PHYSICS ENGINE ON THE GPU WITH OPENGL COMPUTE SHADERS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Quan Bui

March 2021

© 2021
Quan Bui
ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE:	Physics Engine on the GPU with OpenGL Compute Shaders
AUTHOR:	Quan Bui
DATE SUBMITTED:	March 2021
COMMITTEE CHAIR:	Christian Eckhardt, Ph.D. Assistant Professor of Computer Science
COMMITTEE MEMBER:	Maria Pantoja, Ph.D. Associate Professor of Computer Science
COMMITTEE MEMBER:	Franz Kurfess, Ph.D. Professor of Computer Science

ABSTRACT

Physics Engine on the GPU with OpenGL Compute Shaders

Quan Bui

Any kind of graphics simulation can be thought of like a fancy flipbook. This notion is, of course, nothing new. For instance, in a game, the central computing unit (CPU) needs to process frame by frame, figuring out what is happening, and then finally issues draw calls to the graphics processing unit (GPU) to render the frame and display it onto the monitor. Traditionally, the CPU has to process a lot of things: from the creation of the window environment for the processed frames to be displayed, handling game logic, processing artificial intelligence (AI) for non-player characters (NPC), to the physics, and issuing draw calls; and all of these have to be done within roughly 0.0167 second to maintain real-time performance of 60 frames per second (fps). The main goal of this thesis is to move the physics pipeline of any kind of simulation to the GPU instead of the CPU. The main tool to make this possible would be the usage of OpenGL Compute Shaders. OpenGL is a high-performance graphics application programming interface (API), used as an abstraction layer for the CPU to communicate with the GPU. OpenGL was created by the Khronos Group primarily for graphics, or drawing frames only. In the later versions of OpenGL, the Khronos Group has introduced Compute Shader, which can be used for general-purpose computing on the GPU (GPGPU). This means the GPU can be used to process any arbitrary math computation, and is not limited to only process the vertices and fragments of polygons. This thesis features Broad Phase and Narrow Phase collision detection stages, and a collision Resolution Phase with Sequential Impulses entirely on the GPU with real-time performance.

ACKNOWLEDGMENTS

My first thanking words are for my advisor, Dr. Christian Eckhardt, for his invaluable feedbacks throughout this study and also his initial idea that started this thesis. I would like to also say thanks to Dr. Zoe Wood for showing me the wonder of computer graphics. Then, my appreciation goes out to Dr. Maria Pantoja and Dr. Franz Kurfess for coming to my defense as my committee. Next, I would like to express my gratitude for my family, especially my parents, for their tremendous and unconditional support. Lastly, my appreciation goes out to my friends for their kind encouragement.

TABLE OF CONTENTS

Page

LIST OF FIGURES . . . viii

CHAPTER

1 Introduction . . . 1

1.1 Why . . . 1

1.2 Design Philosophy . . . 3

1.3 Thesis Outline . . . 3

2 Related Work . . . 4

3 Methodologies . . . 5

3.1 Physics Pipeline . . . 5

3.2 Broad Phase with Sweep and Prune . . . 9

3.3 Narrow Phase with Separating Axis Test . . . 13

3.3.1 Signed Distance and Support Point . . . 13

3.3.2 Query for Closest Feature . . . 14

3.3.3 Contact Generation to Form Contact Manifolds . . . 16

3.3.4 Contact Point Reduction . . . 20

3.4 Resolution Phase with Sequential Impulses . . . 21

3.4.1 Resolving Overlapping with Contact Constraint along the Contact Normal . . . 22

3.4.2 Simulating Friction with Contact Constraint along the Contact Tangents . . . 26

3.4.3 Solving for λ . . . 26

3.5 OpenGL Comes into Play . . . 27

4 OpenGL and Optimization . . . 30

4.1 OpenGL Asynchronous Nature 31
4.2 Optimization in Data Streaming 32
5 Results 35
6 Conclusion 38
BIBLIOGRAPHY 39

LIST OF FIGURES

Figure		Page
1.1	Simulation loop	2
3.1	A typical physics pipeline	6
3.2	Broad Phase shows colliding bodies in red, and Narrow Phase displays the contact points as pink dots.	7
3.3	Resolving collision - both body A and body B will receive impulses that will change their linear velocities and angular velocities.	8
3.4	Final condition - what happens after the collision has been resolved.	9
3.5	An Axis-aligned Bounding Box	10
3.6	Sweep and Prune. From: [7]	12
3.7	Separating Axis Test	17
3.8	Sutherman-Hodgeman Polygon Clipping	19
3.9	Contact reduction. From: [6]	21
3.10	Parallel Broad Phase	28
3.11	Parallel Narrow Phase	28
3.12	Parallel Sequential Impulses	29
5.1	Simple stacking	36
5.2	Pool of boxes	37
5.3	Extreme stacking	37

Chapter 1

INTRODUCTION

1.1 Why

The first question you would ask is why would I reinvent the wheel; there are other physics engines out there that had had multiple engineers working together and spending years (roughly 10 years for Bullet engine for instance) to perfect them. Without a doubt, these engines will straight up beat mine in performance; however, I would argue that there is no other way to learn the architecture of a specific system without making it, even if it is just a prototype. That would beat just reading the source code.

Furthermore, this physics engine uses the parallel computing power of the GPU for the backend, which not a lot of physics engines out there utilize. A physics engine can be used for both games and simulations, where the later requires more accuracy than the former. Games tend to not have a lot of rigid bodies colliding so most of the physics implementations are on the CPU side, for which the code can be benefited from perks that the CPU has that help with the branchiness nature of a physics tick, such as branch prediction and cache locality. Although optimizers for GPUs are getting better, they are still not as matured as that of the CPUs. For that reason, branching is awfully expensive for the GPU to process, most of the time the GPU would process all branches and the results of the unwanted branches would be masked out.

Figure 1.1 shows a typical simulation loop. After initializing the simulation, the CPU first has to process any simulation logic that is not physics related, for example in

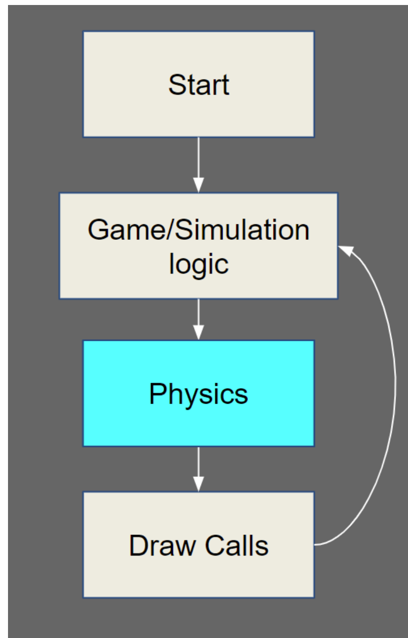

Figure 1.1: Simulation loop

a game there might be some logic for NPCs that needs to be processed before any physics logic. Then, it has to perform numerous calculations to apply physics into the simulation. Finally, the CPU has to issue draw calls to the GPU so that frames can be rendered and displayed to the screen for the users to see. As you can see, the CPU has to manage quite a mountain of work; so to reduce that weight, I propose to move the entire physics pipeline onto the GPU, see if that would help the performance a bit. That is the main motivation for this thesis.

Due to how performance-sensitive of the GPU implementation can be, multiple optimization techniques have to be used, and they will be discussed later in their own sections. This is also to increase the robustness of the GPU implementation; however, depending on the hardware and the scale of the simulation - which is quite little in games and moderate to huge in a full-fetched scientific simulation - users can manually switch to use the CPU implementation instead, which is also included in the source code.

1.2 Design Philosophy

OpenGL is great for prototype because it can show the results relatively quickly as there are not a lot of setups (comparing to DirectX 12 or Vulkan). However, it is bad for performance as it leaves little room for optimization because it likes to assume and do things for the users, even without being asked. This sort of abstraction is great for beginners and maybe for some who do not like to work so close to the metal. Personally, I dislike this abstraction. I want to design something that would lay bare with minimal to no abstraction.

With that being said, my physics engine will do exactly what the user tells it to do, nothing more and nothing less. Furthermore, it will not do a lot of things feature-wise either; it can do only a few things, but it will do those well. As of now the engine only supports cube as hitbox.

1.3 Thesis Outline

There are 3 main components for a physics engine: (1) Broad Phase, (2) Narrow Phase, and (3) Resolution Phase; and this thesis is split up like so. Each phase can be focused and expanded to a whole thesis. However, I will not be presenting any new idea on these phases; mostly, I will follow the recipes of successful physics engines out there and present my own interpretation of the implementations.

I also will focus on the performance side of the engine as well. As mentioned before, OpenGL is not great for performance, in both graphics and general computing. However, I will be attempting to use whatever available in OpenGL specs to squeeze out as much performance as I can.

Chapter 2

RELATED WORK

Physics engine is a huge topic that combines many concepts together, from the advanced mathematical concepts in collision detection to the Newtonian Dynamics in the Resolution Phase. Broad and Narrow collision detection phases are hugely inspired by Bullet engine, which was developed and is maintained mainly by Erwin Coumans. It has been the top-of-the-line and open-sourced 3D physics engine in countless large-scaled and popular projects, including, but not limited to, modelling softwares - such as Maya, Houdini, and Blender, etc - movies, and games.

The Resolution Phase utilizes the concept of Sequential Impulses to prevent rigid bodies from overlapping and solve the collisions accordingly to Newtonian Laws of Motion. Sequential Impulse was popularized by Erin Catto. He is the developer of another popular physics engine, though only for 2D but still one of the best, named Box2D. Randy Gaul has his physics engine, dubbed qu3e, fairly similar to Box2D, but it is for 3D instead.

Out of the 2 aforementioned engines, only Bullet has a rigid body pipeline implemented on the GPU. Coumans uses OpenCL for his implementation. OpenCL is a computing API from Kronos Group, which also developed OpenGL. PhysX, another well-known physics engine developed by Nvidia Corporation, also has its rigid body simulation pipeline on the GPU in CUDA. As far as I know, there is no known implementation with OpenGL Compute Shaders, which makes my engine potentially the first of its kind.

Chapter 3

METHODOLOGIES

Let's first define some terminologies. In our engine, the user can create an abstract world that is affected by physics, and we shall call it the dynamics world. This dynamics world supports 3 types of objects (or bodies, the term object and body will be used interchangeably throughout this thesis; although from this point on the term body would be used more dominantly):

- **Static**: unaffected by gravity and cannot be moved by any mean, effectively has infinite mass. For example, the ground or some sort of platform that the controllable character can jump on.
- **Rigid**: Affected by gravity and will move accordingly to Newton's Laws of Motions. Has finite mass defined by user. For example, particles or fragments from a destroyed door.
- **Kinematics**: Can be affected by gravity and the movement is controlled by the user. For example, the controllable character in games.

3.1 Physics Pipeline

As shown in figure 3.1, in order to apply physics onto bodies in the world, the engine needs to know which bodies are colliding first. The user first needs to assign what kind of collision shape to each body. Then the collision checks will be done accordingly to these collision shapes. Each collision check is always done in pairs, for example between body A and body B - which form a **collision pair**. Without a proper

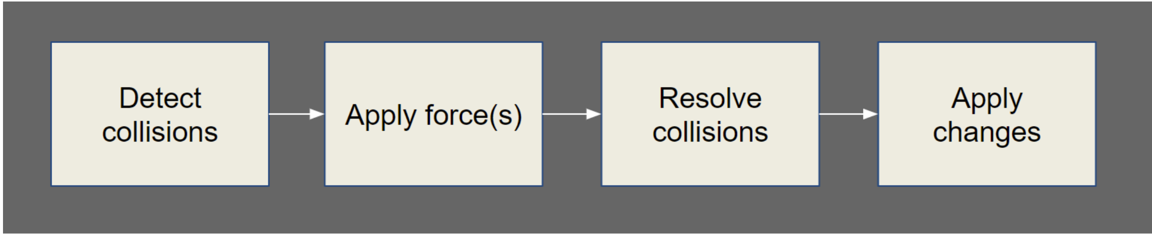

Figure 3.1: A typical physics pipeline

pipeline, the easiest thing to do is to just do an n^2 check with everything in the world:

Algorithm 1: Brute Force Collision Check

Input: A list of **Box Colliders**
Output: A list of **Collision Pairs**

1 **for** *each **body** in the world* **do**
2   **for** *each **other body** in the world* **do**
3   **if** *they collide* **then**
4   add the collision pair to the list

This is generally fine if there are not a lot of bodies in the dynamics world, however it is still extremely inefficient depending on how expensive (time wise, the more clock cycles each check requires, the more expensive it will be) each collision check is. Also, there are no precise information on where the collision has occurred. More sophisticated solution is needed. In general, the collision detection pipeline consists of 2 stages as shown in figure 3.2:

- **Broad Phase**: Apply rough collision checks to eliminate pairs that can never collide, for example 2 balls can never interact if each one is in the opposite corner of the room. This is where many clever algorithms and data structures are used to dramatically reduce the number of collision checks that the Narrow Phase has to do. Some typical examples are Bounding Volume Hierarchy Tree, Octree, Sweep and Prune, etc; this thesis will focus on Sweep and Prune algorithm. Note

that false positives are common and expected from this stage. In figure 3.2, the Broad Phase reports the 2 colliding bodies in red.

- **Narrow Phase**: At this stage, each collision pair is not guaranteed to collide (false positive), so more expensive and elaborate checks are used to make sure a collision has happened. This is where the collision shapes are taken into account. Furthermore, this stage also generates contact points (as shown as the pink dots in figure 3.2) - where specific in the world the collision happens - and penetration depth - how far the bodies are overlapping. These contact points will be sent to the constraint solver in the Resolution Phase to resolve the collision.

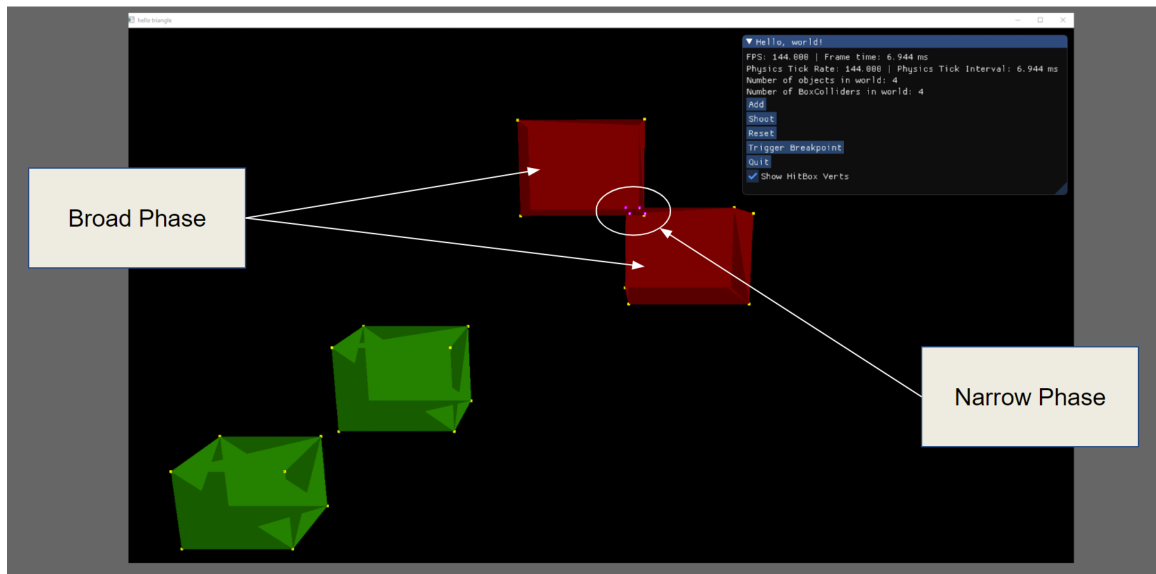

Figure 3.2: Broad Phase shows colliding bodies in red, and Narrow Phase displays the contact points as pink dots.

After the collision detection stage, we shall apply the gravity to all rigid bodies by reducing their velocities in the y direction with some amount. After that the physics has to resolve any collision detected from the Narrow Phase. The 2 most basic things the Resolution Phase has to do is to update the world space positions of all the moving

bodies so that they do not phase past each other and also their orientations (if their collisions cause rotations).

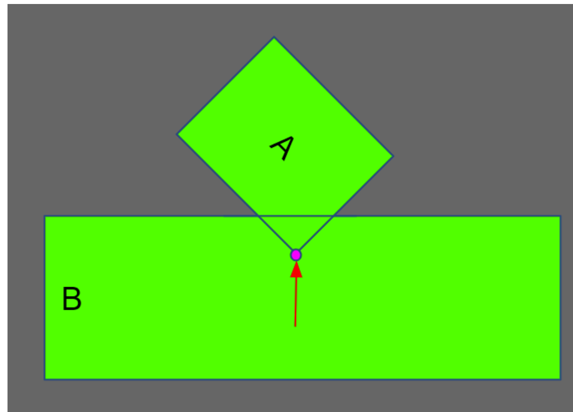

Figure 3.3: Resolving collision - both body A and body B will receive impulses that will change their linear velocities and angular velocities.

As shown in figure 3.3, when body A and body B collide, an impulse would be applied onto the contact point on A (the purple point) to push A out of B. By Newton's Third Law, an equal and opposite impulse would also be applied back onto body B. Both of these 2 linear impulses are going to cause a change in the linear velocities of both bodies. Furthermore, depending on the how offset the impulse is applied relative to the center of mass of each body, there can also be some changes in their angular velocities.

Once the changes in linear velocities, Δv's, and angular velocities, $\Delta \omega$'s, are computed, they can be applied to the initial conditions, v_i's and ω_i's, to get the final conditions, v_f's and ω_f's. From that, the final positions and orientations of both bodies can be computed by integrating the v_f's and ω_f's with respect to time.

For the Resolution Phase, this physics engine uses the concept of Sequential Impulses as the solver. Resolving a huge number of collisions is essentially solving a system of linear equations; there can be global solutions that satisfy all of the linear equations, but in order to solve all the equations, it would take too long for the engine to run

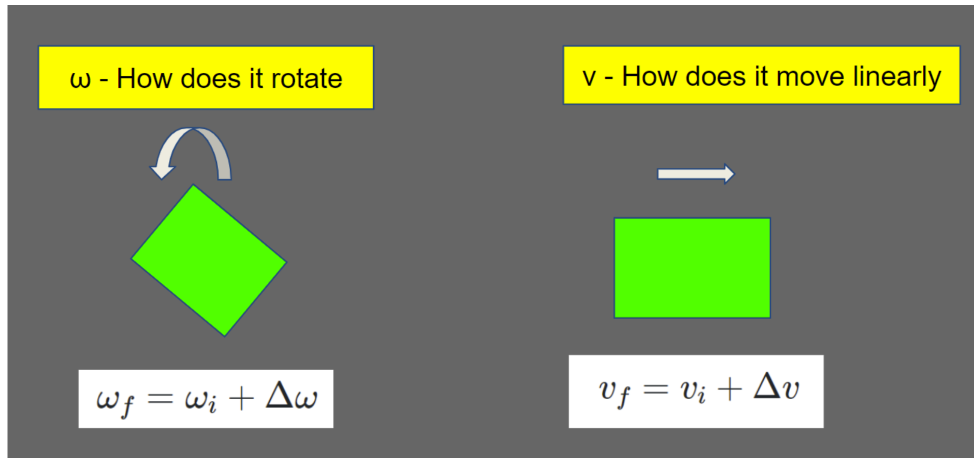

Figure 3.4: Final condition - what happens after the collision has been resolved.

in real time. Instead, the global solution can be estimated with some numerical method, and that is Sequential Impulses. Overall, the engine will formulate multiple collisions with the concept of multiple constraints, and to resolve all the constraints, it is analogous to solving multiple linear equations with Sequential Impulses.

3.2 Broad Phase with Sweep and Prune

In a collision pair, each body has extensions in space; if we project the extensions of the 2 bodies onto all 3 Cartesian axes (x-axis extends from left to right, y-axis extends from down to up, and z-axis extends from backward to forward) and found an overlap on 1 axis, then we shall add the pair to the list that will be checked again by the Narrow Phase.

Axis-aligned Bounding Box (AABB) represents this concept of extensions in 3D. To keep this simple, each AABB has 3 pairs of floats: x_{min} and x_{max}, y_{min} and y_{max}, z_{min} and z_{max}. Each pair represents the extension of a physical object in the x-axis, y-axis, and z-axis respectively.

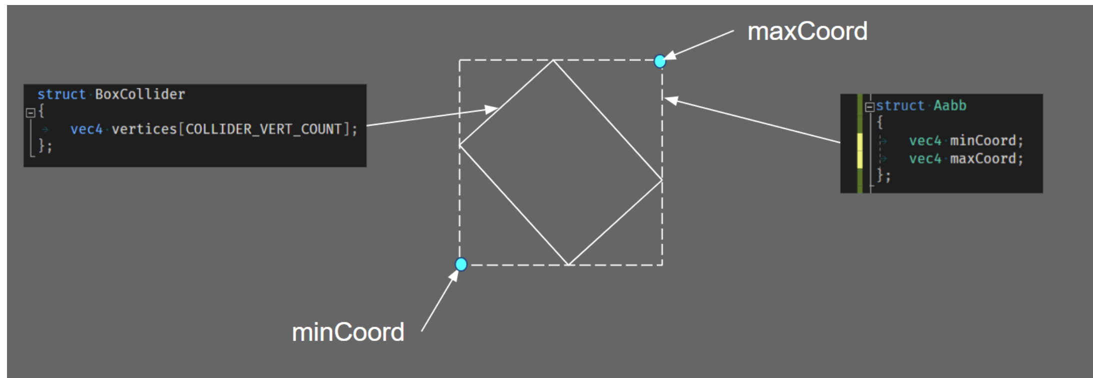

Figure 3.5: An Axis-aligned Bounding Box

Figure 3.5 shows an example of an AABB for a body in 2D, the minimum coordinates $(x_{min},\ y_{min})$ would be the lower-left teal point, and the maximum coordinates $(x_{max},\ y_{max})$ would be the upper-right teal point. BoxCollider struct stores all the raw vertices in space that describe a physical body in our dynamics world.

We are going to process each dimension separately and in pairs of AABBs; sometimes, this is also called sweeping an axis. So when we are only looking at the x-coordinates, it is said that we are sweeping the x-axis. The method starts with sweeping the x-axis, if 2 AABBs are overlapping, then we shall check if they are also overlapping in the y-axis, and finally the z-axis.

For overlapping to happen in a certain axis, in our case the x-axis, these 2 conditions must be satisfied: $x_{min,\ A} < x_{max,\ B}$ and $x_{max,\ A} \geq x_{min,\ B}$, where A and B denoting the 2 unique AABBs. Note that the equal condition is arbitrary; it won't hurt to have it set as shown, but it's also fine to not have it - it really depends on how you handle edge cases. We shall process the y-dimension and z-dimension similarly as the presented conditions.

If the above conditions fail on any dimension, that means there is at least one axis of separation, and the pair of AABBs is immediately removed from any further con-

sideration; this step is called pruning, hence the name of the method: we sweep a particular axis and prune the pair if necessary.

So let's expand the algorithm 1 a bit to algorithm 2. Note that we now use a new struct called CollisionPair. A CollisionPair struct simply holds the IDs of the 2 objects that potentially in collision, and it can look something like this:

```
struct CollisionPair
{
    int objectAID;
    int objectBID;
};
```

Algorithm 2: Sweep and Prune

Input: A list of AABBs

Output: A list of Collision Pairs

1 **for** *$AABB_A$ in the list of AABBs* **do**
2   **for** *$AABB_B$ in the list of AABBs* **do**
3   **if** $x_{min,A} < x_{max,B}$ *and* $x_{max,A} \geq x_{min,B}$ **then**
4   Add the 2 bodies into the list of Collision Pairs

5 **for** *each **Collision Pair** in the previous list of Collision Pairs* **do**
6   **if** $y_{min,A} < y_{max,B}$ *and* $y_{max,A} \geq y_{min,B}$ **then**
7   Keep the 2 bodies in the list of Collision Pairs
8   **else**
9   Remove the 2 bodies from the list of Collision Pairs

10 **for** *each **Collision Pair** in the previous list of Collision Pairs* **do**
11   **if** $z_{min,A} < z_{max,B}$ *and* $z_{max,A} \geq z_{min,B}$ **then**
12   Keep the 2 bodies in the list of Collision Pairs
13   **else**
14   Remove the 2 bodies from the list of Collision Pairs

To visualize the algorithm, here's a nice diagram by Nilson Souto [7]:

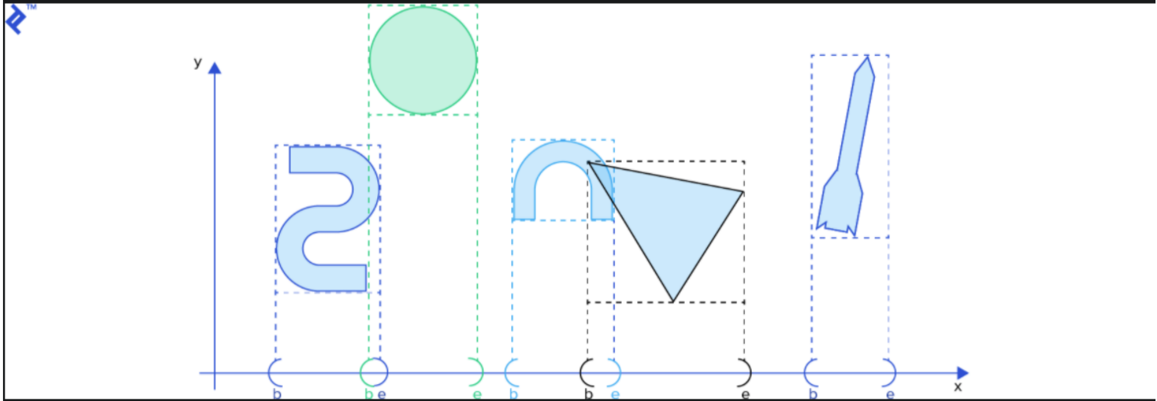

Figure 3.6: Sweep and Prune. From: [7]

Some final words for this section, this is rather a simple testing method for broad phase. To be frank, this is just the naive big O(n^2) comparing-every-possible-pairs method with some tunings. A more efficient method would be using a tree structure to represent an area hierarchy for testing collision. For instance, if the object we are trying to test for collision is in the upper right quarter of the world (for the sake of simplicity, we are looking at a 2D world), then the object only needs to be tested for that specific region of space. In 2D, this method is called Quad Tree, and in 3D, it's called Octree. The region of space can be nonuniform as well (as opposed to being uniform like the Quad Tree would divide the world into 4 equal quadrants, and the Octree would divide the world into 8 equal octants), if so the method would become Bounding Volume Hierarchy (BVH) tree. These methods are highly recommended to if the Broad Phase were to be implemented on the CPU. However, there are several disadvantages to do this on the GPU, albeit Nvidia engineers have figured out how to efficiently implement a binary tree structure on the GPU. Firstly, OpenGL Shading Language (GLSL) does not allow the usage of pointers nor recursion, which are two things are heavily used when working with tree structures, though not strictly needed. Secondly, the branchiness nature of working with tree structure would not

suit the shader code, remember we would like to avoid as much branching as possible. However, no benchmark has been done in this thesis to support this statement.

3.3 Narrow Phase with Separating Axis Test

The Narrow Phase is more complicated that the Broad Phase presented above, since it involves multiple small algorithms together. Overall, the Separating Axis Test is done in pairs of bodies (i.e body A and body B), and there are 3 tasks to finish: (1) query for closest feature, (2) create manifold for said closest feature, and (3) reduce the contact points in the manifold if needed.

The physics engine currently only supports box colliders (however, the overall pipeline should support a generic convex hull of vertices, meaning almost all convex polyhedra, given that the user define the faces and edges of the convex hull correctly). With the box colliders, there are 2 features that they can collide, the face and the edge. For a stable contact manifold, if the colliding feature is a face, 4 contact points are needed; and if the colliding feature is an edge, then only 1 contact point is needed. Having more contact points would just make the contact solver in the Resolution Phase work a bit harder for no good reason.

3.3.1 Signed Distance and Support Point

Before we get our hands dirty with the Separating Axis Test, we first need to know about the 2 essential algorithms that will be used very regularly: (1) finding the signed distance between a plane and a point, and (2) computing a Support Point in a certain direction.

Computing the signed distance of a point A is done by projecting the position of said point onto the plane in question. If the point is on the positive side of the plane - meaning if we were to draw a vector from any point on the plane to the point A, the vector would have the same direction as the plane normal - then the signed distance is positive to indicate that. Similarly, if point A is on the negative side of the plane, then the signed distance would be negative. To project the point onto the plane, we simply compute the dot product of the normalized plane normal and the vector formed from any point on the plane to point A.

Algorithm 3: Compute Signed Distance

Input: A **point** and a **plane**
Output: The signed distance of **point** to the **plane** as a float
1 **return** *dot(plane.normal, point - plane.point)*

Computing the support point is simply about finding the point that is farthest in a certain direction. The easiest way to do this is to compute all the dot products between each point and the given direction, whichever point has the greatest dot product is the support point.

Algorithm 4: Compute Support Point

Input: A collection of **points** and a **direction**
Output: The support point
1 **for** *each* ***point*** *in the input collection of* ***points*** **do**
2   Compute the dot product of the point and the **direction**
3   **if** *The dot product is the greatest dot product so far* **then**
4   Store the current dot product and the current **point**
5 **return** *the* ***point*** *with the greatest dot product with the input* ***direction***

3.3.2 Query for Closest Feature

The definition for a Face Query is:

```
struct FaceQuery
```

```
{
  int faceIdx;
  float largestDist;
  vec3 faceNormal;
};
```

Before the global positions of the contact points are computed, we need to first figure out which are the 2 features involved in this contact pairs, faces or edges. In the case of the box colliders, only faces can be the contact features. However, we still need to know which faces are involved:

Algorithm 5: Query Face Feature

Input: Box Collider A and **Box Collider B**

Output: A **Face Query**

1 **for** *each* ***face*** *from* ***Box Collider A*** **do**
2   Expand the face into a plane by using the face normal and any arbitrary face vertex
3   Compute the support point from **Box Collider B** in the inverse of the plane normal
4   Compute the signed distance from the support point to the plane
5   **if** *the signed distance is the smallest signed distance so far* **then**
6   Store the signed distance and the current face index into the Face Query struct
7 **return** *the final* ***Face Query*** *with the smallest signed distance and the corresponding face index from body A*

The definition for an Edge Query is:

```
struct EdgeQuery
{
  float largestDist;
  vec3 pointsA[2];
  vec3 pointsB[2];
  vec3 edgeDirA;
  vec3 edgeDirB;
  vec3 edgeNormal;
```

```
9 };
```

Just for completeness, algorithm 6 shows how to query edge feature.

Algorithm 6: Query Edge Feature

Input: **Box Collider A** and **Box Collider B**
Output: An **Edge Query**

1 **for** *each* ***edge*** *of* ***Box Collider A*** **do**
2   **for** *each* ***edge*** *of* ***Box Collider B*** **do**
3   Find the edge normal by computing the cross product of 2 edges
4   Correct the direction of the computed edge normal if needed. For consistency, make sure the edge normal is always pointing away from **Box Collider A**'s center
5   Expand the edge normal to a plane by using the edge normal and a vertex from **Box Collider A**
6   Compute the support point from **Box Collider B** in the inverse direction of the plane normal
7   Compute the signed distance from the support point to the plane
8   **if** *the signed distance is the smallest signed distance so far* **then**
9   Store the signed distance and the current edge pair indices into the Edge Query struct
10 **return** *the final* ***Edge Query*** *with the smallest signed distance and the corresponding edge pair indices from both* ***Box Collider A*** *and* ***Box Collider B***

3.3.3 Contact Generation to Form Contact Manifolds

Before everything, we need to determine which one is the reference box and which one is the incident box. This can be arbitrary in the implementation, but it needs to be consistent over nearby frames to avoid feature flip-flopping. In the case of having faces as the colliding features, once the reference box and incident box are identified, then the reference face and incident face can be determined by using the generated FaceQuery struct from the querying face feature step.

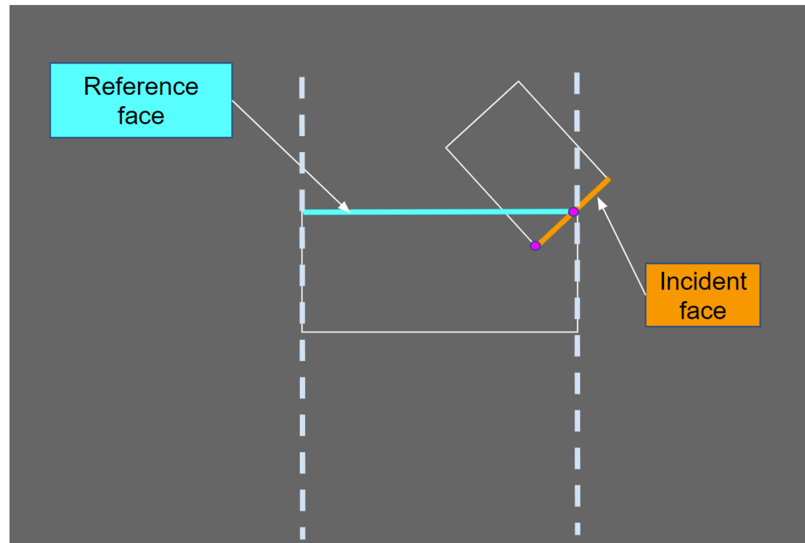

Figure 3.7: Separating Axis Test

The next step is to clip the incident face against all the side faces of the reference face (on the reference box, of course). We use Sutherman-Hodgeman polygon clipping algorithm to generate contact points as shown in algorithm 7.

In figure 3.8, the teal points are outside the plane, meaning they have positive signed distance to the clipping plane, so we discard them. We only keep the points with negative signed distance and the intersection points between the face and the plane (as shown in purple points).

In the case of having edges as the colliding features, we just need to find the one closest point between 2 line segments (since edges are simply line segments). This method is mentioned in [4] on page 149. This algorithm will not be discussed in this thesis because it is about following some number of formulas, which is quite difficult to write pseudocode for. Note that in 3D, there are multiple geometric degenerate cases that might cause problems, i.e 2 vertices have the same position in space resulting in a point instead of a line, etc. However, since we are dealing with convex polyhedra, we shall assume that we will not be encountering these degenerate cases; therefore, most of the initial if checks mentioned in that book can be ignore for simplicity.

Algorithm 7: Sutherman-Hodgeman Polygon Clipping

Input: Reference Face Query, **Incident Face Query**
Output: A collection of **points** clipped by the reference side faces

```
1  for each side face of the reference box do
2      Set the first vertex of the incident face as the start vertex
3      while there are still vertices of the incident face to iterate do
4          Set the next vertex of the incident face as the end vertex
5          Compute the signed distance of the start vertex
6          Compute the signed distance of the end vertex
7          if the signed distance of start vertex is positive and the signed distance
             of the end vertex is negative then
8              Store the end vertex and the linear interpolation the intersection
                 of the edge with the clipping plane
9          else if the signed distances of both the start vertex and the end vertex
             are negative then
10             Store only the end vertex
11         else if the signed distance of the start vertex is negative and the
             signed distance of the end vertex is positive then
12             Store the linear interpolation of the intersection of the edge with
                 the clipping plane
13         Set the start vertex to be the end vertex
```

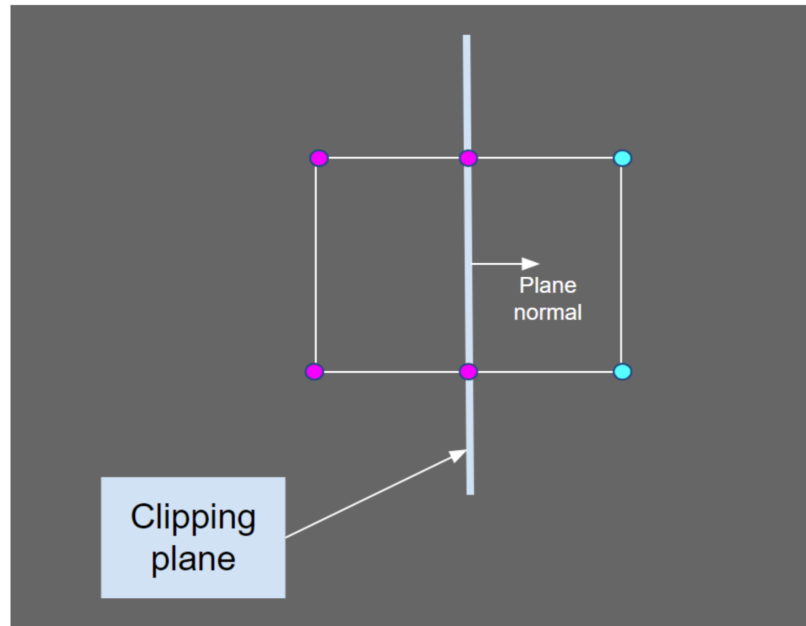

Figure 3.8: Sutherman-Hodgeman Polygon Clipping

After the clipping algorithm, we should have a collection of contact points. We can use that to fill out an information struct called the Contact Manifold. It should have all the information about the collision between each pair of Box Collider. An example of it can be:

```
struct ContactManifold
{
    int referenceBoxID;
    int incidentBoxID;
    int contactCount;
    Contact contacts[MAX_CONTACT_POINT_COUNT];
    vec3 contactNormal;
};
```

where the Contact struct can look like this:

```
struct Contact
{
    vec3 position;
```

```
    vec3 normal;
    vec3 referenceRelativePosition;
    vec3 incidentRelativePosition;
};
```

and MAX_CONTACT_POINT_COUNT can be set to whatever you want, I have it set to 16; it would not be matter much because we shall reduce the contact point number down to 4 in the next optimization step. But before we do that, we must project all the contact points onto the reference face. It is just a simple algorithm to project a point in space onto a specific plane. This would make all the contact points to be "flat," which then turn the next step to be a bit more easier since it is all about finding the collection of contact points that would cover the largest area.

3.3.4 Contact Point Reduction

This only needs to be employed when we have faces as colliding features. The reason why we have to do this is we only need 4 contact points per colliding pair for a stable contact manifold, having more than that would just make the solver work harder for no good reason. Note that this algorithm 8 only is being used when there are more than 4 contact points.

Algorithm 8: Contact Point reduction

Input: A **Contact Manifold**
Output: None

1. For the first point, find the support point in the direction of the contact normal
2. For the second point, find the contact point that is farthest away from the first point
3. For the third point, find the contact point that would form a triangle with the greatest area with the first 2 points
4. For the fourth point, find the contact point that would form a rectangle with the greatest area

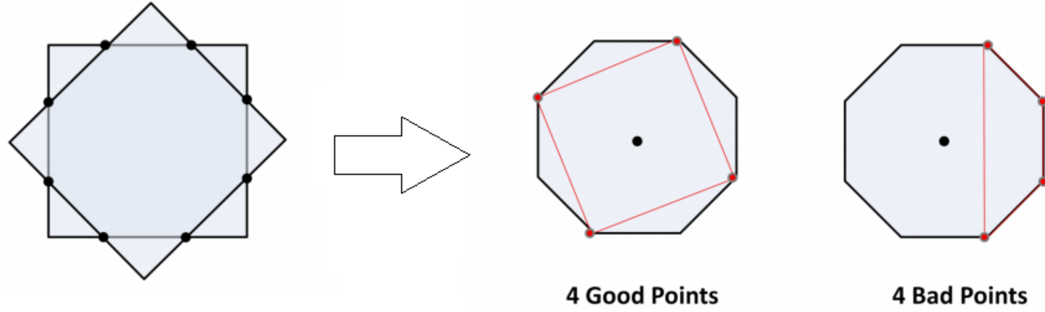

Figure 3.9: Contact reduction. From: [6]

In figure 3.9, this is the top down view for 2 boxes on top of each other. The points in black on the left are the contact points, but we do not need all of them. On the right, there are examples of a stable group of contact and an unstable group of contact points. Algorithm 8 essentially want to get the collection of contact points that would produce the largest area.

This algorithm assumes that all the contact points are in the same plane. This is a safe assumption because we have projected all the contact points onto the reference face after the Sutherman-Hodgeman polygon clipping.

3.4 Resolution Phase with Sequential Impulses

So now we have the contact points information from the Narrow Phase, we then need to figure out what to do with them so resolve the collisions. A collision is resolved when the 2 colliding bodies are no longer overlapping in space, and friction force is correctly simulated. In the layman's terms, we essentially answer 2 questions for a particular rigid body: (1) Where is it going, and (2) How fast is it spinning.

3.4.1 Resolving Overlapping with Contact Constraint along the Contact Normal

To formulate the physics concepts to mathematical terms, constraints are needed. A constraint C is defined as a function of positions, in our case since 2 bodies should not be overlapping, the constraint would be a function of 2 positions, of body A and of body B, in particular their points of deepest penetration P_A and P_B [2]:

$$C : (\overrightarrow{P_B} - \overrightarrow{P_A}) \cdot \vec{n} \geq 0 \tag{3.1}$$

This means in world space, their relative position should be greater or equal to zero. This is often called a contact constraint, and typically, a constraint is satisfied when it is equal to zero. When a contact constraint is violated, we need to fix it by applying some change to the position, ΔP. But, when applying a change of position onto each body would make its movement jittery. In order to make the correction movement to be smooth, we apply a change in velocity, ΔV, instead. In order to do that, we take the derivative of the contact constraint with respect to time, $\dot{C}$, to get the velocity constraint [3][2]:

$$\begin{aligned}
&C : (\overrightarrow{P_B} - \overrightarrow{P_A}) \cdot \hat{n} \geq 0 \\
&C : (\overrightarrow{CM_B} + \overrightarrow{r_B} - \overrightarrow{CM_A} - \overrightarrow{r_A}) \cdot \hat{n} \geq 0 \\
&\dot{C} : (-\overrightarrow{V_A} - \overrightarrow{\omega_A} \times \overrightarrow{r_A} + \overrightarrow{V_B} + \overrightarrow{\omega_B} \times \overrightarrow{r_B}) \cdot \hat{n} \geq 0
\end{aligned} \tag{3.2}$$

where $\hat{n}$ is the unit normal vector, $\overrightarrow{CM_A}$ and $\overrightarrow{CM_B}$ are the centers of mass of body A and body B respectively, $\overrightarrow{r_A}$ and $\overrightarrow{r_B}$ are the relative positions of the contact points relative to the body centers of mass, $\overrightarrow{\omega_A}$ and $\overrightarrow{\omega_B}$ are the angular velocities. After

simplifying and bringing it to the form of:

$$\dot{C} : JV + b = 0 \tag{3.3}$$

where J is the Jacobian and defined to be a 1-by-12 matrix (a row matrix):

$$J = \begin{bmatrix} \overrightarrow{J_{V_A}}^T & \overrightarrow{J_{\omega_A}}^T & \overrightarrow{J_{V_B}}^T & \overrightarrow{J_{\omega_B}}^T \end{bmatrix} \tag{3.4}$$

and the velocity matrix, V, is defined to be a 12-by-1 matrix (a column matrix):

$$V = \begin{bmatrix} \overrightarrow{V_A} \\ \overrightarrow{\omega_A} \\ \overrightarrow{V_B} \\ \overrightarrow{\omega_B} \end{bmatrix} \tag{3.5}$$

Finally, b is our bias term. More on this later, but essentially, its function is to make the simulation more stable.

By rearranging 3.2 to the form of equation 3.3, we are essentially saying that the velocity constraint is a linear combination of the Jacobian and the velocities, both linear and angular.

The Jacobian of the normal component is:

$$J_{\hat{n}} = \begin{bmatrix} -\hat{n}^T & (-\overrightarrow{r_A} \times \hat{n})^T & \hat{n}^T & (\overrightarrow{r_B} \times \hat{n})^T \end{bmatrix} \tag{3.6}$$

Now that the Jacobian is defined, what can we do with it? Again, the ultimate goal is to find the ΔV to apply to current V so that the constraint is satisfied, that means:

$$J(V + \Delta V) + b = 0 \tag{3.7}$$

The important thing to understand here is that ΔV is directly proportional to the product $M^{-1}J^T$, where M is the mass matrix:

$$M = \begin{bmatrix} M_A & 0 & 0 & 0 \\ 0 & I_A & 0 & 0 \\ 0 & 0 & M_B & 0 \\ 0 & 0 & 0 & I_B \end{bmatrix} \tag{3.8}$$

$$\tag{3.9}$$

$$M^{-1} = \begin{bmatrix} M_A^{-1} & 0 & 0 & 0 \\ 0 & I_A^{-1} & 0 & 0 \\ 0 & 0 & M_B^{-1} & 0 \\ 0 & 0 & 0 & I_B^{-1} \end{bmatrix} \tag{3.10}$$

$$\tag{3.11}$$

$$M_A = \begin{bmatrix} m_A & 0 & 0 \\ 0 & m_A & 0 \\ 0 & 0 & m_A \end{bmatrix}, \; M_B = \begin{bmatrix} m_B & 0 & 0 \\ 0 & m_B & 0 \\ 0 & 0 & m_B \end{bmatrix} \tag{3.12}$$

To make the relation between ΔV and $M^{-1}J^T$ an equation, we need a “correction” factor called the Lagrangian multiplier, λ:

$$\Delta V = M^{-1}J^T\lambda \tag{3.13}$$

Plug 3.13 to 3.7, then solve for λ:

$$J(V + M^{-1}J^T\lambda) + b = 0$$
$$\lambda = (JM^{-1}J^T)^{-1}(-JV - b) \tag{3.14}$$

The left product is sometimes called the effective mass:

$$M_{eff} = JM^{-1}J^T \tag{3.15}$$

The last thing to define is the bias term b. So far all the math we have to do is simply to find the ΔV so that the relative velocities of the 2 bodies along the contact normal (hence the dot product with the normal unit vector) is zero. That simply stops the penetration from increasing. But that is not sufficient to resolve the collision. The 2 colliding bodies must also move away from each other. Since we use the velocity constraint by taking the time derivative of the position constraint, the information of the position is lost, i.e the separating vector from A to B: $(\overrightarrow{P_B} - \overrightarrow{P_A})$. We need to somehow reintroduce the information of the penetration depth into the contact constraint; and we do this by using a bias b, which scales with the penetration depth:

$$d = (\overrightarrow{P_B} - \overrightarrow{P_A}) \cdot \hat{n} \tag{3.16}$$
$$b = -\frac{\beta}{\Delta t} \cdot d \tag{3.17}$$

where β is the Baumgarte stabilization parameter, and it can have value from 0 to 1.

3.4.2 Simulating Friction with Contact Constraint along the Contact Tangents

In 3D, 3 unit vectors are needed to form an orthonormal basis, which is a new frame of reference if you will. Our first unit vector is the normalized contact normal from the Narrow Phase. Our new goal is to define the other 2 unit vectors $\hat{t_1}$ and $\hat{t_2}$, which are tangent to the contact surface, then compute the 2 tangent Jacobians. Similar to the normal component of the Jacobian 3.6, the two tangent components of the Jacobian are defined as:

$$J_{\hat{t_1}} = \begin{bmatrix} -\hat{t_1}^T & (-\overrightarrow{r_A} \times \hat{t_1})^T & \hat{t_1}^T & (\overrightarrow{r_B} \times \hat{t_1})^T \end{bmatrix} \tag{3.18}$$

$$J_{\hat{t_2}} = \begin{bmatrix} -\hat{t_2}^T & (-\overrightarrow{r_A} \times \hat{t_2})^T & \hat{t_2}^T & (\overrightarrow{r_B} \times \hat{t_2})^T \end{bmatrix} \tag{3.19}$$

The geometric meaning of the 2 tangent Jacobians is to zero out the 2 tangent components of the relative velocities of the 2 bodies, which is physically what friction does; friction force can only act upon the tangent (again, to the contact surface) components of object's acceleration vector.

3.4.3 Solving for λ

The goal for Sequential Impulses is to solve for λ; once λ is found, it can be used to find the change in linear velocites and angular velocites (as described by equation 3.13) to satisfied all the constaints along the contact normal (to prevent penetration) and contact tangents (to simulate frictions). Solving for a global solution of λ is solving a system of equations, and that takes a long time. In order to make the calculations real-time, approximation for the global solution would be done instead. Sequential Impulses would keep applying small impulses to satisfied the system of equation when approximating λ. This mean Sequential Impulses has to do this in

many iterations. The more iterations Sequential Impulses being executed the more accurate the approximation is, as you would see in the results section.

3.5 OpenGL Comes into Play

All the basics of a physics engine are discussed already, it is time to parallelize those algorithms. Normally on the CPU, to repeat one calculation over and over this is what you do:

```
for (int i = 0; i < (some max limit); ++i)
{
    // Do something
}
```

Of course, there are threadings on the CPU where to can repeat the calculation all at once. But the concept is similar to the GPU, and the GPU has way more compute threads than CPU threads. Here is how to repeat the same calculation on the GPU with OpenGL Compute Shader:

```
uint i = gl_GlobalInvocationID.x;
// Do something
```

The description of iteration is stored within i. What do we iterative over is described by i. On the GPU, we use the compute thread ID (obtained from gl_GlobalInvocation OpenGL built-in struct) as i, and it can range from 0 to 1024 (or with some set up it can range from 0 to 65535 or even more). So in each physics pipeline stage, I will show what i represents.

For Broad Phase, shown in 3.10, i first represents each BoxCollider object to calculate all the AABBs. Then i represents each AABB used for the sweep step of the Sweep and Prune algorithm to get all the CollisionPairs.

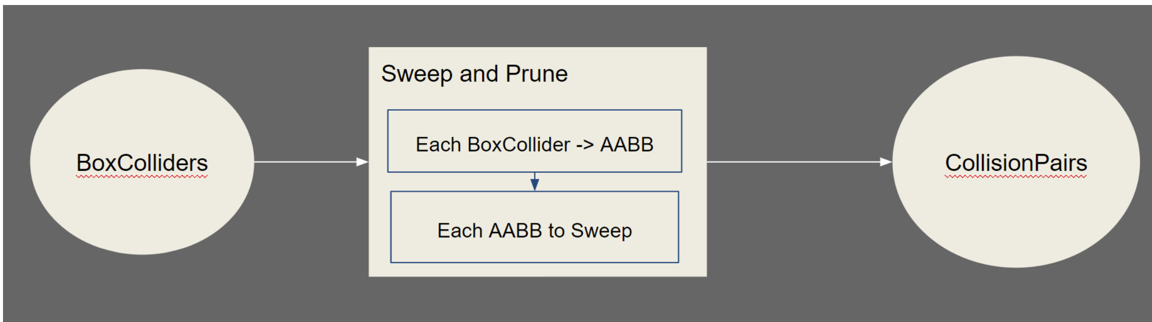

Figure 3.10: Parallel Broad Phase

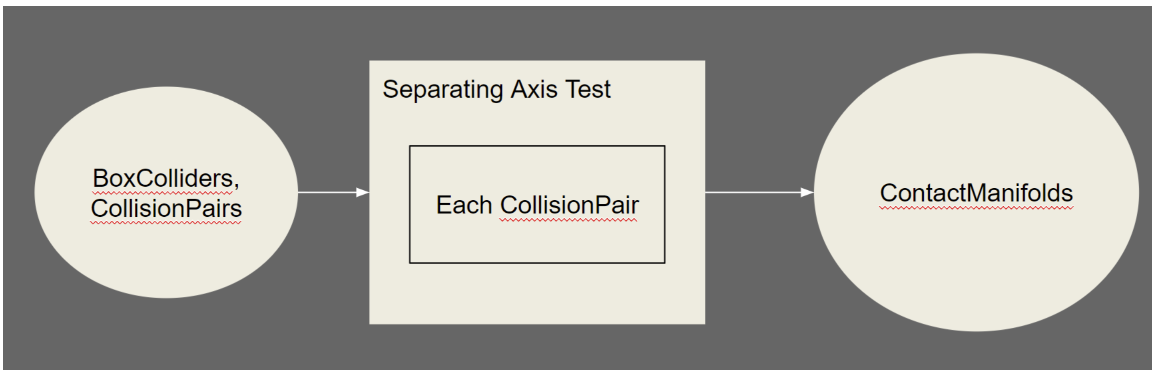

Figure 3.11: Parallel Narrow Phase

For Narrow Phase, shown in 3.11, i represents each CollisionPair to build all the ContactManifolds.

Now for the Resolution Phase, particularly Sequential Impulses algorithm, things get a bit rough. Sequential Impulses is sequential in nature, hence the name. That means colliding pairs described in the ContactManifolds must be processed one after the other, in no particular order however. If there is any attempt to parallelize it, data will be tampered, and results will be wrong. To show my point, let's look at the scenario in figure 3.12:

What you see here is 4 bodies in collision, and the solver would have to go through each pair and apply the correction impulses to each of them. Let's treat the Sequential Impulses solver as a black box, and for each pair that needs to be solved, there are initial conditions, i.e. initial linear velocity and initial angular velocity; if we feed the

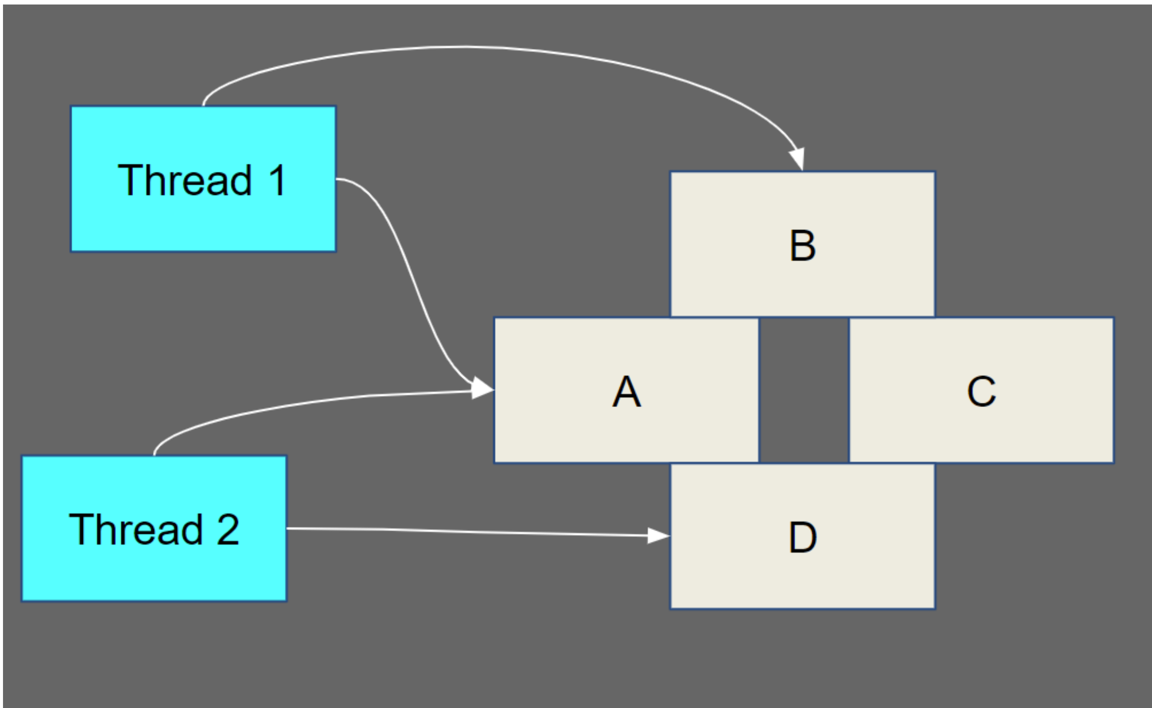

Figure 3.12: Parallel Sequential Impulses

initial conditions to the black box, it will produce the final conditions, i.e. the final linear velocity and final angular velocity, which we need. With parallelism, you cannot guarantee the ordering of which the compute threads starting or finishing without paying a price, if you force guarantee an ordering then you would lose performance. But the main problem here is that thread 2 might have already finished and applied the changes to body A, while thread 1 is sill calculating the changes for body A. That means the initial conditions of body A that thread 1 uses is no longer valid. This would end up producing the wrong results. We can remedy this my forcing GPU to do the calculation sequentially, i.e. increment the i on the CPU instead then send it to the GPU. There would be absolutely no parallelization, and the performance is reduced tremendously as will be shown in the results section.

Chapter 4

OPENGL AND OPTIMIZATION

On a high level, to gain high performance, whether in games or simulations, the round trip of data transferring between the CPU and GPU must be minimized. Since the physics engine must execute its implementations tremendously frequent, this potential issue is exacerbated greatly.

The easiest thing to do is to submit the initial conditions of the dynamics world to the GPU once and once only, then every update loop will be processed from the GPU side. However, that is not always possible. For instance, a game engine update loop probably has more than just the physics update; it has to process the game logic as well depending on what are colliding in the game world - imagine when the controllable character is in a certain zone to receive healing, initiate a cutscene, or active a switch. These logic are probably more efficiently processed on the CPU side as the GPU cannot handle branching quite as well.

Another optimization possibility would be using the same API for both rendering and computing. Since this engine is implemented with OpenGL, if the renderer also uses OpenGL, then there should be a seamless integration between the 2 systems. For OpenGL to work, an OpenGL context must be created. This OpenGL context can be shared by both the renderer and the physics engine, providing that they are on the same thread - which is a limitation of OpenGL; Multi-threading and OpenGL do not go well together. This means when the physics engine update the physics tick on the GPU, the renderer will have access the updated data - positions, velocities, momenta, etc, assuming the data coherency is corrected set up. Again, this is not

always possible because there are many other graphics APIs that the renderer can use, such as Vulkan, DirectX, Metal, etc, and trying accommodate seamless integrations with all of those graphics APIs is beyond the scope of this thesis.

4.1 OpenGL Asynchronous Nature

It is important to know and understand about the asynchronous nature of OpenGL for one crucial reason: to not tamper with any data already in flight. The abstraction from the implementation of OpenGL driver completely mystifies the process going on under the hood. Once an OpenGL call is issued from the CPU, the call probably will not be completed until much later. The reason for this is each OpenGL call potentially has a huge overhead due to it can only be submitted to the GPU during the kernel mode, and switching between modes is an extremely expensive operation, so the driver would rather queue up all the OpenGL calls issued from the CPU to a command buffer and submit them all at once during kernel mode. This asynchronous nature leads to a massive synchronization issue. Typically, the OpenGL driver should be smart enough to force some certain sync points between the CPU and GPU so that the data will not be tampered with during later on OpenGL calls. However, forcing these sync points will block, leading to tremendous performance loss.

With that being said, there are 3 stages in this physics engine that are sequential. The Narrow Phase must wait for the Broad Phase to finish all the computations, and the Collision Resolution Phase must wait for the Narrow Phase to finish. To guarantee this sequence, some performance must be sacrificed, with the use of glFinish(). glFinish() would halt the CPU thread until the GPU has finished all of the computing commands. There are better techniques to handle this situation such as

triple buffering [1] [5]. However, doing so would increase the memory cost threefold; so to keep the engine light-weight, I have decided to not implement the feature.

4.2 Optimization in Data Streaming

This topic has to be mentioned because the way this engine works, there are a huge amount of data being moved from the CPU to the GPU, i.e. in the case when uploading the initial conditions of the bodies to for the Broad Phase, and then from the GPU back to the CPU, i.e. to check when 2 bodies are colliding for some simulation logic check so only the information of the CollisionPairs are needed from the Broad Phase. Furthermore, the simulation is done in an infinite loop, so if you want to constantly moving the data back and forth from the CPU to the GPU. Traditionally for OpenGL, to get the data from the GPU, you would have to map the chunk of memory on the GPU that you need to a temporary memory region on the CPU with the call to glMapBuffer, called the pinned memory[5]. From then on, the CPU would get a pointer to access the data. When done, you would have to unmap the memory region with glUnmapBuffer, else you would get critical error when trying to use an unmapped buffer for any other GPU operations. This is quite slow because there might be some syncing that OpenGL driver might be doing under the hood [5]. The problem gets exacerbated even more when you do this constantly in the simulation loop. The solution to this was shown by various graphics engineers - Cass Everitt, Tim Foley, Graham Sellers, John McDonald - at Game Developers Conference in 2014 [1], and it is persistent mapping. The idea is since we need to map the data from the GPU so often, we might as well map it once and keep it mapped forever; or until we unmap it. This idea only possible in the later versions of OpenGL, particularly 4.4, where buffers can be allocated with glBufferStorage. Buffers when created with glBufferStorage would have immutable data store, meaning its size cannot be changed

by any mean, and they can be mapped forever by calling glMapBufferRange with the correct mapping flag bits set:

```
glGenBuffers(1, &bufferID);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, bufferID);
glBufferStorage(GL_SHADER_STORAGE_BUFFER, (some size in bytes),
    nullptr, createFlags);
pointerToPinnedMemory = static_cast<SomeStruct *>(glMapBufferRange(
  GL_SHADER_STORAGE_BUFFER,
  0,
  (some size in bytes that you want to map to pinned memory),
  mapFlags
));
```

pointerToPinnedMemory is the pointer that the CPU can access to read from or write to, providing that the creating flag bits (createFlags in the code above) and mapping flag bits (mapFlags in the code above) are set up correctly. These are the bits that I use quite often for almost all of the buffers in this thesis.

```
GLbitfield mapFlags = GL_MAP_READ_BIT
          | GL_MAP_WRITE_BIT
          | GL_MAP_PERSISTENT_BIT
          | GL_MAP_COHERENT_BIT;

GLbitfield createFlags = mapFlags | GL_DYNAMIC_STORAGE_BIT;
```

GL_MAP_READ_BIT and GL_MAP_WRITE_BIT allow the CPU to read and write with the pinned memory pointer. GL_MAP_PERSISTENT_BIT tells OpenGL driver to map the GPU memory region until it got unmapped.
GL_MAP_COHERENT_BIT simply asks OpenGL driver if there is any change in the buffer done by the GPU, it will let the CPU see the results. Lastly, the creating flag

bits are similar to the mapping flags, but it needs to have GL_DYNAMIC_STORAGE_BIT if you plan on updating the data of the buffer with glBufferSubData.

Chapter 5

RESULTS

The engine was tested on Nvidia RTX 3070 (the GPU) and AMD Ryzen 5600X (the CPU). In the demos shown below, the blue boxes are the static bodies, which are not affected by gravity, and the green boxes are the rigid bodies, which are subjected to the downward (-y direction) acceleration from gravity. The collision detection was turned on for both kind of bodies. Kinematic bodies are not shown because it is just a special kind of rigid body. The engine exposes the linear transforms, which includes linear velocities and positions, and angular transforms, which include orientations and angular velocities, of all kinds of bodies so the users can always manipulate them on their end. They would have to handle movement inputs and then change the transforms accordingly. It is not the responsibility of the engine to handle that.

Stacking is one of the most fundamental thing to test the stability of an physics engine, mostly the solver of course. If the engine can handle tall stack, then it is a decent engine; although for games, this might not be relevant, as not a lot of games have many things stacking on each other. The taller the stack the more difficult for the engine to keep it stable. In the figure 5.1, some basic stacking is shown. The next 2 figures, 5.2 and 5.3, are just some twists of the stacking scenario. For all of these demos, the performances are superb when the solver is on the CPU, and rightfully so since it is sequential, with the collision detection on the GPU. To be specific, the framerate is around 2500 frames per second (fps) with high solver iteration count (at 150 iterations). When the full physics pipeline is on the GPU, the performance tanks down to only 30 to 60 fps with low solver iteration count (at 5 iterations). Due to

low iterations, the physics simulation tends to be unstable when the solver is on the GPU.

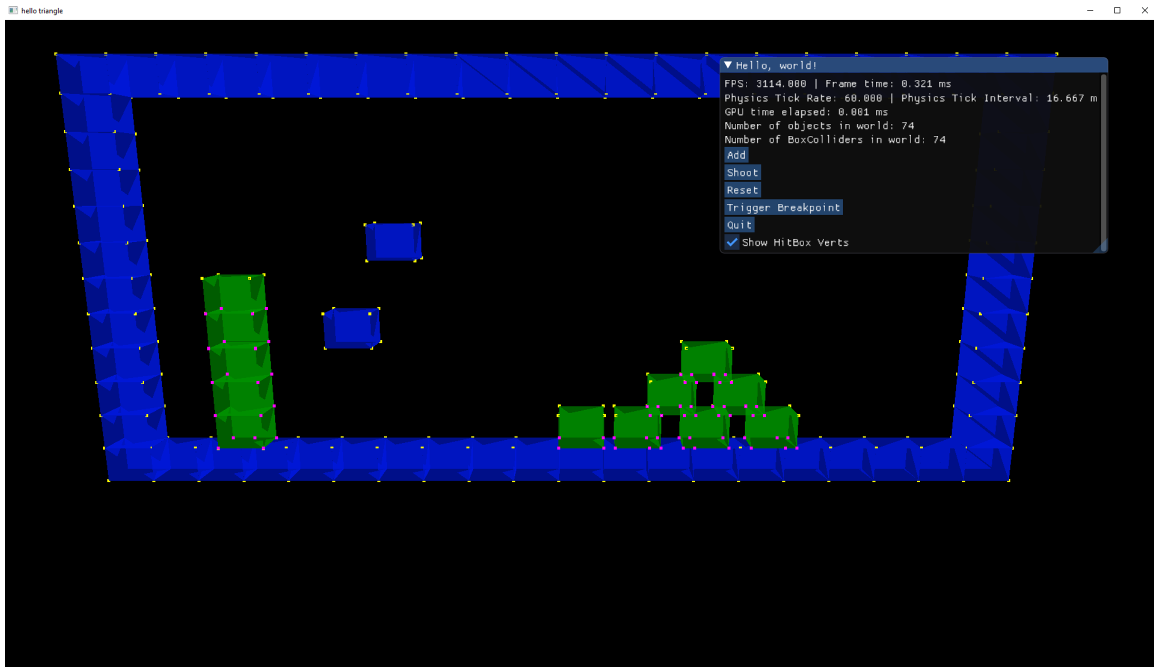

Figure 5.1: Simple stacking

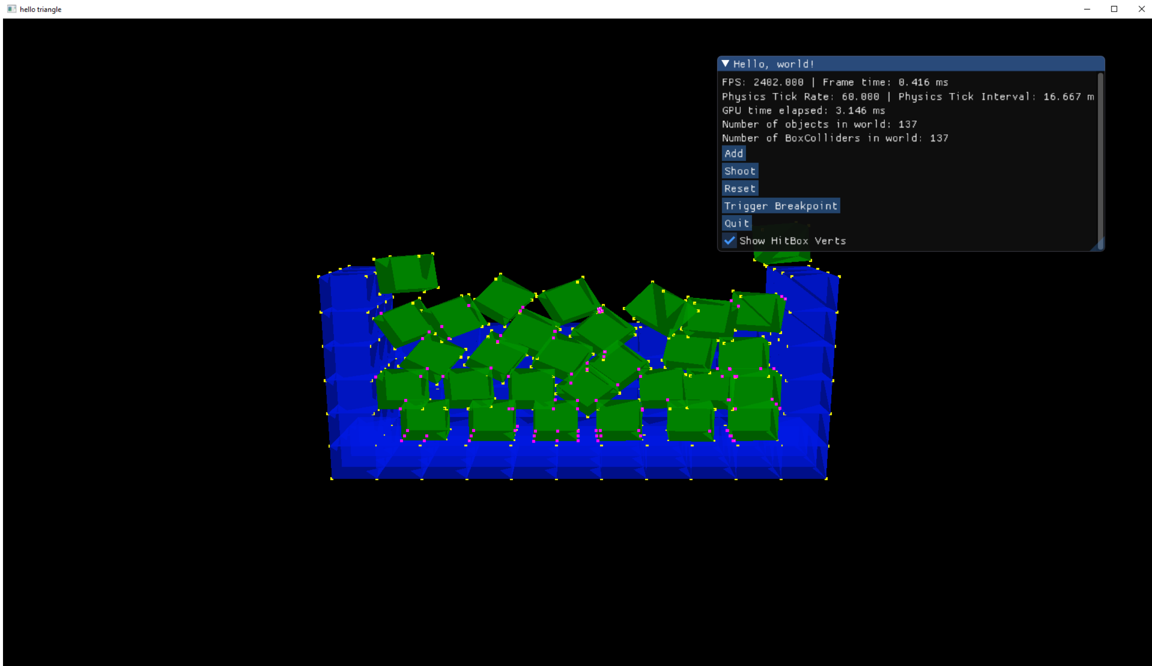

Figure 5.2: Pool of boxes

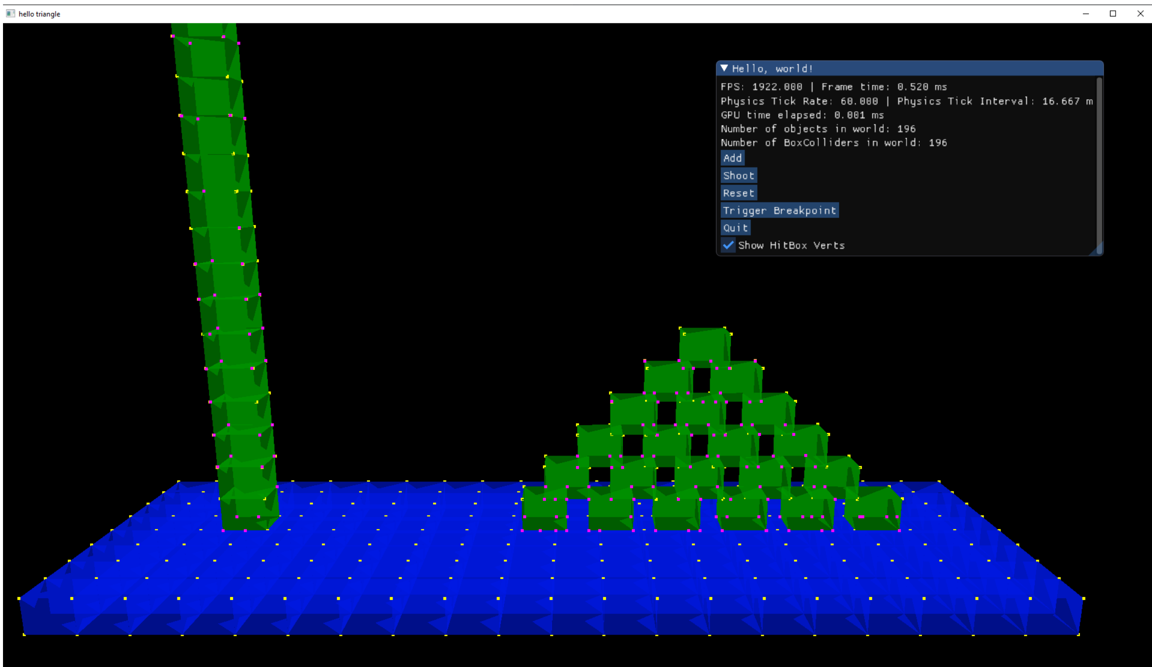

Figure 5.3: Extreme stacking

Chapter 6

CONCLUSION

It is possible to implement the physics pipeline fully on the GPU with the help of OpenGL Compute Shaders. However, for the best performance, the solver must be on the CPU because it is sequential. One possible solution to solve this issue is to batch the Manifolds with the pairs that are unique from each other. For instance as mentioned in section 3.5, we first only send to the Compute Shader pairs (A, B) and (C, D), wait for those to be finished with the call to glFinish(); then we send pairs (A, D) and (B, C) to the Compute Shader and wait for them to be finished with glFinish(). We still have to process things sequentially with Compute Shader for the most part, but this time we can process some pairs (that are not related to each other - if the any 2 pairs share one similar body, then they are related) in parallel. However, due to time constraint, this method did not get tested.

BIBLIOGRAPHY

[1] Gdc 2014: Approaching zero driver overhead (azdo) in opengl, Feb 2017.

[2] E. Catto. Physics for game programmers: Understanding constraints, Mar 2018.

[3] M.-L. Chou. Game physics: Resolution – constraints sequential impulse, Dec 2013.

[4] C. Ericson. *Real-time collision detection*. Elsevier.

[5] B. Filipek. Persistent mapped buffers in opengl, Jan 2015.

[6] D. Gregorius. Robust contact creation for physics simulations.

[7] N. Souto. Video game physics tutorial - part ii: Collision detection for solid objects, Mar 2015.