necessary for rendering the object to the screen as well as determining collision responses in future steps. If our object happens to be a circle, which would contain infinite vertices, we simplify the specification of vertices by utilizing the radius of the object.

When creating polygons we always assume that the vertices are listed in counter-clockwise order. This allows us to quickly find the normal associated with each edge, a very important quantity that will be required in collision response. A normal is the unit vector that points outward from the body's center at a right angle from the edge. If our body's vertices are given in counter-clockwise order, a 270 degree rotation of the edge will give the direction of the normal.

4.1.3 State

A state is a complete description of a body that would be needed to render this object in space. In 2d rigid body dynamics this takes the form of a 2-dimensional position vector and a scalar indicating orientation. This is simplified into a 3-dimensional vector consisting of $<x, y, \theta>$, where x is the center's position along the x-axis, y is the center's position along the y-axis, and θ is the orientation of the object, in radians, assuming the x axis is 0 and positive is a counter clockwise rotation around the origin.

The state also conveniently stores the center of mass of the object as well. In rigid bodies a center of mass will be unchanging (because the body itself cannot change). Additionally, we assume that our rigid bodies have a uniform density. This guarantees that our centroid is also our center of mass, which saves a bit of memory.

4.1.4 Linear Velocity

Linear velocity describes how fast an object is moving. Velocity is the rate of change of the object's position, a vector form of the object's speed. An object's velocity determines how far we move the object on screen during each loop. Velocity can be calculated as the time derivative of the object's position, that is **$v = dx/dt$.**

More often in physics engines we will be going the other way, that is determining the position changes based on an objects velocity, by multiplying the velocity by the change in time.

4.1.5 Angular Velocity

Similar to linear velocity, angular velocity describes how fast an object is rotating. Angular velocity can be calculated in the same way, by determining the starting and ending orientations and the time it took, that is $w = d\theta/dt$.

4.1.6 Linear Acceleration

Linear acceleration is the rate at which a body's linear velocity changes with time. This does not directly affect the object's position in each game loop, but rather directly affects the object's velocity. That is $v_f = v_i + a * t$, where v_f is the final velocity, v_i is the initial velocity, a is the object's acceleration, and t is the time of the step we took. Linear acceleration in a physics engine is usually represented by gravity, which applies a constant linear acceleration field to all objects that are affected by it.

4.1.7 Angular Acceleration

Much like linear acceleration, angular acceleration is the rate at which a body's angular velocity changes with time. As with linear acceleration, in each game loop we will be adding an object's angular acceleration to its angular velocity, that is $w_f = w_i + \alpha * t$, where w_f is the object's resulting angular velocity, w_i is the initial angular velocity, α is the object's angular acceleration and t is the time duration.

4.1.8 Moment of Inertia

A body's moment of inertia describes how the body is affected by rotational changes. Like mass is a measurement of resistance to change, the moment of inertia can be thought of as a rotational mass. Calculating the moment of inertia can be a CPU intensive task. For our engine we use some approximations in order to speed the process up. For circular objects, a body's moment is $I = (\pi / 4) * r^4$. For polygons, we assume a rectangular approximation $Ip = (b * h^3) / 12$, where b is the object's base along the x axis and h is the object's height along the y axis [17].

4.1.9 Linear Momentum

Linear momentum is the product of the object's mass and velocity (***p = m * v***). In our engine this is a derived quantity, meaning that we do not directly store it with the rigid body since we do not directly use this quantity, but utilize momentum in deriving our final velocities after a collision.

4.1.10 Angular Momentum

Angular momentum is the product of an object's moment of inertia and its angular velocity (***L = I * w***). Again this is a derived quantity in the engine.

4.2 Collision Detection

Collision detection has a variety of possible implementations. A very naïve approach can simply check each vertex of a rigid body against every other rigid body, looking for containment. Obviously this would be a quite slow process and the complexity would be $O(n^2)$. As our world contains more and more bodies this approach would grow horrendously sluggish. Utilizing a few tricks, we can make our engine run much quicker. Below is the implementation we chose to use but it is certainly not an exhaustive approach to collision detection as outlined by *Real-Time Collision Detection* [7].

4.2.1 Discrete Collision Detection

Our physics engine is a discrete collision detection engine. This means that collision detection is performed on a discrete basis at the instant of our check. If two bodies are penetrating one another while a check occurs, those bodies are said to be colliding. If two bodies are not penetrating at the time of the check then the bodies are not colliding.

This can create problems when rigid bodies are moving *extremely* fast such that the bodies move completely through each other without the collision check detecting it. However, for all but the fastest objects, performing a continuous collision detection to catch these cases will be unnecessary overhead so we leave it for future work.

The discrete collision detection approach introduces the problem of penetration. As we integrate bodies in time it is highly unlikely that the bodies just touch each other as would happen in the real world. More likely is the scenario where two bodies will become overlapped in one another and some penetration will occur. Since two rigid bodies are not supposed to be penetrating one another we need to resolve this penetration prior to resolving the physics associated with the collision. To resolve the penetration we want to find the *Minimum Translational Distance*, MTD, which is the minimum distance required to push these two objects apart so that they are just touching at the point of contact, as shown in **Figure 7**.

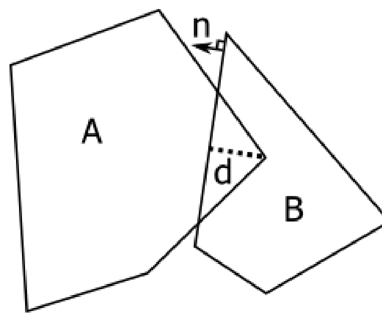

Figure 7. Two objects penetrating one another by the distance d.

Because our bodies should not technically be allowed to penetrate in the first place, it is important we find the minimum distance to reduce as much error as possible by resolving collisions. Again, for all but the fastest rigid bodies, the error introduced by penetrations is extremely small and unnoticeable by humans. As a result, the speed increase we gain from doing discrete collision detection outweighs the small error introduced for our targeted environment.

4.2.2 Broad Phase

Mentioned previously, broad collision detection is a phase of collision detection that is often implemented in physics engines. The main goal of broad collision detection is to provide a quick prune of the data to remove objects that are definitely not colliding.

4.2.2.1 Quad-Tree

A quad-tree is a tree data structure where each node has four children (similar to the more popular binary-tree, but four children per node instead of two). A regional quad-tree is a special type of quad-

tree where the partitions are according to geometry. Each section of the world is subdivided into four smaller sections. Each subdivision is represented by a separate node on the quad-tree and each node on the quad-tree has a list of bodies that is associated with that section as shown in **Figure 8**.

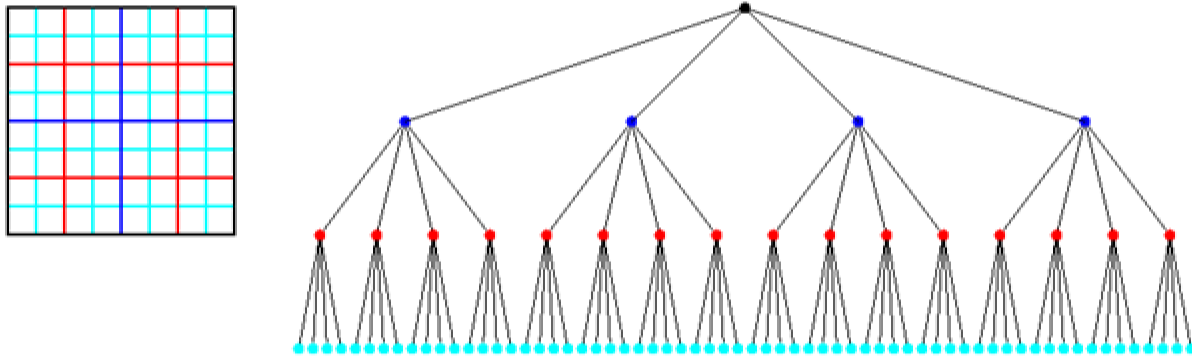

Figure 8. A regional quad-tree showing the nodes corresponding to regions on the space to the right.

When inserting a rigid body into the tree we first check the body's center and find the leaf that that the body's center belongs to. Next we check if the leaf's boundary can fully contain the rigid body. If it can, we insert it into that node. However, if the leaf's boundary is too small to completely contain the body we move to its parent and perform the same check. This is repeated until a node is found or when we arrive at the root node. This algorithm guarantees that each rigid body will be placed in the smallest region possible.

This tree not only provides a data structure to house our bodies in, it also allows us to utilize this information in the broad phase. The regional quad-tree ensures that an object can only be colliding with bodies on the same node or on any parent node of that node. Because every node fully contains the body, it is impossible for two bodies on nodes that are not hierarchically related to be colliding. In some way this is similar to having a group bounding box of sorts, which can easily check when objects are touching. Objects in different group's bounding boxes cannot be intersecting.

Therefore, for the broad phase, for any rigid body **A** we can quickly receive a list of possible collisions, **L**. We first call quad-tree's retrieve function on **A** to get **A**'s node **N.** Once retrieved, we create a list of **N**'s bodies and add all of **N**'s parenting nodes bodies to the list. Note there is no need to check **A** against children nodes of **N** because that check will occur when we look for collisions on the children node of **N**,

if there are any. There is also no need to check **A** against neighboring nodes or variations of neighboring nodes such as an “uncle” node because it is geometrically impossible for **A** to be in collision with a body that lies outside, or does not contain, **N**’s boundary.

Every broad phase technique has its own benefits and downfalls. A particular downfall of the quad-tree approach shown here is that the exact trimming that the quad-tree provides depends largely on the geometric makeup of the bodies. If bodies are clumped within one corner, and therefore one node, this trimming will not help us as we will be essentially comparing every body against every other body like in the naïve case. We will also run into the same problem if the bodies are too large to be contained in the leaf nodes and are all forced to be contained at the root node.

However in the vast majority of video games it is a safe assumption to expect bodies to be relatively spread out amongst the world. It is also safe to assume that the bodies within the tree will generally be much smaller than the entire world. With these assumptions, a quad-tree made the most sense to implement in our physics engine. This is not a one-size-fits-all approach however, and as always, it is important to choose the correct tools for the job.

4.2.2.2 Bounding Box

In addition to our quad-tree, we also make use of bounding boxes to assist our broad phase detection. A bounding box is the smallest possible rectangle that contains every point of our rigid body. Every rigid body in our system is assigned a bounding box. This requires slightly more memory for each rigid body but the reduction in calculation times makes it worth the additional memory cost.

Bounding boxes allow for crude but efficient checks for collision detection. If two bounding boxes are not colliding, then the objects are certainly not colliding since the objects are fully contained in the boxes. Assuming we have two bounding boxes for our objects that we wish to check, **RectA** and **RectB,** we can perform the check using simple pseudo code below:

```
if     (RectA.leftX < RectB.rightX && RectA.rightX > RectB.leftX &&
       RectA.leftY < RectB.rightY && RectA.rightY > RectB.leftY)
       collision = true
```

Our engine utilizes axis aligned bounding boxes, meaning that the box is un-rotatable and always aligned along the x-axis. This makes overlap checking a simpler process when all boxes are aligned in the same direction.

For circles, an axis aligned bounding box is simply a square with height and width equal to the diameter of the circle. For more complex polygons, a simple loop through the polygons can be done checking for the min and max values for width and height.

4.2.3 Fine Phase

If our broad phase pruning above still shows that we have a collision then we must look at our bodies in closer (and thus slower) detail. The fine collision steps will need to be looked at on a case-by-case basis depending on the geometries involved as the methods and shortcuts we can use vary depending on the bodies involved in the collision.

4.2.3.1 Circle vs. Circle

Two circles colliding against each other is the simplest and quickest scenario. To check for penetration we can check the distance between the circle's two centers. If the distance between the two centers, **AB**, is smaller than the sum of the radii, $\mathbf{r_A + r_B}$, then we have a collision. In the code it is actually more efficient to calculate the square of this value, as taking the distance squared is easier for the CPU by avoiding the square root, so the check becomes:

```
if (AB^2 < (r_A + r_B)^2)
      collision = true
else
      collision = false
```

Once we know we have a collision we need to gather information so that our contact solver can resolve this collision at a later time. As stated previously, the information that the solver needs for 2d rigid body collisions are the two rigid bodies, the contact normal, and the contact point. Additionally our solver will

need the penetration between the objects as it is extremely likely that the two objects have moved inside one another during the last integration phase.

The penetration is simply the difference of the value calculated above. The sum of the two object's radii is subtracted from the distance between the two centers, **AB**. The contact normal for any two circular objects is always the unit vector in the direction of the difference of the two centers, **AB**. The contact point is the point where either circle intersects with the vector **AB**.

4.2.3.2 Circle vs. Polygon

A collision between a circle and a polygon (and vice versa, of course) requires a bit more computation than the above case.

To find if these types of objects are penetrating one another we find the closest point on the polygon to the circle's center. To do this, we loop through every edge that belongs to the polygon and check the distance to the circle's center. The edge that has the smallest distance to the circle must be the edge that is in contact. The contact normal will be this edge's normal and the contact point is the point on the circle that is in the direction of the normal.

Once we have the potential contact edge we can find the penetration. Consider the distance between the circle's center and the contact point, **CP.** The penetration will be the difference between **CP** and the radius of the circle. If the radius of the circle is larger than this distance, then we will be calculating the penetration based on the difference between these two values. If the circle to contact point distance is larger than the radius of the circle, then the objects are not actually colliding and we can exit early as shown below.

```
CP = Circle.Center – ContactPoint
penetration = CP – rc
if (penetration > 0)
      collision = false
```

4.2.3.3 Polygon vs. Polygon

Two polygons colliding is the most complicated case to handle of the three. Like everything in physics engine creation, there are multiple ways to go about checking this. A popular approach is the Separating Axis Test [13]. However, implemented below is the Minkowski Difference as it was slightly easier to conceptualize.

The Minkowski Difference, also known as the Minkowski Sum, is a quick way to determine body penetration by essentially merging the polygons into one [18]. Consider rigid bodies **A** and **B**. To calculate the Minkowski difference we take each edge of **B** and add it to **A**. The result is we are shrinking the volume **B** and adding it to **A**'s geometry. The picture in **Figure 9** provides an easier way to conceptualize this information.

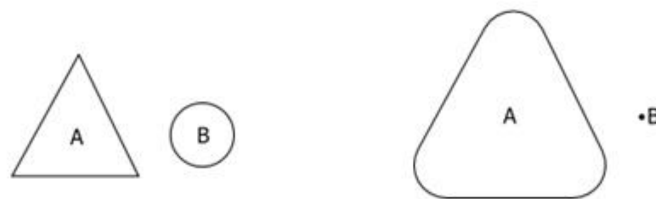

Figure 9. Two polygons on the left and their Minkowski Difference on the right.

Adding the vectors of **A** and **B** will change the shape and location of the new polygon of course. The interesting part about this is that when our new shape contains the origin (0, 0) our two original bodies **A** and **B** will be intersecting. The distance between the origin and the closest point on our Minkowski polygon will also be the penetration.

To create the Minkowski Difference we need to create a new rigid body, **MD**. Each edge of **MD** will be made up of a vertex from one body subtracted by an edge from the opposing body. Our supporting vertices determine which vertex we will be subtracting from which edge. A supporting vertex is found mathematically by determining the maximum dot product a vertex has with a given edge vector.

Therefore, we loop through every edge of each body. For every edge we find the supporting vertex on the opposing body and subtract the values to receive a new edge of our body **MD**. We can then project

the origin onto this new edge and determine the penetration. If we find a positive distance we can exit early as this means our bodies are not in contact. However, if no positive distance is found, we take the smallest of the negative values found for penetration.

This process is quite expensive and is expected to run in $O(E_a * V_b + E_b * V_a)$ where E are the edges of the rigid body **A** or **B** and V are the vertices. However, this is a necessary evil as we must perform these fine checks in order to carry out collision response. With the much quicker broad phase and with early out checks for positive penetration, the Minkowski Difference is only fully calculated when necessary.

In the pseudo code below, we forego the actual creation of the Minkowski Difference and only check it with one edge at a time. The idea is the same as creating the entire Minkowski Difference object, but it allows us to save memory and exit early without creating the entire polygon if we find a positive penetration. Here are the checks for every vertex in a body **a**, and the same would need to be repeated for the opposing body.

```
foreach vertex in a
      v0 = vertex
      v1 = vertex.next
      normal = getNormal(v0, v1)
      supVertex = getSupportVertex(b, vertex.normal)
      minkowskiP0 = supVertex – v0
      minkowskiP1 = supVertex – v1
      dist = ZERO_VECTOR.projectOntoEdge(minkowskiP0, minkowskiP1)
      if (dist.magnitude > 0)
            collision = false
      else if (dist > smallestPenetration)
            smallestPenetration = dist.magnitude
            contactNormal = normal
            contactPoint = supVertex
```

4.3 Collision Response

At this point in the loop we have enough information to resolve any collisions that were found above. We rely mostly on the law of conservation of momentum to solve the problem.

4.3.1 Penetration Resolution

Prior to resolving the contact with physics, it is important to first resolve the penetration. Our bodies, being infinitely rigid, are not physically allowed to be penetrating one another. We must resolve this symptom of discrete collision detection before we can look at the physical aspect of the collision.

We cannot just move the bodies apart as we must consider the masses of the two objects. Just as it is unrealistic for a small pebble to be able to move a giant rock by bouncing against it, we cannot allow our small rigid bodies to move much heavier objects through penetration. Therefore to resolve a penetration, a ratio of the masses will be used to move each body accordingly. That is, the distance moved by each object is calculated as follows:

$$d = \frac{m}{(m + m0)} p$$

Where *d* is the distance moved, *m* is the mass of the object, *m0* is the mass of the opposing object and *p* is the penetration found previously.

4.3.2 Contact Resolution

Now that our collision is physically correct we can resolve the contact. We do this through the use of impulses. While forces are applied over some time period, an impulse is the value of a force multiplied by the time exerted. It is an instantaneous look at what the force did over its lifetime.

In physics engines it is much simpler to work with impulses rather than forces because force is applied over a period of time. We cannot simply instantaneously change an object's velocity through forces like we would want to do when resolving a contact. By utilizing impulses instead, we are implicitly able to instantly change our body's velocities without having to worry about maintaining forces for multiple loops. Impulses come in the form of linear impulses (for moving bodies linearly) and rotational impulses (for rotating bodies about its center of mass).

First, to find a body's linear impulse, we can simplify the equations by changing our frame of reference. Despite the frame of reference we are in, the outcome of the equations will be the same. Therefore we can consider the equation as if one body was standing still and the other body was traveling towards it at the relative velocity, $\mathbf{V_b} - \mathbf{V_a}$, between the two bodies. We can also rotate so that our reference frame's x-axis is aligned with the contact normal by performing the dot product between our relative velocity and our contact normal. We can now apply the law of conservation of momentum:

$$p_{total} = p_{ai} + p_{bi} = p_{af} + p_{bf}$$

where $\boldsymbol{p}$ is momentum as described in section 4.1. Therefore, our final velocities will be a ratio of our initial velocities along our contact normal.

The final ingredient in our linear velocity resolution is the coefficient of restitution. The coefficient of restitution is a fractional value of the energy before and after an impact. We do not have perfectly elastic collisions in the real world and all collisions result in some lost energy. How much energy lost is dependent upon the materials involved in the collision. For example, a rubber ball tends to retain much more of its total energy in a collision as compared to that of a steel ball.

In reality this coefficient is measured empirically by comparing velocities before and after a collision. However, when simulating, we cannot make this comparison as we are attempting to solve for the object's final velocities. As a result, it is left to the programmers' discretion to assign coefficients of restitutions to their rigid bodies based on the types of materials they are trying to simulate. A coefficient of one is a perfectly elastic collision in which bodies will retain 100% of their energies, while a coefficient of zero will result in both bodies losing all of their energy.

Putting this all together, our impulse becomes:

$$j = -(1 + restitution) * Vrel * (Ma + Mb)$$

And we can apply this to our initial velocities to find our new final velocity:

$$Vaf = Vai - \frac{j}{Ma}$$

$$Vbf = Vbi + \frac{j}{Mb}$$

We can also apply friction at this step. Frictional forces will always be in the direction tangent to our contact normal and against the object's velocity. Like the coefficient of restitution, friction has a coefficient dependent upon the materials in contact. Again, this coefficient must be provided to the engine as it is normally found empirically. A very high coefficient provides very strong frictional forces (such as two rubber materials) while a low coefficient imposes small frictional forces (such as ice).

Now that the linear components of the collision are resolved, we can look to the angular components. Instead of impulsive forces, impulsive torques will be applied to rotate a body instantaneously. By definition, torque is dependent upon the force being applied and the radius at which it is applied:

$$t = (ObjectCenter - PointOfForce) x\ Force$$

Therefore our resulting angular velocities for any particular collision are:

$$Wa \mathrel{+}= (CenterA - ContactPoint) x\ (ContactNormal * j)$$

$$Wb \mathrel{-}= (CenterB - ContactPoint) x\ (ContactNormal * j)$$

Both bodies have now been completely resolved for their angular and linear components. We repeat this contact resolution process until all of our contacts have been resolved.

4.4 Integration

While our bodies have had their velocities altered in the above steps, they have not yet physically moved. We need to integrate our engine forward in time. The time-step that we move forward by can be a static time, but in our case we will time the above steps and use that instead:

```
begTime = time
collisionDetection()
collisionResponse()
timeStep = time - begTime
integrateAllBodies(timeStep)
```

For integration we update each rigid body within our simulated world. As described in section 4.1**,** linear acceleration and angular accelerations will be added to our linear and angular velocities respectively. The position of the body is then moved forward by the linear velocity multiplied by the time step. The body's orientation is moved forward by the angular velocity multiplied by the time step.

The physics engine has now completed an entire game loop, and the entire process can repeat indefinitely – detect collisions, resolve the collisions, update the bodies.

4.5 Parallelism

With the implementation of a rigid body physics engine as described above, we can begin to look at ways to parallelize the engine. Parallelization can usually take two main routes in software, data partitioning or task partitioning. Data partitioning consists of separating the data into small subsections, having every processor perform the required tasks on this smaller subset of data. Task partitioning will keep the data un-partitioned, but each processor will have one specific task that it will perform on the data.

4.5.1 Task Partitioning

Task partitioning in physics engines can be quite difficult and usually not worth the effort. As observed above, a physics engine has multiple steps that all rely on the previous step. We cannot resolve our contacts before we generate the contacts and we cannot integrate our rigid bodies before we know

their resulting velocities from the collisions. In most steps the engine cannot proceed with future steps until the previous step is completed and future steps cannot begin until they are needed. In general, this causes physics engines to perform poorly under a task partitioning scheme. Task partitioned designs will often lead to threads waiting for previous steps to complete, performing their task, and again pausing to await their turn.

4.5.2 Data Partitioning

Data partitioning on the other hand is a much better candidate for software parallelism [10]. Our data, the rigid bodies, can be divided amongst threads and the threads can operate as miniature worlds. The world would now be made up of smaller subworlds, with each subworld working in parallel on a smaller subset of the data. Each subworld would perform the basic physics loop tasks as outlined in the above sections. This solution provides a highly scalable environment as most physics steps are high run-time complexity. Therefore, reducing the number of bodies can have exponential decreases on run-time in many scenarios.

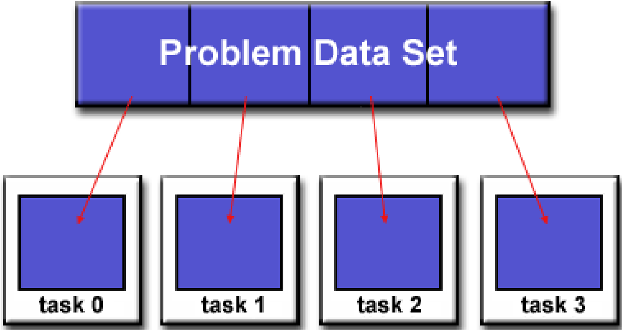

Figure 10. Data partitioning used to parallelize tasks.

4.5.3 Pitfalls

While data partitioning will decrease run-time through decreasing the number of bodies for each thread, it does introduce some subtle problems that need to be specifically addressed. The most important of these is synchronization. If every thread is operating at different speeds and with a different number of bodies it is very easy for one or two threads to lag behind the rest.

This can be remedied by forcing syncs between threads at some point in the loop (such as after the integration phase). However, this will cause the majority of threads to be in an idle state while they wait for the slowest thread to finish. This may not be a problem if the engine must already wait at this point. Many game engines will run a single physics loop and then perform other necessary actions such as drawing or sound. But if we are seeking maximum CPU utilization by the physics engine, this sync solution is not the best idea.

An alternative approach would be to allow threads to continue their loops regardless of synchronization, but force each loop to calculate their own time steps forward. For instance, if thread **A** is completing the physics loop extremely quickly, it will use a much smaller Δtime for its computations than thread **B**, which was lagging behind. While both threads will be running at very different speeds, it will appear fluid on the screen because thread **A** will be taking much smaller time steps. If the threads have vastly different loads then this may become noticeable to the users, as **A**'s bodies will be quite smooth in movement whereas **B** will display some erratic behavior exhibiting jerky movements.

Another subtle problem occurs when two bodies from different threads collide. If both rigid bodies belong to the same thread, the contact generation and resolution is exactly the same as the single-threaded case, but when the two bodies are of different threads we need to have some rules to prevent weird things from happening.

Since we will be comparing two bodies on two different threads we need to ensure that the bodies will not be modified in the middle of contact generation. If a thread modified a body while another thread was in the middle of contact generation with that same body we would get inconsistent results. There also needs to be a hierarchy when resolving contacts. Because each contact only needs to be resolved once, it's important that two threads don't resolve the same contact during the same loop, as you would double the normal effects.

4.5.4 Parallel Implementation

We introduce parallelism to our engine using a data partitioning scheme. Data partitioning allows for significant speedup gains by decreasing the number of bodies each world is responsible for.

There are multiple ways to partition the bodies, but like Igleberger et all [12], we decided to utilize a geometric partition due to its simplicity. In this way, each thread will have a boundary of influence and any rigid bodies whose center lies within this boundary will belong to the thread. Threads will be responsible for detecting collisions on their bodies, resolving these collisions and integrating all rigid bodies that belong to them. A master world will be a mediator between these sub-worlds that will perform actions such as synchronize the threads and take part in inserting bodies to threads.

4.5.4.1 Parallel Quad-Trees

The data partition solution fits nicely into our already implemented quad-tree data structure. We can have one single data structure for our entire world that is shared between all threads. This not only makes it much easier to implement but it also makes it easier to share and swap data between threads.

To do this, our quad-tree describe above will be implemented in the same way except it will be organized in such a way where each sub-world has its own sub-tree. That is, our root node becomes a shared node between all threads, belonging to the master world, and each child of the root node represents the boundary of each thread, as shown in **Figure 11**.

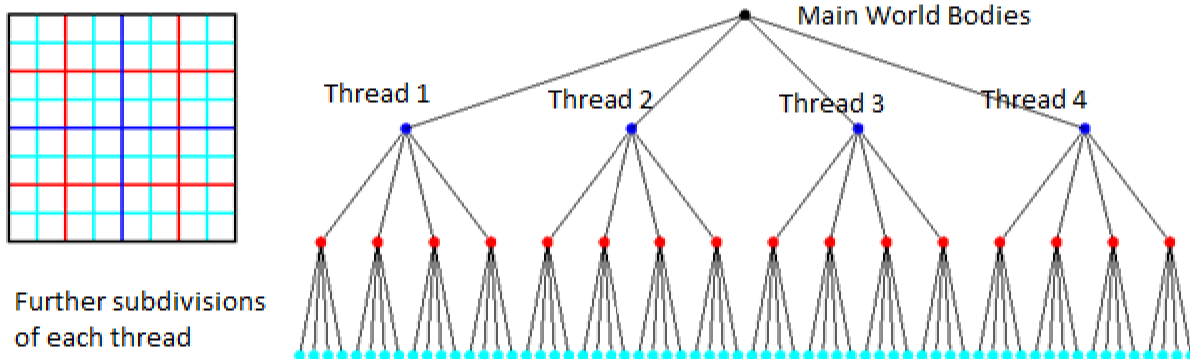

Figure 11. Our original Quad-Tree modified to support four threads.

Just like our single-threaded quad-tree, when a body does not fit completely into the boundaries, it moves up to the parent node. Therefore when a rigid body overlaps the boundaries of two threads, it will be escalated to the master world's bodies. By organizing our data structure as such we are automatically taking care of the collision check between threads. If there is a very large body overlapping multiple threads it will be checked for possible collisions in the same algorithm we implemented in the single-threaded engine.

There is special consideration we must take into account when swapping bodies between threads. The main world must be in charge of the root node. If a body's center crosses from one thread's bounds to another thread's, we simply remove it from the tree and re-insert it to the root node. Like our single-threaded quad-tree algorithm, when we insert this body into the root node it will walk the tree to find the deepest node that the body will fit into. This removal and reinsertion effectively swaps a body from one thread to another. For best efficiency, we should check each rigid body and ensure it fits within the thread's boundaries during the integration phase after its position has been updated.

One major limitation with a geometric data partition must be mentioned. There could be scenarios where one thread is overloaded with bodies while other threads have relatively light loads. If the majority of the bodies are clumped in a corner like this, the engine will slow down to speeds similar to the single-threaded implementation since the data will hardly be partitioned. Possible solutions to this problem would be requiring threads to assist others if high loads are received or utilize alternative methods of partitioning. Other methods of partitioning have their own limitations as well however, so we feel that while the geometric partition has this specific problem, it is not a common enough occurrence to warrant the use of a slower, on average, partitioning scheme.

4.5.4.2 Parallel Contact Resolution

As collisions are being checked between threads, there is the possibility that two identical contacts are simultaneously created on two different threads. In the case where a rigid body is overlapping two threads' boundaries and a collision occurs during this time, there will be two contacts generated in our

current algorithm, one contact on each thread representing the collision. Not only can this lead to a race condition, it will also result in a contact that is twice resolved.

While techniques such as mutexes would solve this problem, an even easier solution would be to only allow processes to change information on bodies that belong to them. In this way, there will still be two contacts generated. However, when each thread goes to solve this contact, one thread will resolve one body while the other thread will resolve the opposing body. By preventing threads from modifying foreign bodies, we will avoid race conditions and prevent double contact resolution. Adding this constraint to threads allows us to use our previously described single-threaded contact resolution algorithm as-is.

4.6 Software Engineering

A large challenge in designing a physics engine is the software engineering aspect. Especially important for parallel engines, we need to be careful how the engine is designed so that we are maximizing run-time efficiency. Additionally, with a large project like a physics engine, it is important that we have a solid modular foundation so that later improvements can be added with ease. A brief design is pictured below in **Figure 12**.

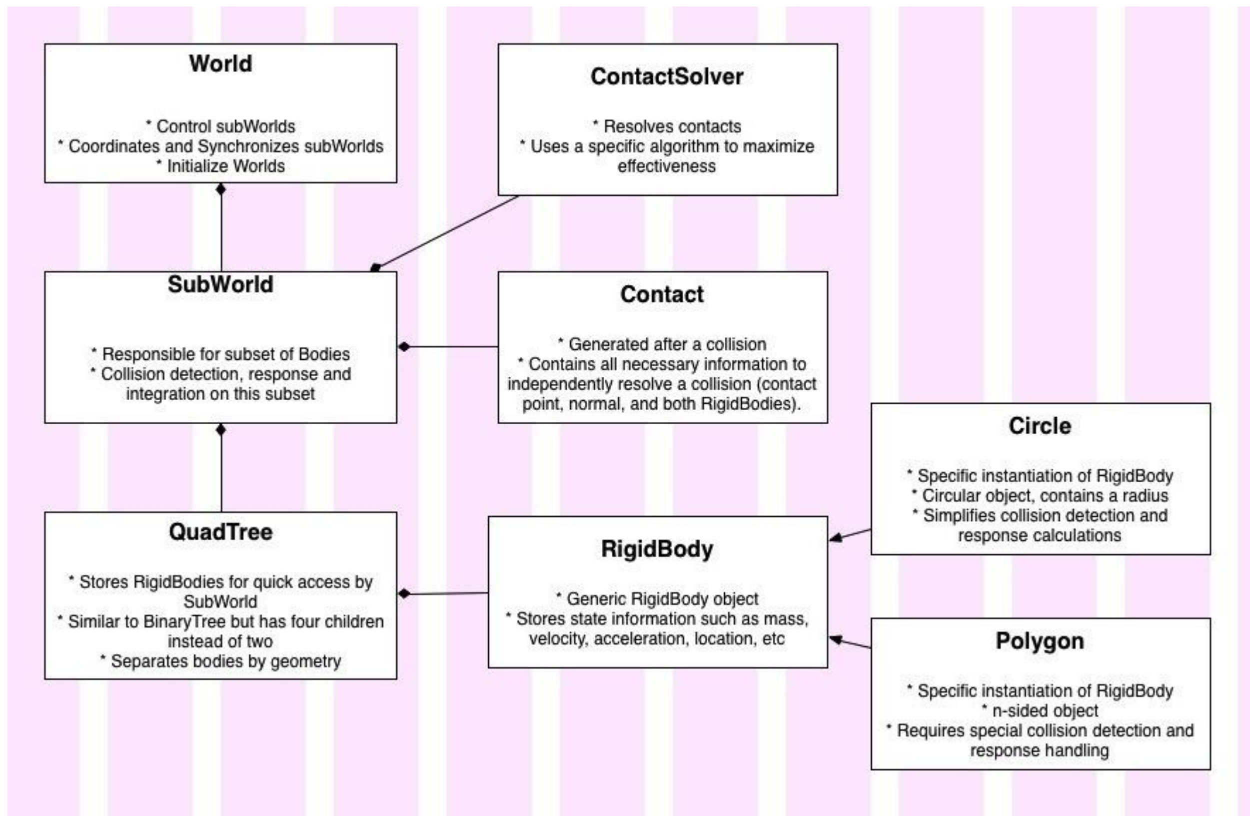

Figure 12. Diagram outlining basic class relationship and responsibilities.

Here, we have set up our physics engine as we have outlined throughout chapter 4. Note that the single-threaded implementation is essentially the same design, except the World and SubWorld are merged into one single-threaded entity.

CHAPTER 5

Results

Now that we have access to a fully working parallel physics engine we will look into what speed increases we have achieved. To do this we must compare to single-threaded implementations. However, this is not a fair comparison. Physics engines vary greatly in all that is implemented so that despite the large amount of engines available, there is not an engine that is exactly equivalent to ours.

The closest engine found that is similar to the physics engine described here is JBox2D. This is a java implementation of Erin Catto's Box2D that was described previously (originally written in C++). JBox2D is the best comparison for us for multiple reasons.

Most importantly both engines are 2-dimensional rigid body implementations. Obviously we cannot compare our 2-d engine to the much more complicated and slower 3-d engines. Additionally JBox2D is written in Java as our engine was. Since Java is ran on a virtual machine, there are typically large speed decreases associated with this. While typically unnoticeable, when we are performing stress tests and comparing very small calculations this speed difference would affect our results.

Data structure usage is also an important consideration. Depending on the data structure used, collision detection can have wildly different run-times for certain scenarios. For example, a static grid can be used extremely effectively if all the bodies are the same size but will perform very poorly if the bodies have varying sizes.

JBox2D utilizes a data structure called a dynamic tree. In brief, this is similar to our quad-tree, but the boundaries of the nodes are dynamic and will resize depending on the bodies inside. The overhead is generally higher than our quad-tree, but the two are comparable in the situations that we present.

Despite all these similarities it is important to note that comparing these two engines is still an unfair test. The code and features for our engine only amount to about 7,000 lines of code. We will be

comparing this to a very large open-source project of many years with nearly 120,000 lines of code. Obviously JBox2D will have many more features implemented and most of these features will cause slower run-times. Unlike our engine, JBox2D was not built from the ground up with speed as the number one concern. And so while we cannot get a fair comparison, we can still compare the two engines under certain stress tests, not to say which is better, but to determine that the parallel approach will show speed increases.

Note that there will be places where JBox2D has speed-ups that we do not support. For example, bodies in JBox2D are put to sleep after a certain amount of time of inactivity. Putting a body to sleep effectively removes it from the collision detection and therefore makes this process much quicker. Another speedup of note is the “warm-startup” technique that JBox2D implements. Warm-startup refers to the process of saving the contact points of the previous physics loop to keep the engine “warm” for the next loop. Instead of deleting the contact upon resolution, JBox2D will keep the contact in the potential list and will only remove it if the two bodies drift far enough apart. This allows the engine to skip some contact generation if the two bodies come into contact again shortly. This technique is especially noticeable when the two bodies are bouncing very close together or are resting upon one another.

In addition to comparing our physics engine to JBox2D, we will also compare our parallel approach to our single-threaded approach. Again, this is not a completely fair comparison because our engine was built from the ground up to be a parallel solution. As such, our single-threaded approach is not the most efficient for single-threaded environments. But it will still provide insight on the possible speed increases of parallelism.

5.1 Circle Stress

Our first test is a circle stress. Circles are generally handled much different in physics engines so we only look at circle vs. circle collisions. In each engine an environment is created where the listed number of circles are given random velocities so that there will be a constant supply of collisions.

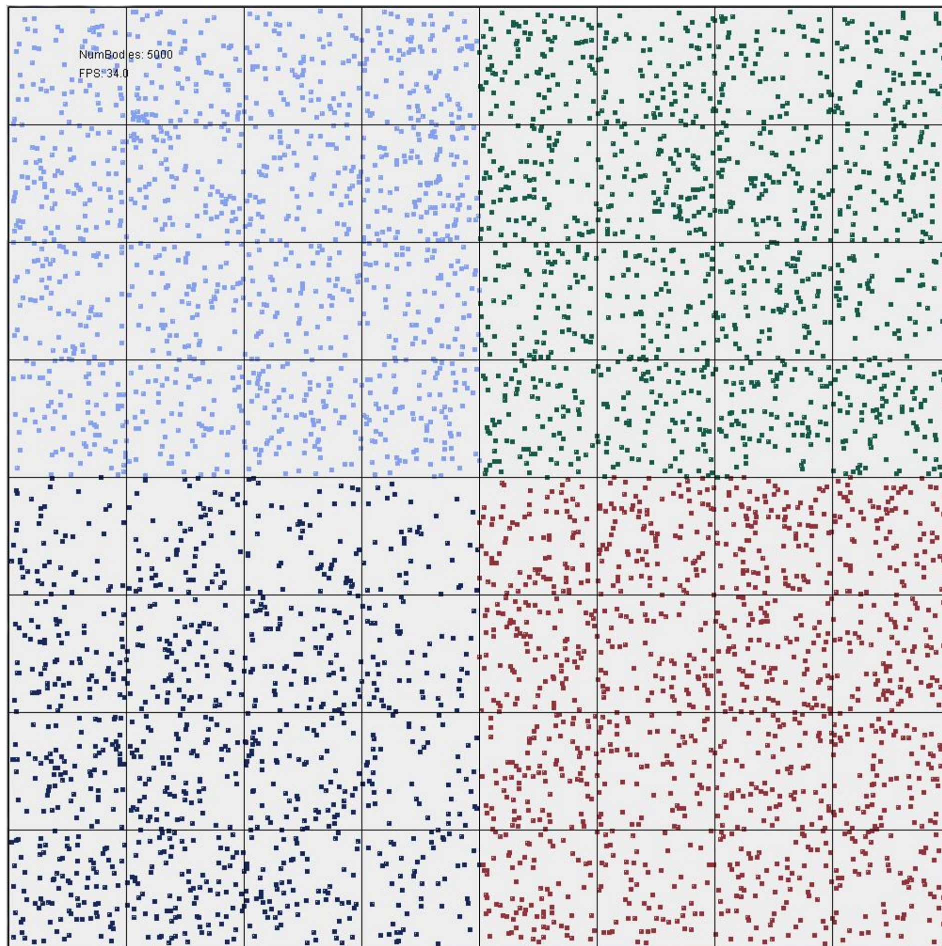

Figure 13. 5k body circle stress test with quadtree drawn. Each color represents a different processor.

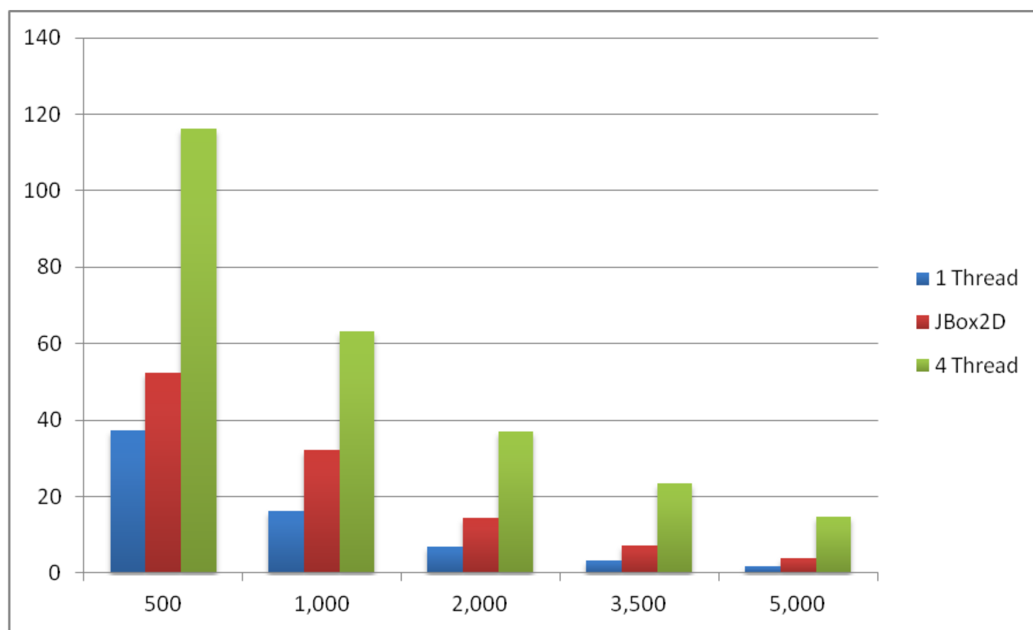

Figure 14. Frames per Second (y-axis) vs. Number of Circles (x-axis) for various engines.

5.2 Polygon Stress

Similar tests as the above for circles, but utilizing polygons with random velocities applied in a closed environment.

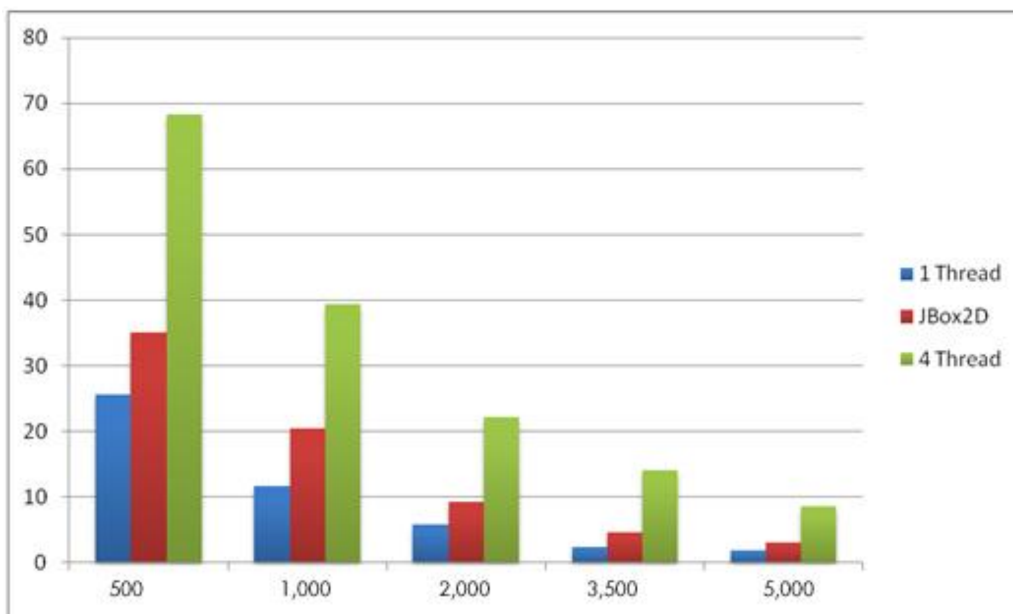

Figure 15. Frames per Second (y-axis) vs Number of Polygons (x-axis) for various engines.

5.3 Mobile Application

Since the engine is written in native Java, the above examples were run on a standard laptop in order to have a stable environment for comparisons. We decided to additionally create a small example game to showcase the physics engine and deploy it to a realistic Android environment.

A "shooter" type game was created where the user must dodge or push meteors out of the way to avoid being hit. This example uses the parallel physics engine exactly as outlined above but in a more graphically appealing environment with the physics happening behind the scenes. The screen is split into four subsections by the quad-tree, collision detection will determine if the player is hit or if the player's lasers have hit a meteor and the appropriate collision response is handled by the engine if a collision is detected.

The controls also utilize the physics engine by applying a velocity to the player's shuttle based on the phone's accelerometer position (i.e. tilting the phone to the right will apply an appropriate impulse on the player's shuttle to the right, etc).

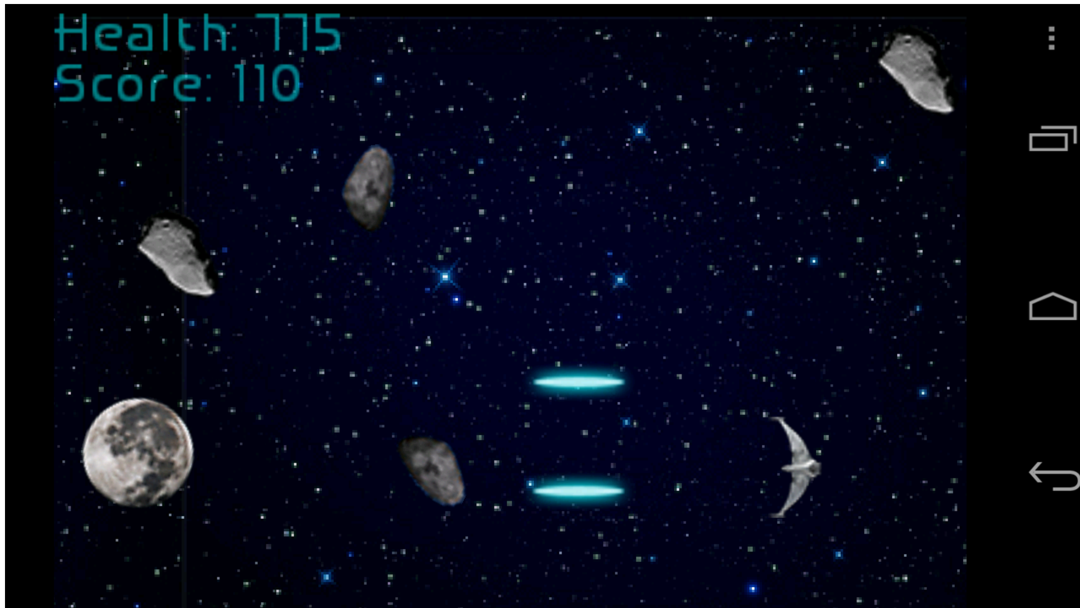

Figure 16. Example game that utilizes the parallel physics engine running on an Android phone.

5.4 Conclusions

Again, it is important not to draw too many conclusions based on these results. While JBox2D is the closest comparable engine it is unfair to say our approach will always give an x% increase in speed. However it does become clear and we believe it is safe to say that a correctly implemented parallel environment will yield a considerable speedup in physics engines. The comparison is not close enough to guarantee a specific coefficient of speed-up but it is evident that a significant speedup will occur.

CHAPTER 6

Further Work and Conclusion

A successful implementation of a parallel 2-dimensional physics engine is outlined above. The topic of physics is enormous, and as such simulating physics is an equally enormous task. Unfortunately we knew that we would not be able to implement everything in the time allowed so we made design decisions from the beginning to leave some work out so that we could focus on a solid parallel implementation.

Due to our design and implementation, many features can be added without having to even consider the parallel element, as the single-threaded approach would work in our environment. The potential feature list is truly limitless, from 3-dimensional support to fluid dynamics to electromagnetic effects. A list of a couple “most effective” features we think would help further enhance the engine are listed below.

6.1 Continuous Collision

Continuous collision detection should replace our current discrete collision detection that is described above. Currently the engine will wait until bodies overlap one another to detect the collision. This means that bodies which are not penetrating one another will not be marked as a collision. While this is completely fine for the vast majority of bodies, if the body is moving very fast or the bodies are very slow this could lead to weird results. Either the body will move completely through the object without being detected or the body could move significantly into the object such that when we resolve the penetration it forces the bodies to unnatural results.

A continuous collisions detection approach would solve this problem, though it is slightly slower and more complex. There are a few possible implementations on how to solve this, but one general technique is to look ahead in time while checking for collisions. This means that instead of checking if each body is overlapping another, we should check to see whether each body will overlap another body within a certain time step. Velocities are then multiplied by the time step and checks would occur to see if any bodies intersected. If a collision is detected the bodies will simply be placed such that they are just touching and no penetration will occur.

6.2 Sleep

Our engine could benefit from a sleep function such as the one implemented in JBox2D. The engine would be able to decide which bodies to put to sleep and once a body is designated as sleeping it would not be considered in future collision detections or integrations. This would reduce the number of bodies that the engine must check or integrate and therefore increase speed. Depending on the environment, this could dramatically increase speed (if the simulation is relatively calm and unmoving) or have no affect (if the simulation is in constant movement and collisions).

Sleep is usually implemented via a boolean value in each rigid body indicating whether or not the object is active or asleep. During runtime the engine will decide when to put this object to sleep and when to make it active again. Typically this is done during the integration phase while we are already being forced to loop through objects to update their positions. If the body did not move in a certain period the body would be a candidate for sleep. Once the body is involved in a new collision the engine would wake it up.

6.3 Warm Startup

Another speed increase could be achieved through the warm startup method that engines such as JBox2D utilize. This method does not delete contacts directly after resolution, but rather waits for some criteria to be met before removing them. This is typically a check on the distance between the two objects and a positive separating velocity. This means that the objects are far apart and moving away from each other (i.e. not likely to collide again soon).

Maintaining a list of contacts will increase stress on computer memory, since it will be keeping track of extra contacts. However, maintenance of such a list decreases CPU time because we can also take advantage of the fact that some contacts are already generated. Objects that are in continuous collision or bouncing very rapidly against each other (consider an object resting on a desk or a ball bouncing very close to the ground) will be constantly creating and deleting contacts in the implementation we used. However, with a warm startup, the engine can save CPU cycles by skipping the contact creation/deletion by skipping directly to resolving this contact.

The largest gains in speed will be noticed in environments that have many resting contacts. Depending on the video game this is usually a quite large speed-up and 10-20% gains are not uncommon.

6.4 Memory Pooling

Finally, the most beneficial speed increase our physics engine could use would be achieved through memory pooling. Memory pooling is the idea of easing the stress on memory allocation and de-allocation processes. In our current physics engine, we are calling Java's "new" operator thousands of times each physics cycle. This also means that Java's garbage collector is responsible for cleaning up these thousands of objects as well.

These constant memory allocations are quite a strain on the system which could be alleviated by using a pool of pre-allocated objects that will not be constantly de-allocated. Memory pools are best used on objects that are used most frequently. Therefore, the best object to do this would certainly be the "Vector2" object we created. This object is created and deleted the most during a physics loop as we need intermediate 2-dimensional vectors in almost all the equations and techniques described above. Instead of creation and deletion, it would be much more efficient to pre-allocate a large pool of 2-d vectors at the initialization of the engine.

Then, rather than calling "new" to create a vector, we ask our pool for an empty vector which will return a pointer to an unused 2-dimensional vector object. We can use this object as we would any other object and once we are done we can ask the pool to reclaim the object. Rather than have Java's garbage collector take care of the cleanup, a call to the pool notifying it that the vector is no longer being used would be much quicker. This vector will then go back onto the unused stack waiting to be called again.

6.5 Conclusion

Physics engines have largely been dominated by single-threaded techniques due to their simplicity and cost effectiveness. Yet with increasing demand for better utilization of physics in video games, movies and science, a change must come soon to the standard practice used in video games. The academic community has made large advances in the subject; especially noteworthy was *pe,* developed by

students at the University of Erlangen-Nuremberg. PhysX is certainly the most advanced of these engines with the ability to scale to hundreds of cores and support multiple platforms and physics environments.

Unfortunately, there is not currently an available solution that is both practical and open-source. We have combined some ideas of previous projects and implemented our own visions to fill this void in the parallel physics engine community. Due to the vast subject encompassed in the scope of physics, our engine is certainly limited in features available. However, with the large push towards parallel environments in all software, parallel physics engines will become the new industry norm. Eventually, with the help of large industries including the gaming and scientific communities, large-scale parallel physics engines that are open to the public will become as common as the single-threaded counterparts are today.

References

[1] Catto, Erin. *Box2D*. Web. 07 Oct. 2013. <http://www.box2d.org/>.

[2] *Nvidia PhysX.* Web. 10 Oct. 2013. <http://www.geforce.com/hardware/technology/physx>.

[3] "Physics Engine." *Wikipedia*. Wikimedia Foundation, 20 Oct. 2013. Web. 05 Nov. 2013.

[4] "Rigid Body Dynamics." *Wikipedia*. Wikimedia Foundation, 31 Oct. 2013. Web. 05 Nov. 2013.

[5] Taylor, John R. *Classical Mechanics*. Sausalito, CA: University Science, 2005. Print.

[6] Millington, Ian. *Game Physics Engine Development: How to Build a Robust Commercial-grade Physics Engine for Your Game*. Amsterdam: Morgan Kaufmann, 2010. Print.

[7] Ericson, Christer. *Real-time Collision Detection*. Amsterdam: Elsevier, 2005. Print

[8] Baraff, David. *An Introduction to Physically Based Modeling*. N.p.: n.p., n.d. Print.

[9] Bender, Jan, Kenny Erleben, Jeff Tinkle, and Erwin Coumans. "Interactive Simulation of Rigid Body Dynamics in Computer Graphics." (2012): n. pag. Web. 6 Oct. 2013. <http://www.cs.rpi.edu/twiki/pub/RoboticsWeb/LabPublications/BETCstar_part1.pdf>.

[10] Goodstein, Michelle, Michael Ashley-Rollman, and Paul Zagieboylo. "Parallelizing the Open Dynamics Engine." CMU, n.d. Oct. 07 Mar. 2013. <http://www.cs.cmu.edu/~mpa/ode/final_report.html>.

[11] "N Tutorial A - Collision Detection and Response." *N Tutorial A - Collision Detection and Response*. N.p., n.d. Web. 05 Nov. 2013. <http://www.metanetsoftware.com/technique/tutorialA.html>.

[12] Iglberger, Klaus, and Ulrich Rude. "Massively Parallel Rigid Multi-Body Dynamics." (2009): n. pag. University of Erlangen. Web. 6 Oct. 2013. <http://www10.informatik.uni-erlangen.de/Publications/TechnicalReports/TechRep09-8.pdf>.

[13] "Collision Detection Using the Separating Axis Theorem." *Gamedevtuts RSS*. N.p., n.d. Web. 05 Nov. 2013.

[14] Tonge, Richard. Iterative Rigid Body Solvers. Nvidia Corporation. Game Developers Conference 2013.

[15] Firth, Paul. "Paul's Blog@Wildbunny." *Pauls BlogWildbunny*. N.p., n.d. Web. 05 Nov. 2013.

[16] Kaufman, Danny, Timothy Edmunds and Dinesh Pai. "Fast Frictional Dynamics for Rigid Bodies". (2005): n. pag. Web. 10 Oct. 2013. <http://graphics.stanford.edu/courses/cs468-05-fall/Papers/p946-kaufman.pdf>.

[17] "List of Moments of Inertia." *Wikipedia*. Wikimedia Foundation, 31 Oct. 2013. Web. 05 Nov. 2013. <https://en.wikipedia.org/wiki/List_of_moments_of_inertia>.

[18] "Minkowski Addition." *Wikipedia*. Wikimedia Foundation, 27 Oct. 2013. Web. 05 Nov. 2013. <https://en.wikipedia.org/wiki/Minkowski_addition>.