Collision Detection of Triangle Meshes using GPU

Nils Bäckman

March 13, 2011
Master's Thesis in Computing Science, 30 ECTS credits
Supervisor at CS-UmU: Daniel Sjölie
Examiner: Fredrik Georgsson

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

Collision detection in physics engines often use primitives such as spheres and boxes since collisions between these objects are straightforward to compute. More complicated objects can then be modeled using compounds of these simpler primitives.

However, in the pursuit of making it easier to construct and simulate complicated objects, triangle meshes are a good alternative since it is usually the format used by modeling tools.

This thesis demonstrates how triangle meshes can be used directly as collision objects within a physics engine. The collision detection is done using triangle mesh models with tests accelerated using a tree-based bounding volume hierarchy structure.

OpenCL is a new open industry framework for writing programs on heterogeneous platforms, including highly parallel platforms such as Graphics Processing Units(GPUs). Through the use of OpenCL, parallelization of triangle mesh collision detection is implemented for the GPU, then evaluated and compared to the CPU implementation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Objects such as spheres, boxes and planes are easily described mathematically in three dimensions. They are also straightforward, if not always simple, to test for collision.

To simulate and test more complex objects for collision, there are numerous ways to represent objects and use these representations.

One way is to build a more complex object of above mentioned geometric primitives put together, commonly called compounds. A car can for example be modeled with spheres as wheels and a box can form the body. This is an efficient and simple solution, but it can take time to build more complex objects in this way. It is also nearly impossible to create arbitrary objects, as round sides are bound to be spheric and intricate objects require many primitives to keep the detail of the object.

There are also efficient algorithms to determine collision between convex objects. Any non-convex object can be subdivided into convex parts, making this another alternative for complex objects. However, these algorithms have some artifacts that can make physical simulations jittery when they ideally should come to a rest.

Other implementations take advantage of a more mathematical representation, such as using splines or being able to add, subtract and take the union of simple mathematical shapes. These are very good for designing models and visualizing them, but determining collision and appropriate reaction are difficult.

Using triangle meshes for collision detection and testing every triangle of one object for collision against every triangle of the other object will be the focus of this thesis. With an appropriate first-step approximate collision method, such as a tree-based structure, the implementation scales well with the complexity of the objects, potentially making it fast enough for use in real-time environments.

General collision detection can be divided into two steps, intersection test and intersection find. An intersection test answers the question whether two objects are in collision or not. For a physics simulation, only intersection test is usually not enough, because to simulate the collision we need to know more about it. This is the more complex part, commonly called intersection find or contact generation. A collision find algorithm returns the depth, point and normal of a collision. The contact normal is the vector that specifies in which direction the two objects should be separated. The depth is the minimal distance to move one of the two objects along the contact normal to separate them. The contact point specifies a point where the two objects are intersecting. An intersection find algorithm should in many cases return more than one contact to ensure a stable simulation.

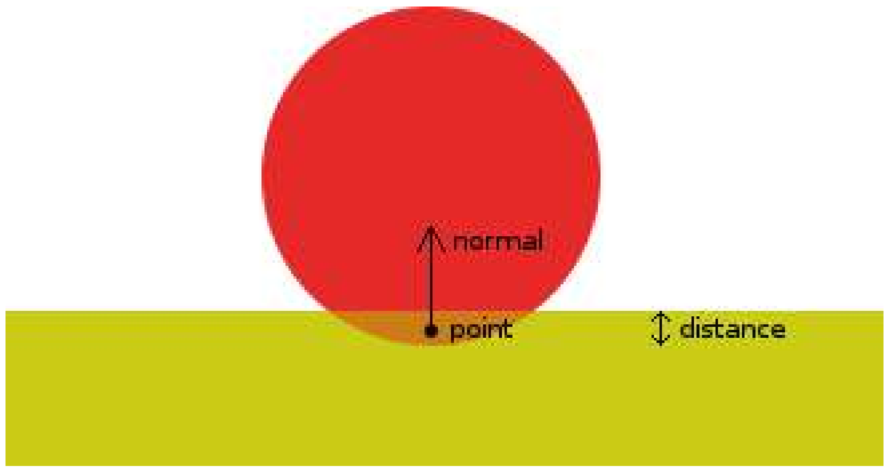

Figure 1.1: *Collision of a sphere into a plane, resulting in a normal vector, a contact point and a scalar depth value.*

1.1 Algoryx Simulation

The thesis was done at Algoryx Simulation, a company from Umeå, Sweden, whose main product is a physics engine named AgX Multiphysics[2], geared towards use in scientific research and training simulators. For these use cases, as opposed to game physics engines, physical correctness of the simulations is vital.

As more complicated shapes in AgX Multiphysics today are constructed using compounds of simple primitives such as spheres, boxes and cylinders, Algoryx were very interested in examining the possibility of using triangle meshes as collision primitives for their physics engine.

1.2 Problem Statement

The purpose of this project was to implement, test and evaluate different algorithms and data structures used in triangle mesh collision detection with regard to efficiency and stability.

Implementing and evaluating triangle mesh collision detection on GPUs using OpenCL was another important aspect. As GPUs nowadays are getting more and more advanced, with better and better capabilities for performing general programming, evaluating whether the massively parallel architecture of GPUs could lead to speedups versus a CPU implementation was an important aspect.

Like any algorithm that is to be used in a general environment, it is important that it meets certain requirements on stability, robustness, memory efficiency and scalability.

No data should cause the algorithm to hang, cause infinite loops or return invalid data. This not only means that the algorithm should be stable and robust, but also that the validity of the data used by the algorithm should be confirmed. Verification of a valid mesh should thus be covered in the project.

1.3 Thesis Report Outline

The layout of the report is such that theory applicable to the thesis work is covered first, followed by implementation details, and lastly the results of the implementation is discussed.

Chapter 2 covers background information such as the general structure of a physics engine.

In the following chapters 3 and 4 theory and knowledge needed to understand the implementation are described.

Chapter 3 covers collision detection specifics, such as different ways to implement middle phase collision detection and an important theorem often used for collision detection of convex objects (such as triangles).

In chapter 4 general purpose programming on GPUs is discussed. An introduction to OpenCL follows, describing its uses as well as the programming layout. One section covers optimization, a very important aspect for achieving satisfying results while programming for GPUs.

Implementation details are covered in chapter 5. The chapter is further divided into one section for the CPU implementation and one for the GPU implementation.

In chapter 6 the results of the implementation are discussed. Here various performance graphs as well as other important observations can be found.

Finally, chapter 7 contains conclusions and thoughts on the project, such as restrictions, limitations and future work.

Chapter 2

Background

2.1 Physics Engine Pipeline

Collision detection is part of the physics engine pipeline. Here follows a brief description of such a pipeline in order to describe in which context collision detection is used.

Collision detection between objects in a scene needs to determine whether objects collide with each other, and if they do, give information that can be used in the next step to resolve the collision.

When the collision detection step is done, it is time to solve the collisions for the next iteration. Here it is common to apply forces to the objects in order to solve the collision overlaps found. The forces are contact forces based on the information from the collision detection as well as other forces, e.g. gravity.

The last step is time integration, where the simulation is moved to the next discrete time step (assuming discrete time integration). Objects properties, such as position, velocity and rotation, are updated to reflect their new states.

When this is done the simulation has moved from one time step to the next. For a continued simulation, these steps are repeated over and over again.

2.2 Collision Detection Pipeline

For n objects in a scene, determining collisions between all these objects against each other means $O(n^2)$ separate collision checks.

Because of this bad complexity, a first step of coarse but fast tests are usually executed, returning objects that are close but not necessarily colliding. This step is usually called broad phase collision detection and can be heavily accelerated because of temporal coherence (or in other words, that objects are not likely to move very far in little time).

Now that several obviously non-colliding pairs have been pruned away, a precise collision detection routine can be executed. This is usually called near phase collision detection.

This thesis will focus on near phase collision detection as that is where triangle meshes differ from other collision primitives.

2.2.1 Continuous versus Discrete Collision Detection

The most widely used way of simulating physics is to discretize time by dividing it in time steps. Objects are tested for collision at every time step and, if they intersect, actions are

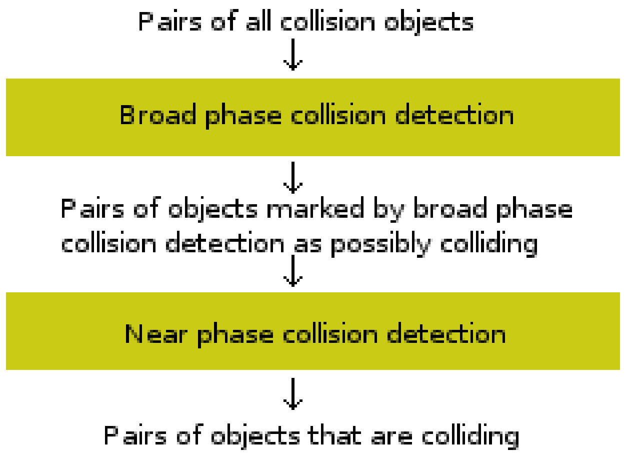
Pairs of all collision objects
Broad phase collision detection
Pairs of objects marked by broad phase collision detection as possibly colliding
Near phase collision detection
Pairs of objects that are colliding

Figure 2.1: *An outline of what a collision detection pipeline might look like.*

taken to solve the intersection. This means that in between time steps, objects can ”tunnel” through each other without a collision being noticed. This puts restraints on velocity, object size and time step size, as small object sizes and/or large velocities and time steps easily cause the tunneling effect. Because of the discretization of time, collisions between objects can be deep when they are found. This can cause problems, as knowing where the initial collision of the objects was is not always clear.

Continuous Collision Detection tries to solve these problems by finding the exact time of impact between two objects colliding[27]. This means each object’s trajectories are calculated, and collisions are found just as they happen, enabling simulations of non-penetrating objects. Finding the time of impact is time-consuming, and the more objects there are in a scene, the more difficult it is to run a non-penetrating simulation in real-time. Because of this, continuous models are often abandoned for discrete ones.

2.3 General Polygon Meshes

Triangles have the benefits of always being convex and always defining a plane. Because of this, intersection tests between triangles are straightforward and inexpensive to compute.

Conversion from general polygon meshes to triangle meshes is simple and can be done as a pre-processing step by modeling software such as Blender[1]. For convex objects, triangulation can trivially be done by creating edges from one vertex to all other vertices. For concave objects, it is not as trivial but still straight-forward. Different algorithms exist, but the one perhaps most easily understood is called the Subtracting Ears Method[7]. By noticing that a polygon always contains a triangle with two edges being edges of the polygon, this triangle can be removed from the polygon, which will still contain at least one such ”ear”. By iterating over the polygon in this manner, it will eventually be fully triangulated.

Chapter 3

Collision Detection for Triangle Meshes

For simple objects such as spheres and boxes, collision detection is straightforward. Representation of a box can differ in size and rotation, but its corners always have the same angles and it always has six faces. A triangle mesh on the other hand can consist of thousands of faces or just a few, and it can be both convex and concave. Still the algorithm is expected to be generic and handle all cases in an efficient manner.

This chapter covers areas in collision detection that are of interest both for general collision detection and for collision of triangle meshes.

3.1 Complex Object Collision Detection Techniques

For objects of complex structure there are different techniques to solving collisions. Complex objects, as opposed to simple objects such as spheres, boxes and planes, can adopt many different forms. They can be described as triangle meshes, but also by e.g. splines or general polygon meshes instead of only triangles. The techniques to solve collision of these objects differ vastly, and the reason so many have been developed is likely because there is not yet a perfect solution that takes care of everything. Below, the different techniques to solve collisions of complex objects are covered and their advantages and disadvantages are discussed.

3.1.1 Triangle-Triangle Collision Detection

For triangle mesh objects, one can use the fact that they consist of triangles by testing each triangle of one object against every triangle of the other object. If two triangles, one from each object, intersect we know that the two objects are in collision.

As a brute force solution of testing every triangle n of one object against the all triangles m of another object are of complexity $O(nm)$ and scale extremely badly for more detailed meshes, objects can be divided in a tree hierarchy to cull non-colliding triangles faster. More on these hierarchies can be found in Section 3.3 of this chapter.

As physics engines using a collision detection library generally are interested in simulating the collision (as opposed to e.g. removing an object if collision is true), contact information must also be supplied. Because of this, we cannot stop the computation when two triangles are found to be colliding, but we must find all colliding triangles. From these triangles, contact information is gathered for the physics engine to use in its simulation.

3.1.2 Signed Distance Fields

A signed distance field is a grid representation of an object. Voxels in the grid (or pixels in case of two dimensions) are estimates of how far away from the surface of the object the voxels are located. A negative value means the voxel is inside the object, and a positive sign means it is outside.

The grid representation can cause flat surfaces to be aliased because of the static positions of each voxel, and the amount of aliasing depends on the resolution of the grid.

When colliding two objects represented by signed distance fields, voxels of the two objects covering the same space are added together. The sums of these additions determine the penetration at each voxel, and from this information along with the grids, contacts can be generated.

Signed distance fields have been used for collision detection in several articles and papers[12][6], but implementations actually used by 3D physics engines are sparse.

3.1.3 Image-Space Algorithms

Algorithms using the rendering properties of GPUs for collision detection are called image-space algorithms[21]. By rendering both objects to a 2D-surface and examining their area of intersection, we can find that for an intersection with zero area, the objects are separated, while if there is an area they can still be intersecting. By rendering the objects from many different angles, a good estimation to whether the objects intersect or not can be made. However, as objects that intersect in all 2D projections made can still be non-intersecting, another pass will likely have to be made for a reliable intersection test.

A problem with image-space implementations is the resolution of the viewport. It puts a limit to the precision of the intersection tests, and because of this objects that actually collide can be reported as non-colliding.

3.1.4 Convex Collision Detection and Convex Decomposition

For non-intersecting convex objects, a plane separating the objects can always be found. The convex property of the objects also means that if a point in one object is closer to the other object than its neightbouring points in the first object (such as a vertex and its neighbouring vertices), then that local minimum is also a global minimum, see Figure 3.1. This makes recursive algorithms searching for such a minimum the primary way of testing general convex objects for intersection[8][29]. A negative aspect of these algorithms is that they only generate one contact point. This makes it difficult to obtain stable simulations, as objects most likely will alter between several possible contact points over time instead of maintaining a resting contact on several of the contact points.

These algorithms also only apply to convex objects. To be able to use them for concave objects, a concave object can be divided into several convex objects. This is called convex decomposition. Any concave object can be divided into convex pieces[3], but dividing an arbitrary object into the least possible number of convex pieces has proven to be NP-hard[26]. Fortunately, we do not need the least possible number of convex pieces. Also, convex decomposition only has to be done once for each object in a pre-processing step.

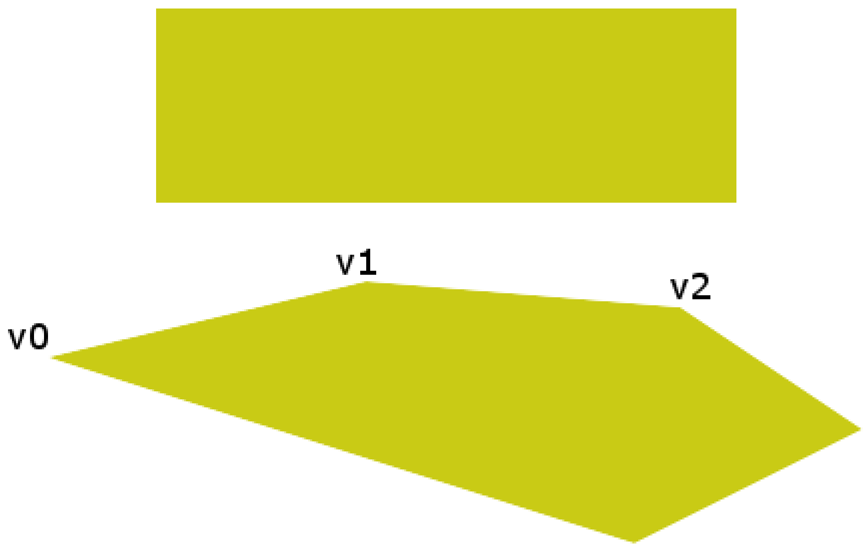

Figure 3.1: *Two convex objects in intersection test. The vertex v1 is closer to the other object than its neighbouring vertices v0 and v2. The convexity of the objects means that the smallest distance between the two objects must be in this area.*

3.1.5 Splines

A spline is a function defined by polynomial parts. Its main property is that smooth curves can be generated with relatively small data. Splines are very popular in CAD and design of 3D models because of this.

However, for collision detection, testing objects defined by such curves is time consuming. Most implementations use a middle step where the objects are converted into meshes, moving away from the splines. Collision detection using splines and not converting to meshes has been implemented[10], but not by any widely used collision detection library.

3.2 Bounding Volumes

A bounding volume is an object enclosing another object. It is useful e.g. when testing two highly detailed objects for collision. By initially bounding complex object with simple volumes such as spheres, the simpler objects can be used in a first step intersection test. As the spheres fully contain the detailed objects, two non-intersecting spheres means the contained objects are also non-intersecting, and an expensive intersection test can be skipped. On the other hand, two intersecting bounding volumes do not mean their contained objects are colliding. In this case, a more detailed collision detection has to be done.

There are many different bounding volumes, each having its own positive and negative sides. For a bounding volume to be effective, the two most important factors are that intersection tests between bounding volumes should be cheap, and that the area around the contained object is as small as possible. Being able to quickly compute the bounding volume as well as small memory footprint are also valuable factors. No bounding volume is good in all these areas. In general, a bounding volume with a cheap intersection test has more excess volume, where tightly fitted bounding volumes are more expensive to intersect.

As mentioned above, bounding volumes are useful for cheap rejection tests when testing for intersection. For triangle-triangle collision detection, a hierarchy of bounding volumes is used (covered in Section 3.3). Although not covered in this thesis, bounding volumes are also commonly used in the broad phase collision detection of the collision detection pipeline.

3.2.1 Bounding Spheres

The sphere is the simplest of the bounding volumes. It has very low memory footprint as it only requires a centre point and radius to be described. Intersecting two spheres is very cheap, and is done by determining whether the distance between them is less than the sum of the radii.

Algorithm 3.1 $boolSphereSphereIntersect(Spheres1, Spheres2)$

```
print  return $s1.radius + s2.radius < ComputeDistance(s1.centre, s2.centre)$
```

However, the tightness of a bounding sphere is generally bad, and false intersections will be more numerous than if using bounding volumes with a tighter fit.

3.2.2 Bounding Boxes

Bounding boxes are probably the most used bounding volumes today, as they are easy to use, position and intersect. There are two different variations of bounding boxes, Axis Aligned Bounding Boxes (AABBs) and Oriented Bounding Boxes (OBBs).

AABBs

AABBs are called this because the axes of a box is aligned to the axes of the world coordinates. When it comes to storage, this means that each bounding box can be described by a position and the length of the box along each axis.

Colliding AABBs is cheap, as they both are aligned to the world axes. It is merely a matter of checking for every axis, if the two objects intersect. If that is the case for all axes, the two boxes fully intersect.

As objects move and rotate in a simulation, an AABB will have to be recalculated every time-step in order to fully include the bounded object and still ensure a tight bound.

OBBs

OBBs are bounding boxes with arbitrary orientations. This enables boxes to bound objects with a potentially tighter fit than AABBs, and in worst case have the same bound as AABBs. Because of the orientation of the box, it must be represented by either a rotation matrix or a quaternion, as well as the position and span of the box as with AABBs.

An intersection test of OBBs can cost quite a bit more than AABBs, but with the usage of the Separating Axis Theorem (described in Section 3.4) introduced by Gottschalk et al[9] it can be done efficiently.

As OBBs can be rotated with the object, the box does not have to be recomputed every step as with AABBs, but the rotation of the OBB will have to be updated with respect to the rotation of the object. This is generally cheaper than recomputing the box, especially when bounding very complex objects.

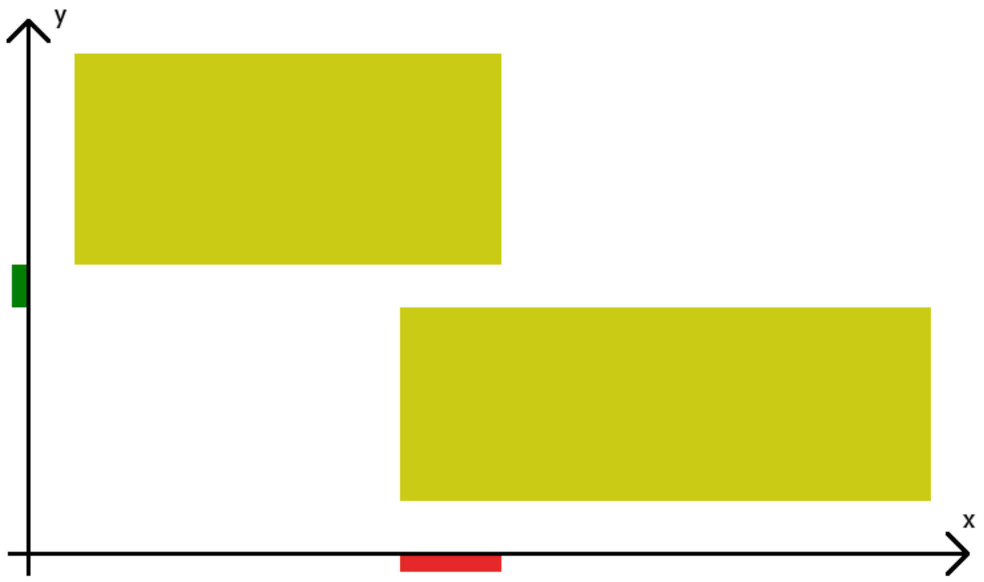

Figure 3.2: *Example of two AABBs in intersection test in two dimensions. As can be seen, they are separated on one axis, which means the two boxes are not intersecting.*

Non-Updated AABBs

Something which I have not found in any litterature are what I would like to call non-updated AABBs. However, they appear in many different collision detection implementations[31][17]. Non-updated AABBs are essentially a mix of AABBs and OBBs. At object creation, the bounding box is created as an AABB, and does not need its rotation stored. Instead, the rotation of the box is the rotation of the objects it bounds. This means, just as with OBBs, that the box does not need to be recomputed for every step.

While the non-updated AABBs might not seem very smart because the memory usage is the same as for an OBB but with a less tight bound, it is good if many boxes share the same rotation. This can be useful while constructing bounding volume hierarchies, which is further covered in Section 3.3.

3.2.3 Other Bounding Volumes

Other bounding volumes worth mentioning are k-DOPs and convex hulls.

A k-DOP (Discrete Oriented Polytope) is a bounding volume that encloses the object within the half-spaces of k planes (a half-space being one of the two parts of space divided by a plane)[16]. I.e. an AABB is a 6-DOP, with the 6 planes being axis aligned. More planes can be used to tighten the bound, with 10-DOPs, 18-DOPs and 26-DOPs being popular choices (beveling all corners, all edges or all corners and edges).

In two dimensions, the convex hull of an object can be described by imagining a tight rubber band around the object. For the general dimensional case, the convex hull is the minimum convex volume bounding the object. This is the bounding volume with the tightest fit of all the described bounding volumes, the downsides being a possibly expensive intersection test and high memory usage.

3.3 Bounding Volume Hierarchies

While bounding volumes are likely to speed up intersection tests, especially when bounding detailed objects, the same number of tests are still performed as without using bounding volumes. By creating a tree structure for the bounding volumes, logarithmic time complexity can be achieved. These tree structures are called Bounding Volumes Hierarchies.

In such a hierarchy, the root node is a bounding volume containing all the original bounding volumes. The root nodes children divide the objects into two or more bounding volumes, and this goes on recursively until the bounding volumes bound only one object each.

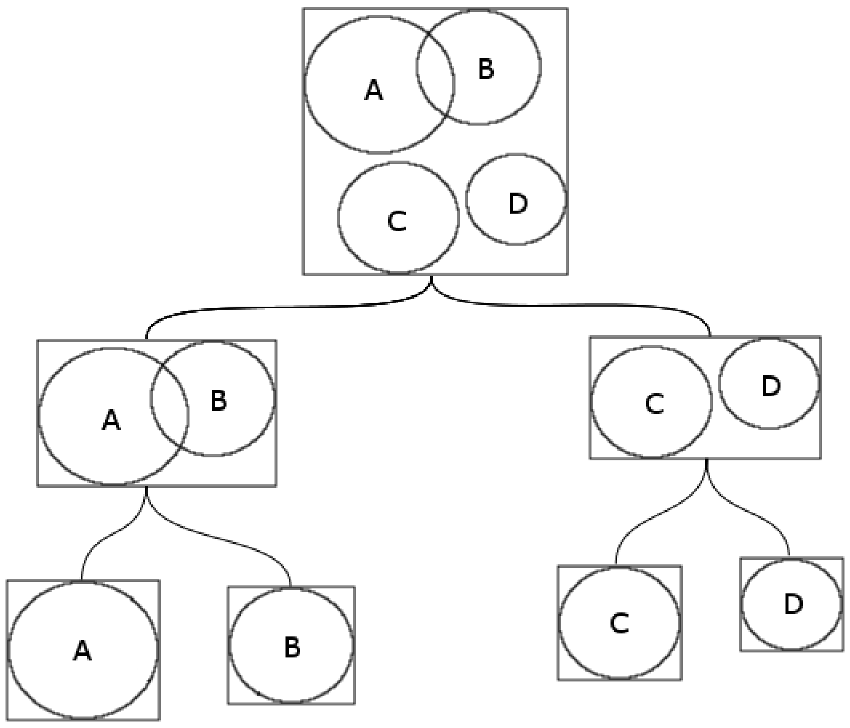

Figure 3.3: *An example of a hierarchy of AABBs bounding four circles. The root of the hierarchy is the top AABB bounding all circles, while its children are bounding smaller parts.*

By traversing such a hierarchy of bounding volumes, colliding them against another hierarchy, whole sets of bounding volumes can be pruned early. In worst case, if all objects of the hierarchy collide with the object it is tested against, performance is worse than with no hierarchy at all because of the intermediate step of the traversal. In reality this never happens, and bounding volume hierarchies are the primary way of speeding up large sets of objects colliding.

The issue with these hierarchies is that if the objects within a hierarchy move, the hierarchy has to be recalculated. This makes hierarchies very suitable for objects that are static in relation to each other, e.g. a non-deformable mesh of triangles.

3.3.1 Construction

Construction of a bounding volume hierarchy can be done in several different ways, but the most important factor is that the result is a hierarchy where traversal is quick for the general case. For this to be true, objects within the same bounding volume should be close to each other, reducing the size of the volume. Also, it is important that the objects, when divided into children, are divided evenly into sub-bounding volumes (in other words, that the tree is balanced). This allows for the largest number of objects possible to be pruned when a branch of the hierarchy is not traversed.

Building Strategies

When constructing a bounding volume hierarchy, there are three main ways to do this. These are top-down, bottom-up and by insertion.

When using top-down construction, the root node is first created, containing all objects. The children of the root are then created by splitting the objects based on their position. This goes on recursively for bounding volumes consisting of fewer and fewer objects, until no more splitting can be done.

The bottom-up technique takes the opposite route, by starting with the leaves of the hierarchy (each object and its bounding volume). The hierarchy is then built by finding bounding volumes that are close to each other and pairing them up, repeated recursively until the root is created.

With the insertion technique, the bounding volume hierarchy is created by inserting one object at a time. The tree is then partially recomputed at every insertion. The main advantage of using insertion is that objects can be added after the hierarchy is constructed for the first time, re-using most of the hierarchy.

From the study of previous collision detection engines, described further in Section 5.2, top-down construction seems to be the most widely used construction technique, probably because it is easier to implement and behaves well, although it might not be the best behaving technique.

3.3.2 Traversal

There are three main methods of traversing bounding volume hierarchies, breadth-first, depth-first and or a more intelligent traversal where traversal is prioritized based on some criterion. As the traversal never exits early when using bounding volume hierarchies for collision detection (or contacts will be lost), there is little to gain by using an intelligent traversal method. The most common method used is depth-first, as breadth-first requires a stack of nodes to traverse while depth-first relies on the program's call stack.

Descent Rules

When the root nodes of two bounding volume hierarchies collide, both hierarchies are traversed. There are several different ways of descending these hierarchies The most important ones are covered in the list below.

– Descend one before the other - Fully descend one hierarchy before descending the other. This can be a bad idea if a hierarchy of small volumes is descended first. The larger volume will then have to be traversed many times and the number of bounding volume intersection tests quickly increases.

– Descend the larger volume first - Here a comparison of the two volumes to be descended is made, and the larger one is descended first.

– Descend the hierarchies alternatingly - This rule is not as optimal as descending the larger volume first in terms of bounding volume intersection tests, but the cost of volume testing is spared.

– Descend both hierarchies simultaneously - Instead of descending one hierarchy at a time, both hierarchies can be descended simultaneously. In the case of binary trees, this means going directly to four new intersection tests, the two leaves of each volume against each other.

It is difficult to name any of these rules as better than the others, as it is highly dependant on the properties of the hierarchies colliding.

3.3.3 Bounding Volumes and Bounding Volume Hierarchies

Different preparations have to be made to hierarchies of different bounding volumes.

For AABBs, every bounding volume used in an intersection test has to be recomputed to keep its alignment to the world coordinate axes.

OBBs do not have to recalculate their bounds as they are rotated along with the object, but when colliding two arbitrarily aligned boxes, it is cheapest to temporarily rotate them so that one of the boxes is aligned to the world coordinate axes. This rotation has to be done for every pair of volumes being tested for intersection.

Non-updated AABBs have the advantage of not having to update their bounding volumes. As opposed to OBBs having to calculate a rotation for every intersection test, with non-updated AABBs this is necessary only once for the two hierarchies, and it can be re-used in all intersection tests since all bounding volumes in one hierarchy are aligned to the same axes.

3.4 Separating Axis Theorem

The Separating Axis Theorem follows from the more general Separating Hyperplane Theorem in the case of three dimensions.

The Separating Axis Theorem states that for two convex objects, there exists a plane separating the objects if and only if they are not intersecting. From this follows that on an axis perpendicular to this plane, the projections of the two objects will not overlap if and only if they are not intersecting. This axis is called the separating axis.

For an implementation using the Separating Axis Theorem, there is an unlimited number of possible separating axes. However, for convex polyhedra, there is a limited number of separating axes that covers all possible cases of intersection. These cases are edge-edge, edge-face and face-face collisions. For face-face and face-edge collisions, testing the normals of the faces as separating axes are enough. For the case of a possible edge-edge collision, the cross products of all edges of one object and all edges of the other object will cover all possible intersections.

For testing two boxes, there are fifteen different axes to be tested to be sure of an intersection, three for each objects normals and nine for the cross products of the edges (where edges and normals are the same in this case).

If a separating axis is found, it follows that a separating plane exists and the test can exit early without testing the rest of the possible separating axes. Because of this, it is a

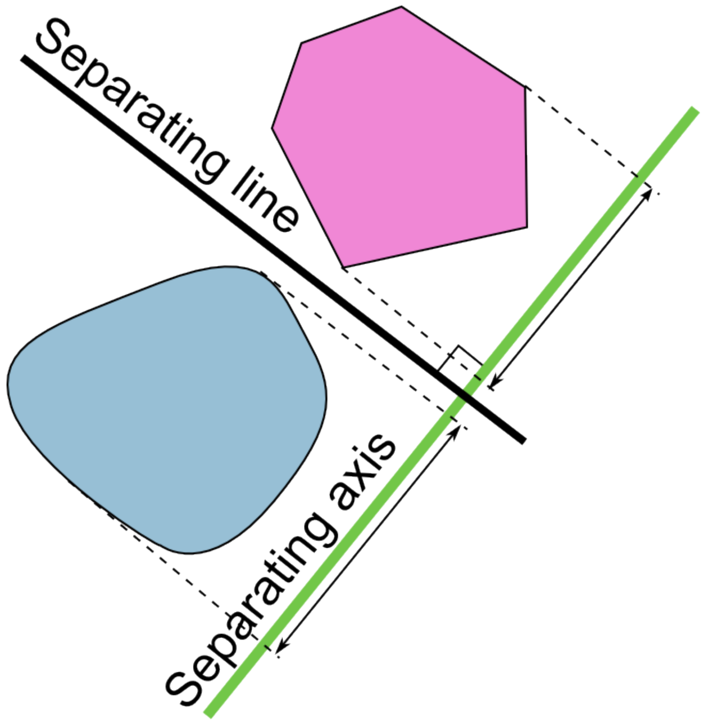

Figure 3.4: *The projections on the separating axis do not overlap if and only if the convex objects are not intersecting.*

good idea to test the axes that are most likely to be a separating axis first. These are often the normals of the faces, but one can also test the axis that was the separating axis in an earlier test, as temporal coherency makes it likely that this is a separating axis this time too.

3.5 Triangle-Triangle Intersection

The triangle-triangle intersection test is vital to triangle mesh collision detection and is covered here in more detail.

Triangle-triangle intersection can be tested by using the Separating Axis Theorem, but it can be unnecessarily expensive if no early exit is found, as eleven possible separating axes have to be tested.

The intersection test described by Möller[19] is much faster in the worst case but also has some early rejection tests that makes it very good. This intersection test will be described further below.

The early rejection test is performed by testing the three vertices of one triangle against the plane of the other triangle. If all vertices are on the same side of the plane, the triangles must be separated. If they are on both sides, the same test is done but vice versa.

If intersection has not yet been rejected by the above tests, the line defined by the intersection of the two planes is computed. Because of the earlier tests, the two triangles are guaranteed to intersect this line. The intervals of both triangles along this line are calculated. If and only if these intervals overlap, the triangles are intersecting.

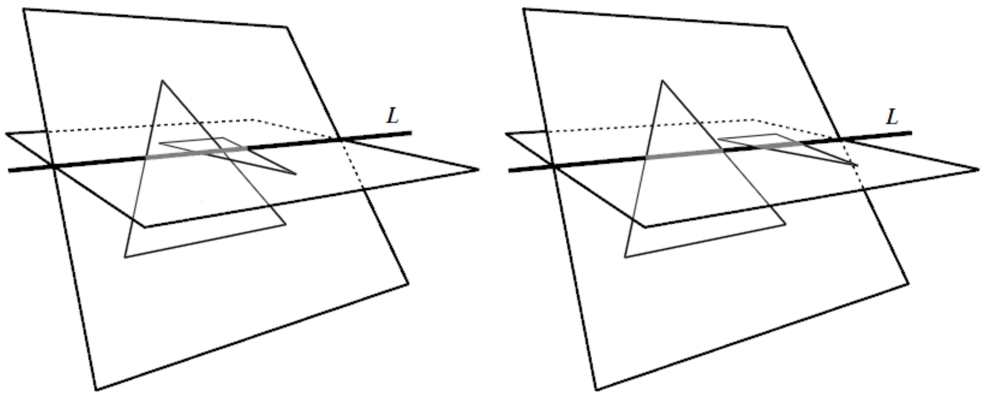

Figure 3.5: *The line L defined by the intersection of the two planes the triangles lie in. The two possible situations are shown, one where the intersections of the triangles on the line overlap each other and the other where they do not.*

The two triangles can also be co-planar, meaning that they lie in the same plane. This will be noticed in the early rejection test if all vertices of one triangle lie in the plane of the other. If this is the case, a simple two-dimensional intersection test can be executed.

Chapter 4

General Purpose Programming on GPUs and OpenCL

Imagine a series of frames being rendered, i.e. a movie being played or a game running. For this, the GPU has to compute millions of pixels every frame. This very much conforms to the SIMD (Single Instruction Multiple Data) architecture, and this is what the GPU excels at.

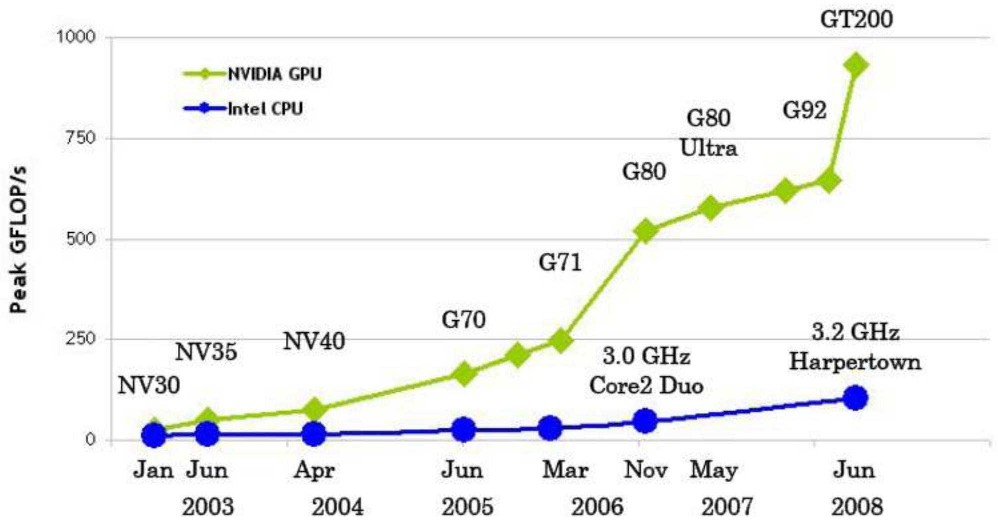

Figure 4.1: *A comparison of peak floating point operations per second between Nvidia GPUs and Intel CPUs over time.*

Since the computing power of a GPU is much higher than that of a CPU (see Figure 4.1), it has been the interest for many years now to take advantage of that power outside of the traditional graphics area. This is called General Purpose GPU programming (or GPGPU) and has evolved very strongly with the massively parallel architectures of the GPUs.

It started with more advanced programmable shaders, where one could bend the graphics pipeline to do other computations than graphics. Although it was still not focused on doing general programming, it was now possible. The latest addition to GPGPU programming is OpenCL[11], a standard focusing on general purpose parallel programming in heterogeneous environments. Other resembling languages for GPGPU programming include C for CUDA[22], developed by Nvidia for use with their GPUs, and DirectCompute[18] from Microsoft included in DirectX11.

This chapter is divided into two parts, the first part is a general introduction to OpenCL and the second part deals with optimization of code for use with GPUs using Nvidias CUDA architecture.

4.1 OpenCL

Since the dawn of computing, most progress on processors has come from increased clock rate. Due to heat generation, power consumption and hardware design issues, this progression is no longer possible to the extent it has been before. Instead, more compute units are added, making parallel programming an issue for most software developers today.

Writing parallel programs for the CPU and GPU can differ a lot, with specific technologies available only for CPUs, and vendor-specific languages for GPUs.

OpenCL is a new standard for general purpose parallel programming in heterogeneous environments developed by the Khronos Group.

The Khronos Group is an industry-driven consortium focused on creating and maintaining free APIs related to graphics and multimedia among other things. Specifications maintained include ones such as OpenGL, a cross-platform graphics API, and COLLADA, a file-format for specifying 3D scenes and models.

Initially, Apple started development of OpenCL, but since they wanted a standard accepted by the industry, they went to the Khronos Group that formed a group consisting of representatives from the major CPU-, GPU- and software companies. Together they worked on and constructed the OpenCL Specification [15].

The specification is written with heterogeneity in focus, allowing OpenCL code to compile on many different devices. A device is hardware with the possibility of running OpenCL programs, it could be CPUs, GPUs or something in between (such as Intels new Larrabee architecture [28]).

The goal with OpenCL is to provide a way to write code for any of these hardware platforms without the need for different languages or other specific tools. It should also be possible to easily interact with and work concurrently on CPUs and GPUs or multiple GPUs.

It is up to the companies behind the OpenCL devices to add support for their own device, as what is given by Khronos is just a specification of the API that must be followed. This provides an abstraction of the underlying hardware that has not been available to both GPUs and CPUs before.

4.1.1 Programming Models

OpenCL is designed for two different programming models, data parallel and task parallel.

The data parallel model is the primary programming model of OpenCL and means that the same set of instructions are applied to a sequence of data in memory. SIMD (Single Instruction Multiple Data) is one way of achieving data parallelism, which is the technique GPUs are based on.

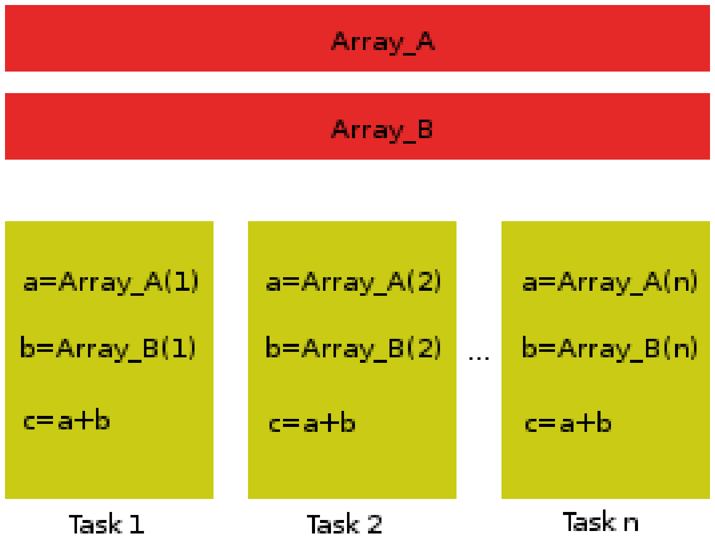

Figure 4.2: *Execution following a data parallel model.*

The task parallel programming model is a model where different tasks are executed on the same or separate data. An example is two CPU threads working concurrently on different tasks. OpenCL implementations do not have to support the task parallel programming model, but for devices such as CPUs it is the native parallel model.

4.1.2 Platform Model

The OpenCL platform model consists of a host and possibly multiple devices. An OpenCL application runs on the host, the host submits commands for the device(s) to execute.

A device can have many compute units, and each compute unit can have many processing elements.

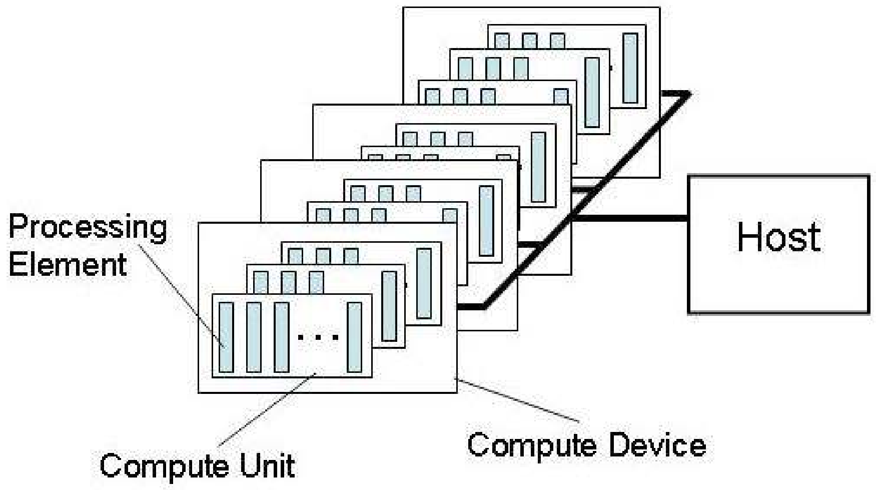

Figure 4.3: *The OpenCL platform model, showing one host with one or more devices, each device having one or more compute units that in turn each have one or more processing elements.*

For a general OpenCL program running on GPU, the CPU is the host that issues commands to the GPU. Todays GPUs have multiple compute units working more or less separately.

4.1.3 Execution Model

A kernel is a function executed on an OpenCL device that is reachable from the host. The host program specifies for the kernel which data to operate on and manages the kernel's execution.

When the host tells a device to execute a kernel, it specifies a work range. The kernel execution consists of different work-items, each with its own global ID. These global IDs are specified in either one, two or three dimensions, and they range from zero to the work range in that dimension minus one. Each work-item executes the same code, but possibly with different data and possibly with a different execution path in branches and loops.

The work-items are divided into work-groups. Each work-group consists of one or more work-items, and all work-items in the same work-group must execute on the same compute unit. This enables efficient sharing of compute unit specific memory. Work-groups are, just as work-items, specified by a unique ID in the same number of dimensions as the work-items.

The work-items of a work-group are given local IDs, unique for their work-group and together with the work-group ID unique for all work-items.

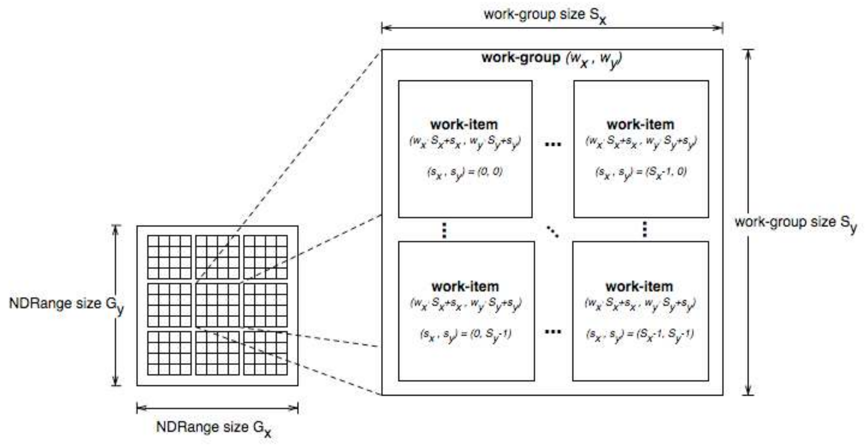
work-group size S_x
work-group (w_x, w_y)
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (0, 0)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (S_x-1, 0)$
work-group size S_y
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (0, S_y-1)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (S_x-1, S_y-1)$
NDRange size G_y
NDRange size G_x

Figure 4.4: *The OpenCL execution model, showing work-items with their global and local IDs specified in two dimensions.*

4.1.4 Memory Model

Work-items have access to four different memory regions. How these regions are implemented by the OpenCL implementation may differ, but it maps very well to how most GPUs natively store their data in hardware. The different memory regions are global, local, constant and private memory.

Global memory can be reached by all work-items of a kernel, as well as by the host application. This means read and writes from the host are usually done through this memory region.

Local memory is memory that is only reached by the work-items of a work-group. As all work-items in a work-group must be executed on the same compute unit, local memory usually resides close to the compute unit and is thus faster than global memory. The host can not access local memory.

Constant memory is much like global memory but must remain constant during kernel execution. This means caching can be implemented efficiently. Only the host can initialize data in constant memory.

Private memory is memory that is only accessible by a single work-item. Private memory is usually located close to each processing element, making it very fast for storing data soon to be used by the same work-item.

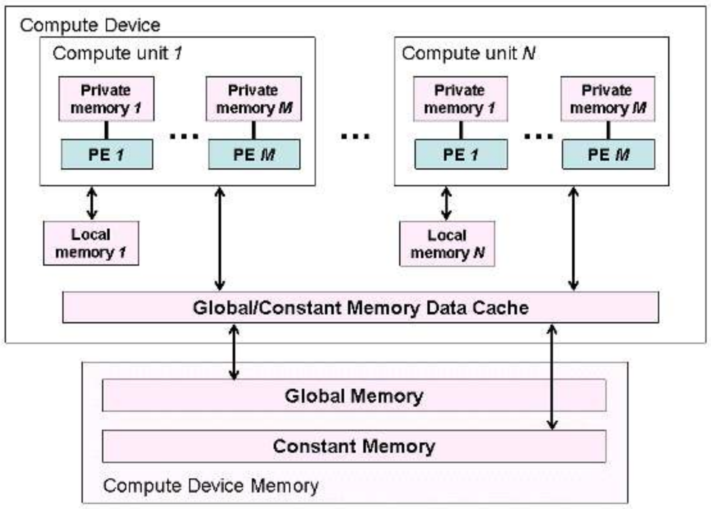

Figure 4.5: *The OpenCL memory model and its relation to the platform model. The processing elements (PEs) have access to private, local, global and constant memory, with differences being where it is located and who else has access to it.*

As can be seen in Figure 4.5 memory regions strongly relate to the division of the platform model.

4.1.5 OpenCL Programming Language

The OpenCL programming language is a subset of ISO C99 with some extensions, which makes it familiar to most programmers. As it is a subset, there are some restrictions such as recursion and function pointers not being supported. The extensions to OpenCL are for parallelism as well as some built-in data types such as vectors and built-in functions that are not part of ISO C99.

Listing 4.1 shows an example of the elements of two arrays being added together.

Listing 4.1: *OpenCL kernel code adding elements of two arrays together*

```
__kernel void add(__global const * vector1,
                  __global const * vector2, __global * result)
{
  int global_id = get_global_id(0);
  result[global_id] = vector1[global_id] + vector2[global_id];
}
```

For a device to execute something like this, the host needs to create a kernel from the source code and then specify the arguments before commanding the device to execute the kernel. The host can also specify size of the work-groups before executing.

4.2 Nvidia CUDA Architecture and Optimization

OpenCL is designed for heterogeneous environments, and written code complying to the specification should compile and run for any hardware that supports OpenCL. However, as different devices' hardware work differently, writing optimized code can differ a lot depending on the hardware to optimize for.

This section will be dedicated to Nvidias CUDA architecture and optimizing OpenCL code for it[23][25]. CUDA is the GPU architecture used in Nvidias latest graphics cards, from the GeForce 8 series and onwards. It is not to be confused with C for CUDA, which is Nvidias programming language utilizing CUDA for development of GPGPU applications.

A brief introduction to the CUDA architecture is first presented, allowing the reader to get an understanding of why the then presented optimization techniques work and how they work.

4.2.1 CUDA Architecture

The CUDA architecture maps very well to the OpenCL architecture. A CUDA device consists of a number of Streaming Multiprocessors (SMs), corresponding to OpenCL compute units.

Each multiprocessor consists of a number of smaller processors able to execute lightweight threads. One thread handles one OpenCL work-item, and one OpenCL work-group is executed as a thread block. The thread blocks are in turn divided in what is called warps. A warp is a fundamental part of the CUDA architecture, and consists of thirty-two threads. Warps will be covered further below.

When a thread block executes, all its threads run concurrently on one multiprocessor, but they are not implicitly synchronized, so if data needs to be shared and synchronized a barrier instruction needs to be executed. After a thread block is completed, a new block can take its place on the multiprocessor. As thread blocks are required to be able to run independently, the scheduler can schedule thread blocks to run on any number of multiprocessors and in any order. This makes writing code that scales with the number of multiprocessors very simple.

A multiprocessor consists of eight scalar processors, two special function units, local memory and, on newer cards, a double precision unit. A scalar processor handles instructions like add, multiply etc., while the special function units execute instructions such as square root, sine and cosine. On the multiprocessor there also resides a set of registers, a constant cache and a texture cache.

Because of the traditional use of GPUs as pure graphics accelerators where floating point operations are used mostly for pixel operations, blending and such, there has never been a need for double precision floating point operations. Because of this, scalar processors can only handle single precision floating point operations. While newer cards have support for double precision, it is still not nearly as fast as the scalar processors, making speed increases versus a double precision CPU implementation very difficult to achieve. This might change in the future, as the coming CUDA architecture, named Fermi, is said to have better support for double precision arithmetics[24].

As mentioned above, threads are bundled in groups of thirty-two called warps. A multiprocessor can only execute one instruction from one warp at a time, but it can handle multiple warps concurrently by very fast context switching between threads. This allows the multiprocessor to hide delays such as memory reads/writes. E.g. when issuing a read instruction, the multiprocessor can switch to another warp while waiting for the read instruction to complete.

There are different versions of the CUDA architecture, categorized by a what Nvidia calls Compute Capability. In general the differences are changes to number of available registers or multiprocessors, and these do not affect programming much except some possible fine-tuning. There is however one change in how data is accessed which makes the newer architectures a bit more forgiving when it comes to optimizing programs. In section 4.2.2 CUDA optimization will be discussed, and CUDA architectures using this newer way of accessing data will henceforth be referred to as architectures with high compute capability, while the older architecture versions will be referred to as low compute capability architectures.

4.2.2 CUDA Optimization

The speed of an implementation on a parallel platform depends entirely on how parallel the algorithms to be implemented are. This of course holds true for OpenCL as well, and easily parallelizable algorithms will see the greatest performance benefits.

Amdahl's law is used in computing to find the maximum possible speedup by parallelizing a sequential program.

$$S = \frac{1}{(1-P) + \frac{P}{N}} \tag{4.1}$$

Here S is the maximum speedup, N is the number of processors and P is the fraction of total serial time of the part of the code that can be parallelized. Speedup is defined as

$$S_N = \frac{T_1}{T_N} \tag{4.2}$$

where T_1 is the execution time of the sequential algorithm and T_N is the execution time of the parallel algorithm using N processors.

By approxating N as a very large number, Amdahl's law can essentially be written as

$$S = \frac{1}{1-P} \tag{4.3}$$

It is now easy to see that the greater P is, the greater is the speedup. It can also easily be seen that even if N is large and P is small, very little speedup can be achieved. Increasing N in this case also does not affect speedup notably. Thus, to effectively implement an algorithm using OpenCL, much thought should go into making sure as many parts of the algorithm as possible are parallelizable.

PCI Express

Today, data between the host CPU and device GPU is being sent over the PCI Express bus. Unfortunately for us, the PCIe bus bandwidth is low compared to GPU memory bandwidth. For the best possible performance of the application, it is important to minimize the amount of data being sent. This may mean running an intermediate part of the application on GPU instead of sending data to CPU, doing the computation and then sending it back, even though the CPU implementation is much faster. Another factor that is important to overcome the PCIe bandwidth bottle neck is that the more complex the computations on the data are, the less impact the data transfer part has on the total time. In general one wants applications with heavy computational complexity and many threads working at the same time.

Global Memory

With a global memory access comes a 400-600 clock cycle latency [23]. While most other operations are handled in less than a clock cycle, it is easy to understand that global memory accesses can play a big role in performance. There are several ways to minimize the role of these accesses and these will be covered below.

Because threads in the CUDA architecture are very lightweight, switching between threads is very time efficient. If threads of a warp have started accessing global memory, the multiprocessor can switch to other warps and switch back when the memory access has completed. By doing this, the latency of a global memory access can efficiently be hidden. This is one reason why a big problem set is important, as even though all threads cannot be serviced at once, you want enough threads to hide as much of this latency as possible.

Because of the way a CUDA GPU accesses its global memory, there is much to be gained by making sure the data is accessed in a way that is optimal for the GPU. This is called *coalescing* of global memory access, and in short means that threads of a warp that access data that are close in the global memory can do it in one or a few big memory transactions. If the data is spread out however, the GPU must do a separate transaction per data object.

First of all, when reading a word, the GPU wants the data to be aligned to either 4, 8 or 16 bytes in the memory. This means that data sent to the global memory should sometimes be padded in order to achieve this alignment. E.g. a struct of three floats takes 12 bytes of memory, but to meet the requirement of alignment it should be padded with another 4 bytes that are not used.

Secondly, the highest bandwidth is achieved when memory accesses by threads of a half-warp (upper or lower 16 threads of a warp) can be coalesced into one access. The requirements of coalescing access differ between GPUs of high and low compute capability. Cards with higher compute capabilities can coalesce accesses that low compute capability cards cannot.

For a data transaction by a card with a CUDA architecture having low compute capability to be coalesced, the threads must access words of size 4, 8 or 16 bytes. Also, the `k`-th thread in a half-warp must access the `k`-th word in a segment aligned to 16 times the size of the words being accessed. This can be illustrated in Figure 4.6 , notice that not all threads need to participate. If above conditions cannot be satisfied it will result in 16 separate transactions.

For GPUs with high compute capability, coalescing can happen more often. The algorithm for accessing memory by threads of a half warp looks like the following:

1. For the lowest numbered active thread, find the memory segment containing the address it wants to access. Segment size varies between 32 and 128 bytes depending on the size of the word to be accessed.

2. Find all other active threads whose address lies in the same segment

3. If the segment is 128 bytes and only the lower or upper part is used, reduce segment to 64 bytes

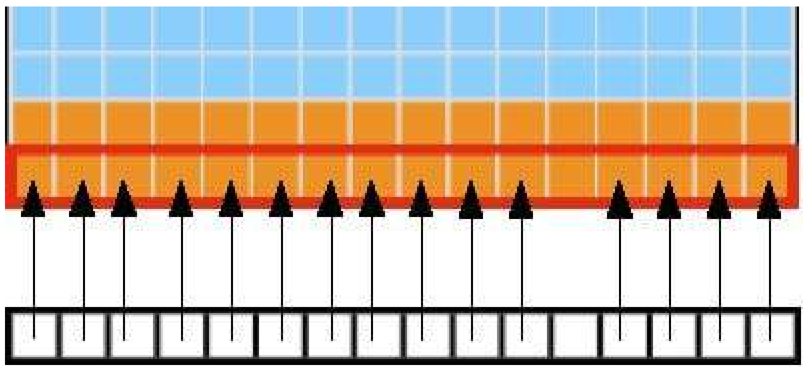

Figure 4.6: *Sequential data in global memory being accessed in one coalesced transaction.*

4. If the segment is 64 bytes and only the lower or upper part is used, reduce segment to 32 bytes
5. Carry out the access and mark the threads as done
6. Repeat until all threads of a half-warp are done

This means that a permutation of the accesses in Figure 4.6 would result in one coalesced access for GPUs of high compute capability.

Access to memory by a half-warp that is sequential but not perfectly aligned (an access with offset) will result in one or more coalesced accesses for a GPU with high compute capability, while with a low compute capability GPU it will result in a separate access per thread.

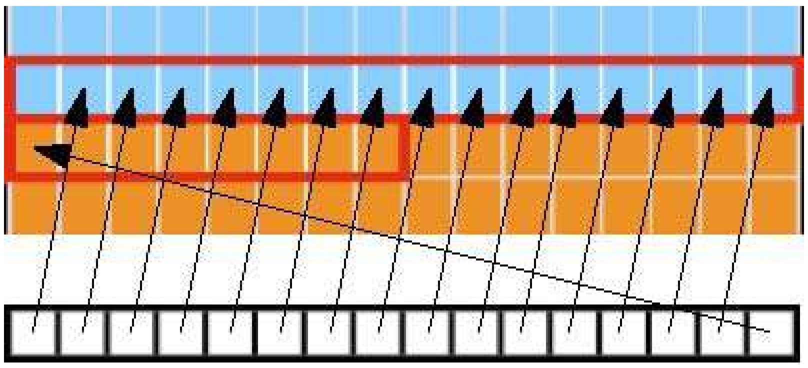

Figure 4.7: *Sequential data in global memory being accessed with an offset, resulting in two coalesced transactions on devices with high compute capability.*

Strided accesses (accesses to data that are separated by a number of steps) can also be coalesced when the compute capability is high, but the longer the stride is between data, the less coalescing is made and the lower the bandwidth is. An example of a strided access can be seen in Figure 4.8

All in all it is recommended to design one's code for low compute capability as that also benefits high compute capability, but if that is not possible then the extra versatility of high compute capability can help coalescing of memory access somewhat.

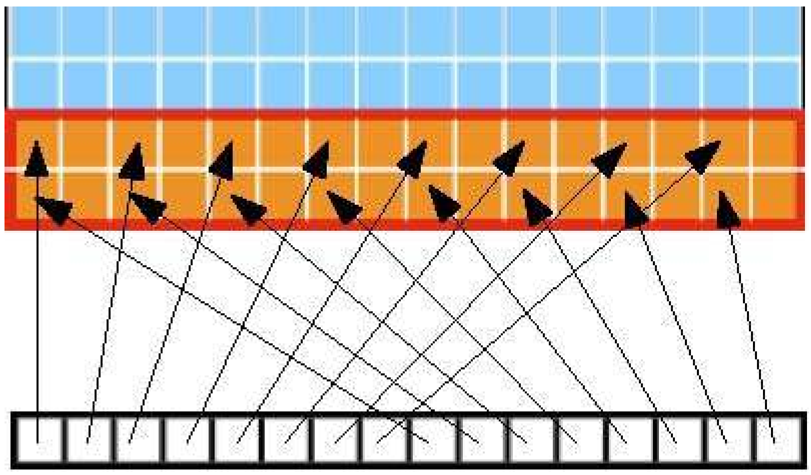

Figure 4.8: *Data in global memory being accessed with a stride of 2. For devices with high compute capability, the transaction is coalesced, but notice that half the data is not used. The longer the stride is, the smaller the effective bandwidth becomes.*

Local memory latency is much lower than global memory, so under some circumstances, pre-loading the data into local memory can be more efficient than loading it directly from global memory. As local memory is shared between a work-group, this holds true when multiple threads of a work-group want to access the same memory objects repeatedly.

Another time that pre-loading data to local memory might be beneficial is if loads from global memory cannot be coalesced. By reading the data to local memory in a coalesced way, and then sorting it in local memory where latency is much lower, faster reads can be achieved. However, one has to remember that the size of local memory is much smaller than the size of global memory, and only threads of the same work-group can access the same local memory, which limits the situations where this can be useful.

Local Memory

As local memory has the limitation of only being usable by threads of the same work-group, it can be situated close to each multiprocessor, and because of this it is very fast. It can be as fast as registers if no bank conflicts arise. Being aware of what banks and bank conflicts are and how to avoid the conflicts is the key to local memory optimization.

To achieve the high bandwidth that local memory has, the memory is divided into *banks.* Each bank can be accessed simultaneously, so an access using n banks yields a bandwidth n times as high as an access using only one bank. If two addresses of a memory access lie in the same bank, the accesses have to be serialized, resulting in a bank conflict.

Todays CUDA GPUs use 16 banks, and sequential 32-bit words are assigned to sequential banks. An example of this can be seen in example 4.1

EXAMPLE 4.1: *Sequentials 32-bit words and their assigned banks*
word 0 $\rightarrow$ bank 0
word 1 $\rightarrow$ bank 1
$\vdots$
word 15 $\rightarrow$ bank 15

word 16 → bank 0

A warp accessing local memory divides the access into two, one for each half-warp. As a half-warp consists of 16 threads, the same as the number of banks, all threads of a half-warp can access local memory simultaneously as long as they use separate banks. Another consequence is that there can be no bank conflicts between two threads that are not in the same half-warp.

Coalescing is straightforward if data to be accessed is aligned and each word is 32 bits.

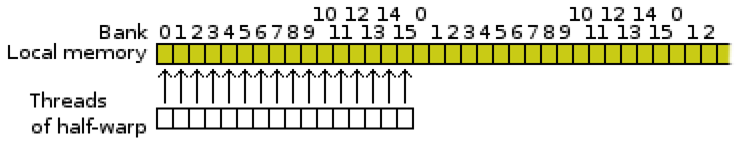

Figure 4.9: *Sequential data in local memory accessed, resulting in each thread using separate memory banks.*

However, it is common to access data using a stride, for example in any tree based algorithm. In Figure 4.10 can be seen that the larger the stride is, the more bank conflicts there are.

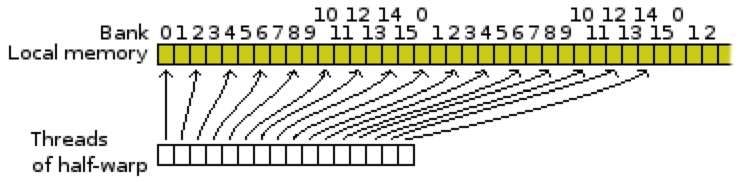

Figure 4.10: *Data in local memory accessed with a stride of 2, even numbered banks are now servicing two threads while odd banks are idle, resulting in a two-way bank conflict.*

Fortunately, the bank conflicts in Figure 4.10 can be resolved by padding the local memory by one for every 16 words. Every thread of a warp will now use a separate bank. This solution works well for any algorithm using striding. An example of strides of 2 can be seen in Figure 4.11. For an access of local memory of stride 3 or any other odd number, every threads access will be handled by a separate bank without any further intervention. A stride of 4 creates a bank conflict where only 4 banks out of the 16 are used. Padding every 16 words with a seventeenth word will solve the conflicts as all threads are serviced by separate banks

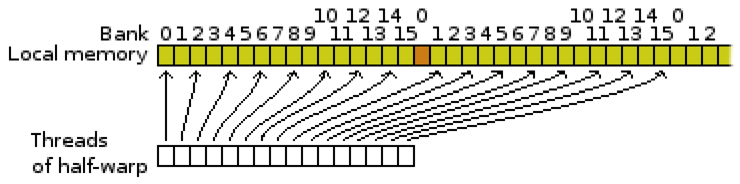

Figure 4.11: *Data in local memory accessed with a stride of 2, data is now padded so that for every 16th word a padding is inserted, resulting in slightly more memory usage but resolving bank conflicts.*

The exception to the rule of bank conflicts is when all threads of a half-warp access the same 32-bit word (and thus the same bank). A broadcast is then issued, servicing all threads simultaneously.

Constant memory

Constant memory is accessible to all threads of an OpenCL application, so it has the same bandwidth as global memory. However, it is cached, which means that this cost is only applied in case of a cache miss, otherwise a read from the constant cache is executed. Constant cache lies on each multiprocessor and is much faster than global memory.

The downside is that constant memory is very small, a total of 64KB on todays CUDA architectures, so one must be aware of how much data can be placed here.

Texture memory

Texture memory resides in global memory, but as with constant memory, it is cached on the multiprocessor so a global read is only issued on a cache miss. The texture cache is optimized for 2D locality, and it will boast very good performance compared to global memory if there is such locality in the data.

Texture memory is of course primarily used for textures and images, and has a few special abilities for use in those areas such as image filtering and normalized coordinates, but it can also be valuable for use with more general data.

Data exhibiting the 2D locality that texture memory is optimized for will likely receive a big increase in bandwidth. Texture memory can also be used to hide the issues of uncoalesced access to global memory, but the improvement here depends heavily on whether any locality can be exposed, and one would likely have try to both approaches to find which one is the best.

The texture cache is not kept coherent between consecutive reads and writes during the same kernel call. This means that reads occurring after a write might be dirty, and the result of a read of the same address as the previous write is undefined. Sometimes an extra buffer texture can be utilized to write to. Another solution is to divide the kernel into smaller kernels.

Occupancy

Calibrating arguments such as work-group size to balance the work-load of each multiprocessor and making sure the multiprocessors always have something to do is very important.

Occupancy is the ratio of active warps per multiprocessor to the maximum number of active warps. As mentioned before, there are latencies in memory accesses. There is also a latency in register usage if one wants to read from a register directly after writing to it. Both these latencies can be hidden by switching to another warp that is waiting to be executed, which makes a high occupancy desired.

Occupancy depends on the number of threads, registers and shared memory used per work-group. The scheduler will try to place as many work-groups per multiprocessor as possible. Unless occupancy is 100% (and thus the maximum number of threads per multiprocessor is reached), it is limited by either a work-group size that makes adding another work-group to the multiprocessor impossible, or registers/shared memory that will not be enough for another work-group.

Occupancy can easily be calculated from knowledge of the GPU's maximum threads per multiprocessor, available registers and shared memory as well as specifics of the program such as registers used, shared memory used and threads per work-group. Nvidia provides an occupancy calculator in the form of an excel spreadsheet, that not only calculates occupancy for the current setup but also for other values of the variable arguments, making it easier to find an optimal balance to use the most of the hardware's resources.

Work-Group Size

Choosing an appropriate work-group size is mostly a matter of testing in order to find what works best for the particular execution, but a few rules of thumb can be given. Work-group size should always be a multiple of warp size (where warp size is 32 in the current CUDA architectures), this is because threads of a warp work very closely together, and not filling up a warp will leave computational power untapped. A bigger work-group size is generally a good idea, unless occupancy takes a big hit from it, but if the problem size is not big enough, one could end up with very few work-groups per multiprocessor, or even multiprocessors not assigned to any work-group. Naturally, a smaller work-group size would then result in more work-groups per multiprocessor. The reason why work-groups per multiprocessor should be more than one is because a whole work-group might be waiting for some execution to complete. In that case, letting other work-groups execute will ensure making use of all possible executional power.

Instructions

There are many alternative instructions to use when speed is preferred over accuracy. Usage of these can give a very good performance boost, but since these optimizations are quite minor in terms of what is changed, it is recommended to postpone them until the more high-level optimizations above are done.

Not all of these optimizations will be named, rather a few are presented as examples of what can be optimized.

One such optimization lets the compiler group an instruction such as $a * b + c$ into one instead of a separate multiply followed by an addition instruction. However, in this optimized instruction, the result of the multiply is truncated and possibly valuable precision might be lost.

Costly operations such as integer division and modulo operations should be avoided if possible, and they can sometimes be replaced by bitwise operations.

There is a native math library for operations such as `native_sin`, `native_cos` and `native_pow`. These operations are much faster than their normal versions, but do not have the accuracy that the standard ones do.

Warp branching

As mentioned briefly above, threads within the same warp work together. All threads of a warp work in a SIMD (Single Instruction Multiple Data) fashion, letting one instruction at a time execute. The data used by the instruction can differ, but the instruction must be the same. This means that if a branch (such as an if statement or while loop) results in two threads of the same warp diverging into different execution paths, one thread has to idle while the other one executes its instruction and vice versa, until they converge onto the same path again.

Different execution paths of threads belonging to the same warp can severely hamper performance and as such, code should be designed to avoid branching as much as possible.

Chapter 5

Implementation

5.1 Methods

As a first step, before deciding on any implementation specifics, existing implementations of collision detection engines were analysed. This gave us knowledge about how others had made their designs and how they had overcome problems, which was helpful when deciding on upcoming design choices.

In order to get a good understanding of the problems ahead, numerous articles, papers and parts of books were read, covering fields of interest to the project such as bounding volume hierarchies, triangle-triangle intersection tests, GPU programming and contact reduction.

An implementation on CPU using C++ was built for use with AgX, the physics engine developed at Algoryx Simulations. Finally, a GPU implementations, also integrated with AgX, was implemented using OpenCL and optimized for Nvidia CUDA architectures. The implementations of near phase and middle phase collision detection on GPU were integrated into AgX (running on CPU) as stand alone modules with no direct interaction between them. Any interaction is done via the OpenCL host running in AgX. This enables the user to choose between using any combination of CPU and GPU implementations, e.g. running middle phase on CPU and then near phase on GPU.

Because of a lack of experience with GPU programming, tutorials and examples on OpenCL and general GPU programming were first studied before starting on the actual GPU implementations.

The study of existing collision detection engines and the CPU implementation was done in collaboration with an employee at Algoryx Simulation.

5.2 Study of Previous Work

As triangle mesh collision detection has been implemented in several other physics engines and stand-alone collision detection engines, the first step was to examine and compare available implementations.

A listing of physics engines and collision detection engines were collected, ranging from small one man hobby projects to university research projects to big commercial engines.

The purpose of the first step was to narrow the number of engines down to a number of engines that should undergo more rigorous testing. It was essentially a quick way of seeing which engines were mature enough to actually merit a deeper study.

Test Phase 1

As a first test, a simple scene of two triangulated spheres (spheres made up of small triangles, so not exactly spherical) with a slight intersection was defined. The scene was implemented in all engines, and the results were examined.

The interesting observations from such a simple test was seeing how many contacts were returned, what the contact data looked like (position, normal and distance) and a quick glance at the structure and maturity of code in the cases where this was possible.

Collision engines using continuous collision detection (*CCD*) were also included in the test. They rely on information such as linear and angular velocity of non-colliding objects to calculate the time of impact[27]. Because of these requirements, the above test scene with two triangulated spheres in intersection could not be used. Instead a similar scene was used, starting with two non-colliding spheric shapes and information about their movements. In the end, the decision was taken to not use CCD for collision detection. This was based on the fact that there are known problems with scalability of CCD implementations when more objects are included and that AgX would need large structural changes for CCD to be a viable option when implementing the triangle mesh collision detection.

Test Phase 2

As a second test phase, a number of more complex test scenes were defined in order to test bounding volume traversal performance, bounding volume buildup performance, contact reduction, contact quality and scaling.

These tests focused mostly on performance, and thus, timings of the different stages in the different test scenes were compared. As different engines used different techniques in terms of bounding volumes, triangle-triangle collision tests etc., only a rough estimate on performance of the different techniques could be made.

Examination of source code for the engines where possible also revealed information about how the bounding volume hierarchy was structured, if and how contact reduction was implemented as well as structuring of data and how contacts were created.

Results

Since the tests described above were mostly for getting an understanding of what was out there in terms of both products and behaviour of commonly used algorithms in an actual implementation, no results will be published. They do not contain enough conclusions to be appropriate for a scientific report, as, e.g., different engines put computations in different parts of code and some engines use double precision floats while others use single precision.

Despite these facts, some of the tests and observations gave good indications of possible solutions. Inspection of algorithms used by well-performing engines with an open source code was especially useful.

5.3 CPU Implementation

The CPU implementation of the triangle mesh collision detection was implemented as a part of the collision detection library in AgX, the physics engine developed at Algoryx Simulation.

This resulted in a number of benefits such as easy to use visualization of both objects and debug data (contact normals and positions) as well as a physics engine that could use the information from the collision detection so that one could test the collision detection in a running simulation environment. Also, things like data types and the ability to load object files was available and easy to use.

There were however some disadvantages to using AgX. The core design of the engine was static, and changing things here would require a lot of time to make sure everything still worked in other parts of the engine that were not involved in this project. This became apparent when considering using more than one thread in middle and near phase collision detection. AgX utilizes parallelization using CPU threads, but on a per object-object collision level. This would mean that multiple triangle mesh objects colliding would result in a significant speedup, while two large triangle mesh objects colliding would only use one thread.

The advantages however heavily outweighed the disadvantages, as it lead to much faster development and testing, resulting in time that could be used for more development and fine tuning.

5.3.1 Triangle Mesh Validation

For a simulation to work properly, center of mass, volume and inertia tensor is calculated for each object. For an accurate calculation of this, it is important that the meshes used are manifold. A manifold mesh is a mesh where each edge must be shared by exactly two triangles. This means that the mesh must be closed (that it must have an inside and outside and that there can be no holes). If a mesh is not closed, there are edges that are only used by one triangle. There can also be more than two triangles sharing an edge in a non-manifold mesh, having an impact on the calculation of mass properties.

By noticing that each face has exactly three edges and that each edge is shared by exactly two faces, the relationship $E = 3F/2$ should hold for all edges(E) and faces(F). If this is not true, the mesh cannot be a manifold.

Another important test that must be done is to make sure that all faces span a plane, i.e. that no face has all its vertices on a line. The collision detection requires the computation of the triangles planes to calculate a normal and penetration depth, which will of course fail if the triangles planes cannot be calculated.

5.3.2 Bounding Volume Hierarchy Construction

The bounding volumes used are non-updated Axis Aligned Bounding Boxes. When the Bounding Volume Hierarchy is constructed, every triangle of the object is enclosed by its own bounding volume.

The bounding volume hierarchy is built top-down by dividing this set of bounding volumes into smaller and smaller parts. This is not a novel approach, it and alternative approaches are described in Christer Ericsons book[5] Section 6.2.

1. Create a bounding volume for each triangle, resulting in a set of bounding volumes. This is the bounding volume set for the root node of the hierarchy.

2. If the set of bounding volumes has more than one member, it does not correspond to a leaf node and we continue splitting.

   (a) Calculate a bounding volume enclosing the whole set. This will be the bounding volume used in the hierarchy for this node.

   (b) Calculate splitting axis chosen as the coordinate axis with greatest variance.

(c) Calculate splitting position chosen as the mean value of the bounding volume centers.

(d) Sort the set of bounding volumes based on splitting position into two sets.

(e) For the two subsets, go back to 2.

Unless the set of bounding volumes are just one, which means it is a leaf node, a splitting axis is calculated along which the split should be made. The splitting axis is chosen as the one of the three primary axes with the greatest variance of bounding box position.

For the next step, a splitting position on the axis is chosen. The splitting position is the position where boxes positioned on one side become one subtree and boxes on the other side become another. This splitting position is chosen as the mean value of the boxes positions along the previously selected axis.

Now the tree is enclosed by a bounding volume, and both subtrees created earlier are divided in the same way as described above. This goes on recursively until only one of the original boxes that enclose a single triangle are part of every subtree.

5.3.3 Middle Phase Collision Detection

From the initial study of existing collision detection engines, appreciation for the engine GImpact[17], nowadays developed and integrated with Bullet Physics engine[4], was found. The speed, design choices and relatively good looking code were the three primary advantages.

This appreciation, along with a license that enabled its usage in commercial products and no intentions to re-invent the wheel, led to the decision to make use of parts of GImpact for collision detection. More specifically for middle phase collision detection.

Integration of GImpact

Integration of GImpact was mostly about understanding the GImpact code in order to integrate it as effectively as possible, both considering time taken to do the integration and the quality of the final result.

As the integration of GImpact within Bullet Physics was coupled quite hard, effort had to be put into only porting exactly what was needed and make it work without Bullet Physics.

Because we did not use GImpact as a library but integrated the code for middle phase, it more or less resulted in writing our own implementation, heavily inspired by GImpact.

Hierarchy and Hierarchy Traversal Implementations

For the hierarchy traversal, three different ways to implement and traverse the hierarchy were tested. These three implementations have previously been tested and discussed in a paper by Terdiman[31]. The first two were available in GImpact, but the third had to be implemented before being tested.

The first one can be called the standard implementation. Pseudocode of this algorithm can be seen in algorithm 5.1. The implementation proceeds down the hierarchy by traversing both objects. In the first iteration, two bounding volumes (the bounding volumes enclosing each object) are tested against each other. If they do not collide, no triangles inside the two bounding volumes can collide with a triangle in the other volume. If they do collide however, one moves down one level in the objects binary hierarchy trees. We now have two bounding volumes for each object, and they should be tested against the bounding volumes of the other object. This results in four new collision tests.

For each collision of bounding volumes, there are two possible paths to execute. If the collision returns a negative result, no triangles inside the two bounding volumes can collide, and as such, no further testing of the bounding volumes children is necessary. If the collision however is positive, the nodes status has to be examined.

There are three different execution paths when examining the two nodes statuses depending on whether they are leaf nodes or not.

1. If the two nodes are both leaf nodes, each node represent one triangle and the triangles are added to the list of triangles possibly colliding by the middle phase algorithm.
2. If one of the two nodes is a leaf node but the other is not, traversing of the node that is not a leaf must continue. Two new collision tests are issued, with the leaf node colliding with the two children of the non-leaf node.
3. If none of the two nodes are leaf nodes, both children of the two nodes are tested against each other, resulting in four new collision tests.

The algorithm continues recursively until the traversing of the tree returns.

Algorithm 5.1 $FindCollisionPairs(bvh1, bvh2)$

```
if not $BoxBoxCollide(bvh1, bvh2)$ then
  print return
end if
if $IsLeafNode(bvh1)$ then
  if $IsLeafNode(bvh2)$ then
    $AddCollisionPair(bvh1, bvh2)$
  else
    $FindCollisionPairs(bvh1, GetRightNode(bvh2))$
    $FindCollisionPairs(bvh1, GetLeftNode(bvh2))$
  end if
else
  if $IsLeafNode(bvh2)$ then
    $FindCollisionPairs(GetRightNode(bvh1), bvh2)$
    $FindCollisionPairs(GetLeftNode(bvh1), bvh2)$
  else
    $FindCollisionPairs(GetRightNode(bvh1), GetRightNode(bvh2))$
    $FindCollisionPairs(GetRightNode(bvh1), GetLeftNode(bvh2))$
    $FindCollisionPairs(GetLeftNode(bvh1), GetRightNode(bvh2))$
    $FindCollisionPairs(GetLeftNode(bvh1), GetLeftNode(bvh2))$
  end if
end if
```

The second implementation is algorithmically identical to the standard implementation, but varies some in how data is stored and accessed. It is called quantized bounding volume hierarchy, and is based on bounding volumes being quantized to short integers when created, which saves space in memory. This improvement in memory effectiveness can overcome the extra cost of unquantizing the data when used for collision detection.

The third implementation's algorithm differs from the standard implementation. The idea is that by not having bounding volumes for leaf nodes, memory footprint can be halved (as the size of every depth doubles in the binary tree). Instead, the triangles are used in collision checks. This means that if two leaf nodes are to be tested for collision, we skip the box-box test that would occur in the standard implementation, and move straight on to triangle-triangle test. In the case of a leaf node being tested against a non-leaf node, a box-triangle test has to be issued.

For the final implementation, the standard algorithm was chosen based on results found in chapter 6.

5.3.4 Near Phase Collision Detection

After middle phase collision detection, a list of pairs of possibly colliding triangles is returned. These triangles have a high probability of colliding as their bounding volumes are colliding, but an exact test has to be made. During this exact test, contact information (points, normals and depths) also has to be calculated and returned.

As described in chapter 2, the near phase collision detection can be divided into two categories, intersection test and intersection find. This implementation uses both. As the intersection test step provides many possible early-outs in case there is no intersection, it is executed first and then contact information is gathered only if the two triangles intersect.

Triangle-Triangle Intersection Test

The intersection test used was an implementation of the algorithm described in Section 3.5.

Generating Contacts

Generating contacts for triangle-triangle intersections was the most difficult part, and a lot of time was spent on getting this working. For a general collision of two convex shapes, contact point, normal and depth are well defined. The contact point is the point at the time of impact where collision occurs first and the normal is the normal of a plane separating the two objects so that they lie on each side of the plane. Contact depth at impact is zero. For discretized collision detection, contact point, normal and depth are not as well defined. When you cannot monitor the collisions continuously, objects are bound to be reported colliding after the initial time of impact. Contact depth is therefore used to report how much the objects are colliding, and is generally defined as the depth of the collision along the contact normal. Because of this discretization of the collision detection, contact point and normal cannot be computed as per definition. They must instead be estimated, and this can be an issue if the penetrations are large.

It is in the area of contact generation that most triangle-triangle collision detection engines struggle. As a consequence of bad contacts the behaviour of the physics simulation using them becomes error prone, resulting in misbehaving simulations.

The problem is that, as collision detection is computed on a per triangle pair basis, they do not have any information about the structure of the rest of the object. These problems are described more thorougly below.

To generate a contact, the two triangles normals are first computed. These triangles are considered to have one side that is in and one side that is out, otherwise the sign of the normal cannot be decided. The distance to move one triangle out of collision along the other triangle's normal is then measured for both triangles. We want to separate the triangles moving them as little as possible, so the triangle moving the least length along the other triangle's normal is chosen. This length is used as depth of the contact, while the other triangle's normal is used as contact normal and a point shared by the triangles are chosen as the contact point.

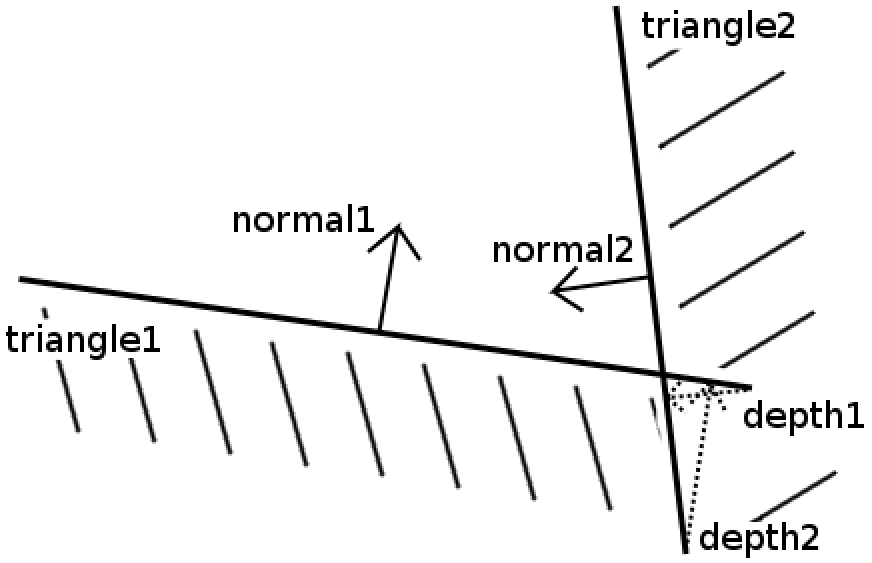

Figure 5.1: *2-dimensional simplification of two triangles intersecting. The depths for both triangles are calculated as the distance to move the triangle along the other triangles normal to separation. Here it can be seen how depth1 is smaller than depth2, which means depth1 is chosen as the depth together with normal2 as the normal.*

To remove unnecessary and possibly misleading contacts, the distance to separate the triangles in the direction of the line they share (see the intersection in Figure 5.1, in three dimensions this is a line) is also computed. If this distance is smaller than the depth of the already computed contact, the contact is discarded. This way triangle intersections that most likely do not tell much about the overall collision state of the objects are ignored, letting neighbouring intersections handle it instead.

Problems of Contact Generation

Soon after implementing a first version of near phase collision detection, we saw that finding contact points that would solve a collision between two generic triangle mesh objects based on information from triangle-triangle contacts would be very difficult.

In Figure 5.2 one can see a couple of contacts pointing to the right. For a box colliding on a plane like this, by seeing it on an object level, the fastest way out of the collision would be upwards. However, a few triangle-triangle collisions are easiest separated by moving the triangle belonging to the box to the right. This inconsistency of contacts is a difficult problem to solve, and many different solutions where considered.

Looking at normals and giving all normals the value of the most common one would give a satisfactory result in the example from Figure 5.2, but for objects with non-flat surfaces this would do more harm than good, as we want to keep the different normals such objects would generate.

Continuous collision detection was also considered (see Section 2.2.1). This means that the state of the colliding objects in the last frame is compared to the current state. The

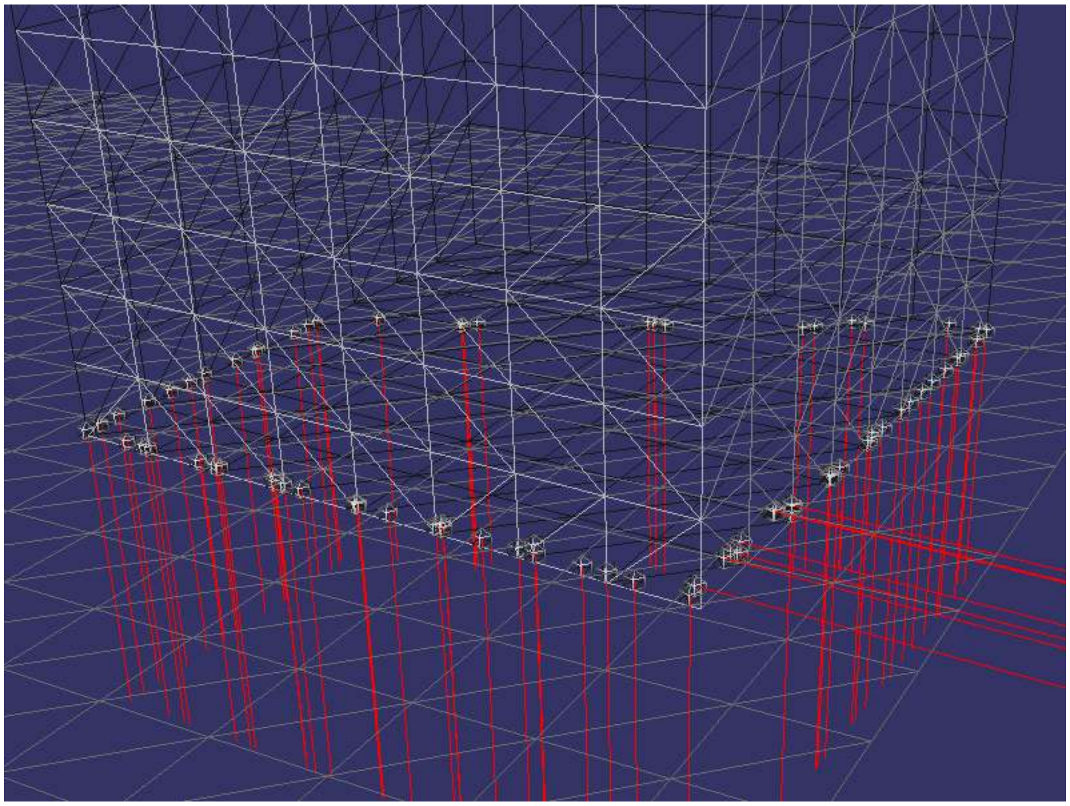

Figure 5.2: *A triangulated box colliding with a triangulated plane. Due to the placement of individual triangles, some normals point to the right while they ideally should point downwards.*

trajectory of the objects can then be computed and positions of the object at the time of collision can be found. By doing this on a triangle-triangle level, one could find the entry point of the collision and more easily compute a correct normal. However, in order to implement this, it would need big changes to many core parts of AgX. As AgX allows small penetrations, and generally does not solve them in less than three iterations, there would be times where the last frame already was in collision, and then this method would not be able to help.

Another possible solution would be using an algorithm for solving collisions between convex objects. However, as we do not want to be restricted to convex objects, these algorithms can not be applied directly. One solution to this is dividing a non-convex objects into several smaller convex objects. This is called convex decomposition (see Section 3.1.4). Once the object is divided into convex pieces, an algorithm for convex objects can be applied. However, these algorithms only return one contact per collision, and will when used in a physics engine result in objects having big difficulties to reach a resting state. To counter this, the idea that came up was to first divide objects into convex pieces and collide these convex objects. After this, the objects would be collided on a triangle-triangle level, but now with the normal of the convex collision detection as the direction in which to separate the triangles. This, however, would result in a very complex implementation that would take a lot more time to execute. Other drawbacks of this idea was that problems like those depicted in Figure 5.3 would occur, and on top of that, the collision algorithms for convex objects that are available are highly recursive and thus not suitable for parallelization.

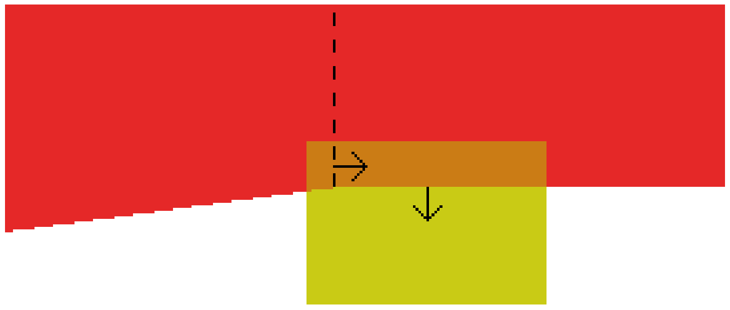

Figure 5.3: *By dividing the concave object into convex pieces, the fastest way to separate the box and the convex piece is along the normal of the newly created side. This results in a contact that while attempting to solve the collision with one of the convex pieces, it does so in a direction that should not be possible.*

In the end, the solution implemented first and described above, where triangles are tested individually against each other to generate contact data, was used. It proved to be stable and correct for the most cases, and in the cases of returning unwanted contacts they did not result in big flaws in the simulations.

5.3.5 Contact Reduction

Reducing the number of contacts to a smaller number that can still represent the collision can be very important. The physics engine solves a collision by doing complicated matrix computations. Some of these matrices grow exponentially with the number of collisions, and so does the time to solve them. As few contacts as possible is very important for solvers. They can even fail to solve a collision if they are fed too many contacts.

Although contact reduction is not a part of triangle mesh collision detection, such collisions can result in a huge number of contacts. Contact reduction is needed for triangle mesh collision detection to be of any use to the physics engine of a real time simulation.

The ideal solution is that the reduced contacts form the minimal convex cone of all contacts. A convex cone is a subset of a vector space that is closed under linear combinations with positive coefficients, closed meaning that any such linear combination will also lie in the same subset. In our problem, the minimal convex cone consists of the reduced contacts. The contacts removed in the reduction are all linear combinations of the set of reduced contacts.

To solve this problem in real time proved to be an incredible challenge, and in the end we could not find a solution that created the minimal convex cone. However, we came up with an algorithm that we call binning, and it is a time efficient replacement at the loss of some accuracy. The algorithm puts contacts in different bins in a 6-dimensional hypercube, allowing only one contact per bin. The idea is a modification of a binning method described in Game Programming Gems 4[20], but where it was used in three dimensions for normals only. The six dimensions of the cube corresponds to the positions and normals of contacts. The bin resolution decides how fine-grained the reduction should be, a higher resolution means more bins and reduces less contacts than a cube with lower bin resolution.

If two contacts land in the same bin, the one with the longest distance to the origin

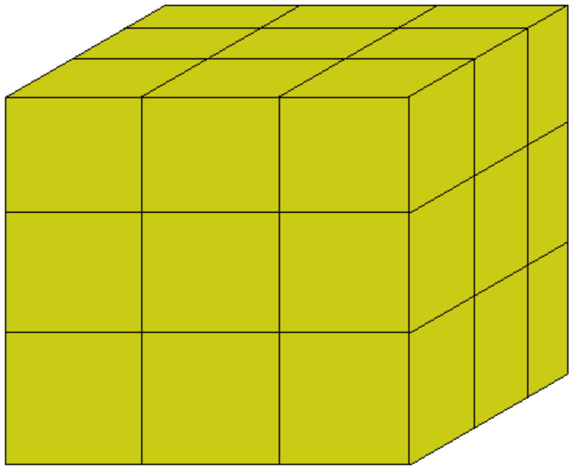

Figure 5.4: *A 3-dimensional cube with a bin resolution of 3 in each dimension. For contact reduction a 6-dimensional cube is used, but is more difficult to visualize. Contacts are assigned to bins, where only one contact can be present in each bin.*

is chosen and the other one is discarded. The contact that remains most likely dominates the discarded contact, meaning that the discarded contact has no impact on the simulation with the other one present. While normals are vectors and because of this can be said to always start from the origin, this is not true for positions. These had to be scaled so that all positions fit in a unit cube. This way both the contacts and normals are centered around the origin and the 6-dimensional cube can be used efficiently.

The outcome is a contact reduction algorithm where contacts with similar positions and normals are reduced significantly. Although it is not a perfect reduction of contacts in theory because contacts affecting the simulation can be discarded while contacts not affecting it are likely to be included, it behaves very well in practice.

5.4 GPU Implementation

An important aspect of this project was evaluating whether an implementation of triangle mesh collision detection could be implemented to run on a GPU with favourable speeds as a result.

This was done using OpenCL and optimized for Nvidia GPUs using the CUDA architecture. The decision to use Nvidia hardware was simple. OpenCL is still very new, and at the start of the project the only implementation of OpenCL supporting GPUs was Nvidia's.

Although the optimization is very much oriented at Nvidia GPUs, the general ideas and algorithms should work well and be applicable to any modern massively parallel architecture.

The following sections 5.4.1 and 5.4.2 are presented in the order of implementation rather than in the order of execution in the collision detection pipeline.

5.4.1 Near Phase

With no prior experience to GPU programming, we chose to implement near phase collision detection first. As near phase consists of a number of triangle pairs where each pair is going through the same computations with no interaction between different pairs, it is easily parallelized and was thus a good starting point.

Algorithm 5.2 Near Phase on CPU

```
for each pair of triangles do
  $TriangleTriangleTest(pair)$
end for
```

Algorithm 5.3 Near Phase on GPU

```
for all pairs of triangles in parallel do
  $TriangleTriangleTest(pair)$
end for
```

The near phase calculations are almost identical to the ones of the CPU implementation in Section 5.3.4, and because of that only the differences will be covered here.

The main problem with implementation of an algorithm like near phase for triangles is that it contains many branches and early exits. These branches will result in different threads running different execution paths. As can be recalled from chapter 4.2.2, threads of the same warp running different execution paths will result in serial execution. This means that, as different branching between intersection test and intersection find is very likely, triangle pairs that are quickly found not to be colliding will still take up computational power when other threads of the same warp keeps executing.

Because of this, the decision was made to completely skip the initial parts of the triangle-triangle intersection test. Instead, the computation of contact points to be added to the simulation is done for all triangle pairs, where this would only be done for triangles that have been flagged as intersecting in the CPU implementation. The contact generation routine is the same as in the CPU implementation, and in the case of a triangle-triangle pair not intersecting, it will not find any contacts. The reason you do not use only this contact generation routine for CPU is because it is much more costly than starting with the relatively cheap intersection test. On the GPU however, with the branching of threads, executing only this routine is favourable unless all 32 threads of a warp are non-intersecting, something that seems highly unlikely to be true for a majority of warps, given that the triangles bounding volumes are intersecting.

For fetching triangle data to the triangle-triangle intersection routine, two different designs were possible. One was to send all triangle data once to the GPU at initialization of the program. When running the intersection kernel, only the indices of the triangles to be tested have to be sent via the slow PCI Express bus. On the downside, triangle data reads will most likely be uncoalesced, as there will be no correlation in where the triangle data is located and which indices are sent. Also, GPU memory is a hard constraint, and the number of triangles that can be stored on the GPU will quite quickly be reached if many detailed objects are used in the simulation, especially if the GPU is also used for other computations that need memory. Because of these issues, the decision was made to send only the triangles used in every time step to the GPU. This enables easy coalescing of memory access and GPU memory usage is lowered, at the cost of having to send more data via the PCI Express bus.

After triangle-triangle intersection, the results are sent to the host CPU to be used by the physics engines solver.

5.4.2 Middle Phase

Implementing tree traversal in a parallel environment required quite a lot of time, as it is not a parallel algorithm by default and it is not simple to make parallel. As detailed in Section 5.3.3, the algorithm executes one bounding volume intersection test at a time and continues recursively until the two bounding volume hierarchy trees have been traversed.

In order to parallelize this, the idea was to divide the algorithm into one step for each depth of the hierarchies, as the bounding volume intersections of each depth can be computed independently of the other bounding volume intersections on the same depth.

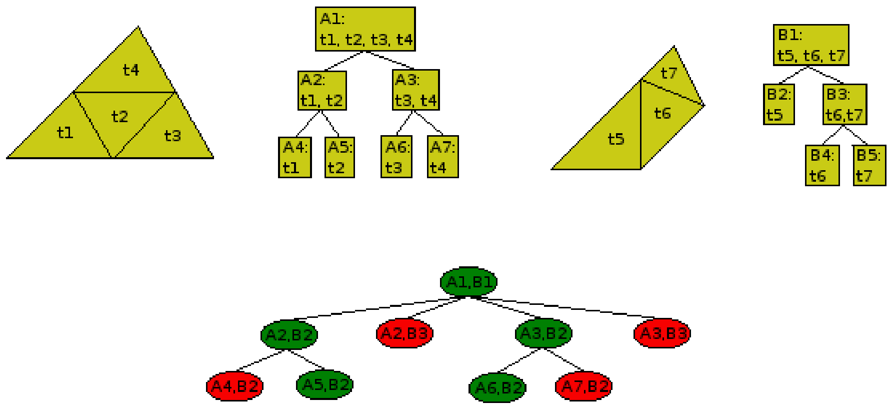

Figure 5.5: *Two bounding volume hierarchies colliding in a way so that triangle t3 in object A intersects with triangle t5 in object B. Note how node A5 reports as possibly colliding with node B2. This is a false intersection that will be handled by near phase, and is generated because the triangles bounding volumes intersect.*

The main issue behind why the structure of the GPU solution looks like it does, is that after testing two arrays of bounding volumes for intersection, we must fill these arrays with new bounding volumes to intersect in the next iteration. As a single thread has no explicit way of knowing the result of other threads, it means the implementation does not know where in the arrays to put the bounding volumes for the next iterations.

This is where prefix sum comes in. It is used in the implementation to sort data based on a binary condition. E.g. when preparing for the next iteration, some threads have processed bounding volumes that are not intersecting and should therefore not spawn any new intersection tests of its children, while other threads have processed intersecting bounding volumes and should. In this situation, the thread n will not know where in the array of bounding volumes to be executed in the next iteration to put its data without knowing how many of the previous threads also wants to spawn new tests. To get this knowledge, some kind of sorting has to be done. If thread n knows that all threads with an id lower than its own id also had intersecting bounding volumes and want to execute new tests in the next iteration, it will know where to put its data. Prefix sum can be used here instead of a more complex sort algorithm because the sorting is done based on a condition that is true or false rather than on data having a wide range of possible values.

Prefix Sum

The prefix sum algorithm takes an array of elements and calculates a new array containing elements that are the sum of all elements of the original array with an index lower than its own (exclusive prefix sum) or lower or equal to its own (inclusive prefix sum). The prefix algorithm is not restricted to the add operation, but can be used with e.g. logical `AND` too.

$$p = [a_0, a_0 + a_1, a_0 + a_1 + a_2, \ldots, a_0 + a_1 + a_2 + \ldots + a_n] \tag{5.1}$$

Example: Inclusive prefix sum of array

[1 2 3 4 5 6 7 8]

returns

[1 3 6 10 15 21 28 36]

A sequential implementation of prefix sum is simple, just iterate over the input array and keep adding the elements together, storing the intermediate results in a results array. A parallel implementation must work a bit differently because it cannot use the results of the previous iteration as the sequential implementation does[14].

Algorithm 5.4 Parallel prefix sum

```
for $d = 0$ to $log_2 n - 1$ do
  for all $k$ in $[0, n-1]$ in parallel do
    if $k \geq 2^d$ then
      $x[k] = x[k - 2^d] + x[k]$
    end if
  end for
end for
```

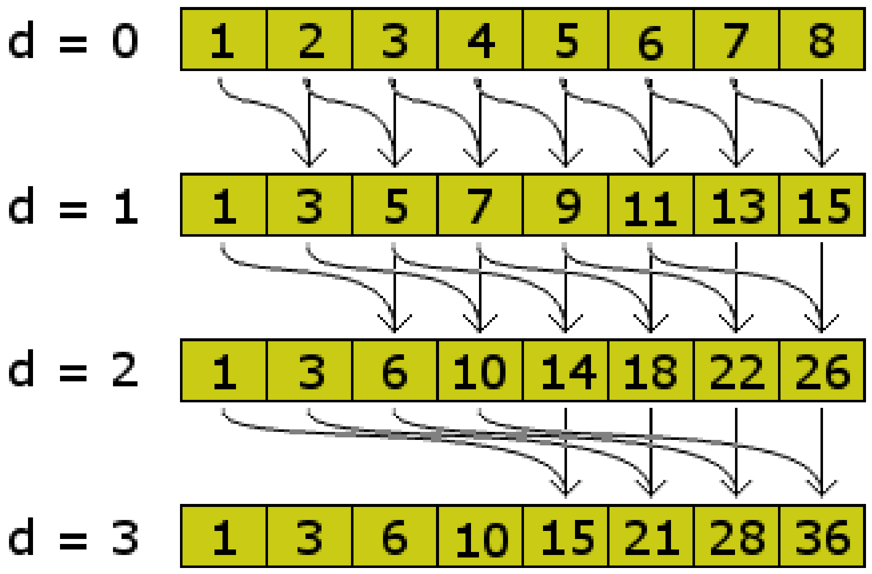

Figure 5.6: *An example of parallel prefix sum using algorithm 5.4.*

The algorithm described in algorithm 5.4 was implemented in OpenCL and an example is depicted in Figure 5.6. For better performance, data was copied from global memory to local memory before computations, and then copied back before returning the results. Usage of local memory is generally preferred over global memory when multiple accesses are made, as in this case with $log_2 n$ iterations.

Because local memory is local to the work-group, the maximum size of the work-group limited the number of elements that could be summed. This was solved by dividing the prefix sum calculation in several passes. By first dividing the elements in groups that were not limited by work-group size, you can then do another pass of prefix sum of the last elements of each group. The resulting elements of the second prefix sum pass can then be added to all the elements in the corresponding groups (depicted in Figure 5.7).

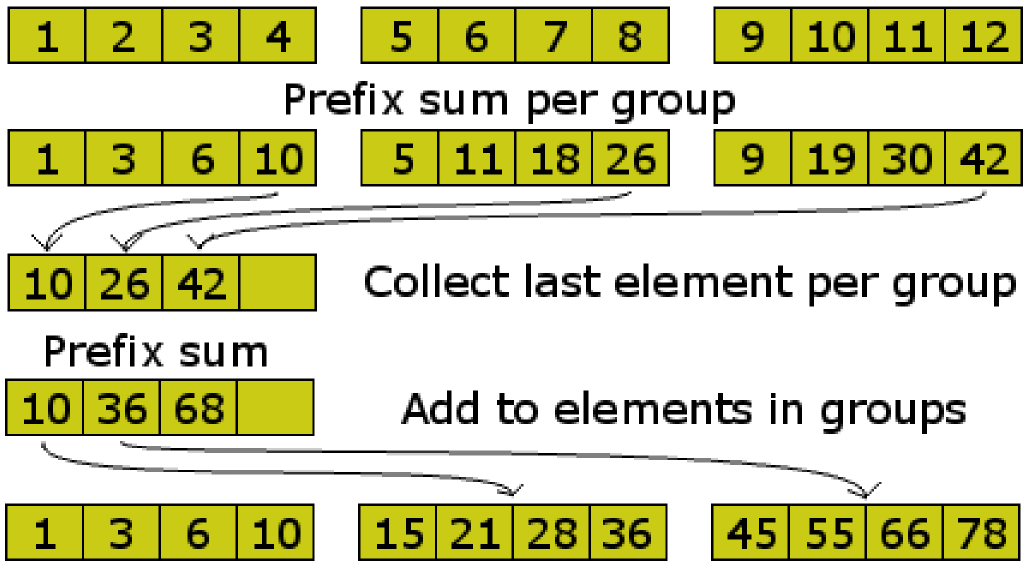

Figure 5.7: *An example of parallel prefix sum limited by a work-group size of 4, making two passes needed. For the second pass, the last element of each work-group is taken and the result is added to all elements of work-group $e + 1$.*

Prefix Sum Usage

By initializing an array with zeros and ones, depending on a binary condition, i.e. bounding volumes intersecting or not, and then doing inclusive prefix sum over said array, data can be sorted depending on whether they fulfill the condition or not.

An example of this can be seen in Figure 5.8. Notice how a comparison of element e and $e - 1$ can tell whether the initial value was a zero or one. Together with this information and the value of element e, elements originally having a one will know how many ones are preceding itself.

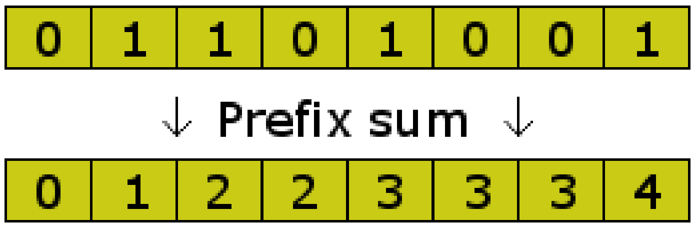

Figure 5.8: *An example of how the parallel prefix sum implemented can be used in our application. From the output of this prefix sum sorting the ones and zeros is possible in one iteration.*

Middle Phase Algorithm

The bounding volume hierarchy traversal was implemented as one iteration for every depth in the bounding volume hierarchy trees. Each iteration consists of many small steps, implemented as separate kernels.

The initial data is two arrays of bounding volumes, one array for each bounding volume hierarchy. Every element is to be tested for intersection against the corresponding element of the other array. There are also two arrays for results, one for each hierarchy. They will, during the execution of the algorithm, be filled with triangles whose bounding volumes intersect. Below is a list of the steps to complete the bounding volume hierarchy traversal.

1. Test bounding volumes for intersection.
2. Compute prefix sum of results from bounding volume intersection.
3. Move data of colliding bounding volumes to front of input arrays.
4. Test if both bounding volumes are leaf nodes or not.
5. Compute prefix sum of results from leaf-leaf test.
6. Move leaf-leaf data to results arrays and move non leaf-leaf data to front of input arrays.
7. Test if both bounding volumes are noleaf nodes.
8. Compute prefix sum of results from noleaf node test.
9. Prepare for next iteration by adding new bounding volumes to input arrays. Noleaf-noleaf data spawns four new tests while noleaf-leaf data spawns two new tests.
10. Return if input arrays are empty, otherwise go to top again.

For this implementation to realize full speeds, a big problem set with thousands of simultaneous bounding volumes possibly intersecting at each depth is important. The starting iteration only contains the two root nodes, and in best case they grow by a factor of four for every step. To remove this waste of computational power, the two hierarchies are first traversed down a number of steps without being tested. On a set depth, all nodes on said depths are tested against the nodes of the other hierarchy, e.g starting with 4096 intersection tests at depth 6. Intersection tests of this first iteration will have much lower intersection rates as their parent nodes have not been tested, but this is outweighed by skipping the first couple of iterations where the problem size has not yet grown.

Data Handling

As with the near phase implementation, there is the decision to either store the data used in every step of the simulation permanently on the GPU or send it via the PCI Express bus for every step. In this case, the data is the bounding volume hierarchies for every object. In contrast to the near phase algorithm, one can not know which parts of the data that are needed, as this is decided when traversing the hierarchies. Thus, coalescing of memory access will not be possible. Instead the decision lies entirely on whether you want to favour speed or smaller memory footprint. For this implementation, the choice was to favour a smaller memory footprint.

The result of middle phase, indices of triangles to collide in near phase, is sent to the host CPU. This is done to facilitate the choice of near phase algorithm, either run on CPU or GPU.

Chapter 6

Results

To compare the different implementations relative to each other a number of tests were defined. Two Stanford armadillos[30] were positioned to be colliding. Both objects were specified as static, meaning that they do not move in between the time steps. This makes it possible to achieve more accurate timings by taking the average of a number of iterations.

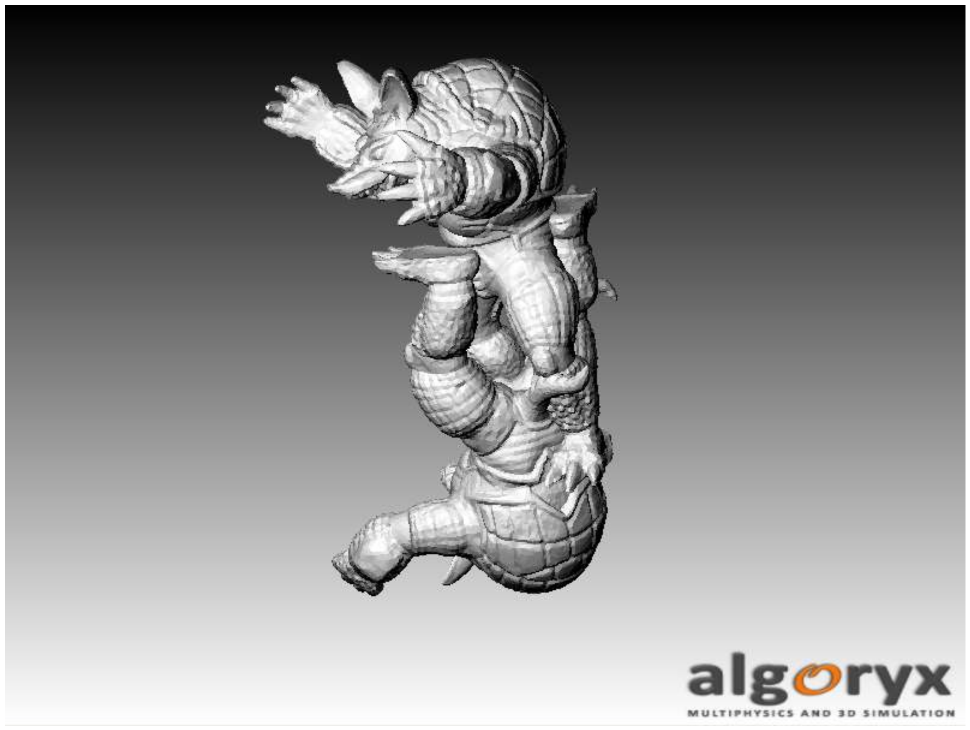

Figure 6.1: *An example of a test scene with two armadillo objects colliding. These two objects are armadillo_3 objects, each consisting of 27 674 triangles.*

To test for different scaling of problem sizes, armadillo objects built by different number of triangles were used. They originate from the same object but with varying quality.

The results below were acquired from test runs on a system consisting of an Intel Core2

Object name	Number of triangles
armadillo	345 944
armadillo_2	138 376
armadillo_3	27 674
armadillo_4	2766

Table 6.1: *Objects used in the tests and the number of triangles they are made of.*

6420 CPU running at a clock rate of 2.13GHz, 3.25 GB RAM and a Nvidia GeForce 8800 GTS. The operating system was Windows XP.

6.1 General Result

All in all, collision detection algorithms for colliding triangle meshes against each other were implemented both for CPU using C++ and GPU using OpenCL. Both were integrated with AgX. The visual result is triangle mesh objects interacting in a physical environment. Simulation in real time can be achieved for scenes with sufficiently few objects or objects with low number of triangles. Putting numbers on what can be done in real time or not is difficult, since it depends on hardware used, how the objects used are designed and the type of interaction between objects. Generally, a dozen objects with around a thousand triangles each should be able to interact in real time. For the GPU implementation, these numbers are a little bit lower, especially for multiple objects.

6.2 Middle Phase

The two implementations were compared to each other by timing collisions in four different scenes. Every scene consisted of two armadillo objects colliding, the difference between the scenes being the resolution of the armadillos.

Scene	CPU time (ms)	GPU time (ms)	Speedup
armadillo_4	2.12	10.16	0.21
armadillo_3	20.02	18.36	1.09
armadillo_2	113.06	48.86	2.31
armadillo	277.64	88.76	3.12

Table 6.2: *Comparison between middle phase CPU and GPU implementations.*

In table 6.2 one can see how the middle phase of collisions of smaller objects is faster on CPU for objects with less triangles, while for objects consisting of more triangles, the GPU implementation is superior. Unfortunately, the break point where the GPU implementation becomes faster than the CPU implementation is at simulations that are too big to be run in real time. However, it is promising for future hardware which will most likely be even more parallel than today's GPUs. The GPU implementation is also suitable for offline simulations where real time is not an issue.

The timings for both the CPU and the GPU version are taken just before the execution branches to the separate versions until just after they are back at the same branch. This means that the timing of the GPU implementation includes the CPU sending the bounding volume hierarchy for both objects, and after computations are done receiving the results. It should be taken into consideration that a timing of an implementation storing all bounding volume hierarchies on GPU memory will not include that initial memory transfer as it happens only once when the object is initialized.

	Intersection tests executed per depth			
Depth	armadillo	armadillo_2	armadillo_3	armadillo_4
6	4 096	4 096	4 096	4 096
7	332	348	396	456
8	536	624	704	848
9	1 040	1 176	1 400	2 000
10	2 020	2 476	3 012	4 080
11	4 236	5 132	6 364	6 136
12	9 872	12 760	16 032	1 648
13	22 016	29 040	38 768	84
14	56 792	77 560	66 320	0
15	138 360	192 696	29 732	0
16	366 040	390 780	1 364	0
17	824 020	252 912	0	0
18	332 704	19 560	0	0
19	58 190	376	0	0
20	2 890	0	0	0
21	66	0	0	0
Total	1 823 210	989 536	168 188	19 348

Table 6.3: *Intersection tests executed each depth during the GPU middle phase algorithm while colliding armadillo objects of different quality.*

For a parallel implementation of middle phase, a big problem set of many parallel executions is important. The first iterations of the algorithm is the crucial part, as in the first iteration there is just one bounding volume intersection test, and at best they can grow by a factor of four for each iteration. As can be seen in table 6.3 the algorithm starts at a depth of six to avoid these first few iterations where full parallelism cannot be achieved. However, as clearly non-intersecting bounding volumes have not been removed by testing for their parents collision, it can be seen that the number of positive intersections in the next iteration is much lower. Despite this, starting at a later depth is beneficial relative to starting at depth zero.

For the best parallel performance, an execution where as much time as possible is spent in iterations with enough elements to maximize the performance of the GPU is the key. From table 6.3 one can see that for the scenes with more detailed objects (and also the scenes exhibiting the most speedup compared to CPU), most of the elements computed are in iterations with large arrays of elements. How many elements (or how large global work-size) there should be to get the maximal performance varies for different GPUs. The GeForce 8800 GTS for example, has 12 multiprocessors. They can each serve one warp simultaneously. That results in 384 threads executed in parallel. However, to hide latencies from memory and register access, Nvidia recommends a work-group size of at least 64 threads, and a total of at least 192 or 256 threads per multiprocessor. This results in at least 2304 - 3072 threads, and more will certainly not hurt.

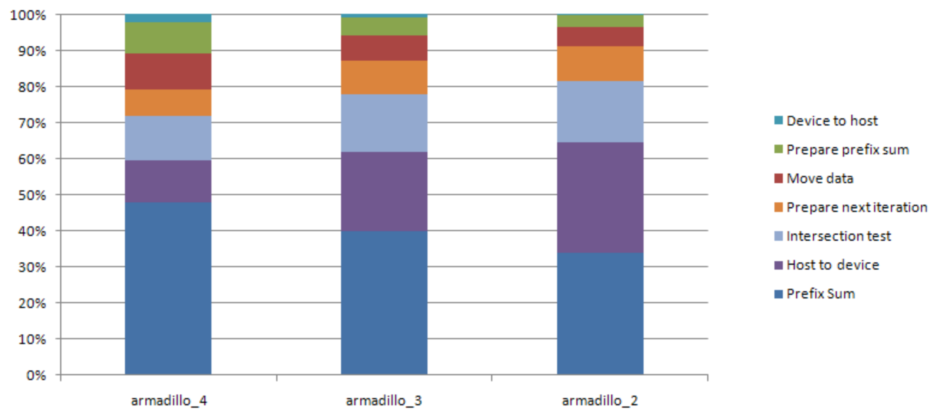

Figure 6.2: *Time consumption of the different parts of GPU middle phase for various test scenes, shown as a 100% stacked column.*

In Figure 6.2 the different parts of the algorithm and the relative time those parts take in execution of the test scenes are shown. Here can be seen that prefix sum is the most costly part, as it takes up to 30-50% of the execution time. Sending the bounding volume hierarchies from CPU to GPU every time the kernel executes is time consuming, and one can see that for larger objects the time relative to the total time increases considerably. The intersection tests of the bounding volumes, being the main part of the algorithm, takes relatively little time to execute. Instead it is the traversal of the hierarchies that is the most time consuming.

6.3 Near Phase

Scene	Triangle pairs
armadillo	12430
armadillo_2	9372
armadillo_3	4746
armadillo_4	1421

Table 6.4: *Number of triangle pairs being tested for collision in the test scenes*

Scene	CPU time (ms)	GPU time (ms)	Speedup
armadillo_4	1.51	1.90	0.79
armadillo_3	5.18	4.23	1.22
armadillo_2	10.78	6.69	1.61
armadillo	15.62	8.29	1.89

Table 6.5: *Comparison between near phase CPU and GPU implementations.*

In table 6.5 it can be seen that for intersections of objects made of fewer triangles, the CPU implementation is the fastest. The more detailed the objects get, the better the GPU implementation becomes compared to the CPU implementation. As can be seen, the speedup of GPU compared to CPU even for very detailed objects is not that big. One reason for this is that register usage is high. This tends to happen for longer kernels where intermediate results has to be stored. The high register usage limits the occupancy of the graphics card, limiting the number of simultaneous threads the kernel can keep active. Another big hit to performance that happens when register usage is high, is that the compiler cannot put new data in registers. Instead, for CUDA devices, it is put in off-chip memory where access times are the same as for global memory. The difference in access time is huge compared to register access. In the finding of a contact for a triangle-triangle intersection, there are some loops where the number of iterations depend on earlier steps. This causes branches to diverge, hampering performance. The final issue with this implementation is that before sending data to GPU, the memory must be allocated and filled with the triangles corresponding to the ones to be tested.

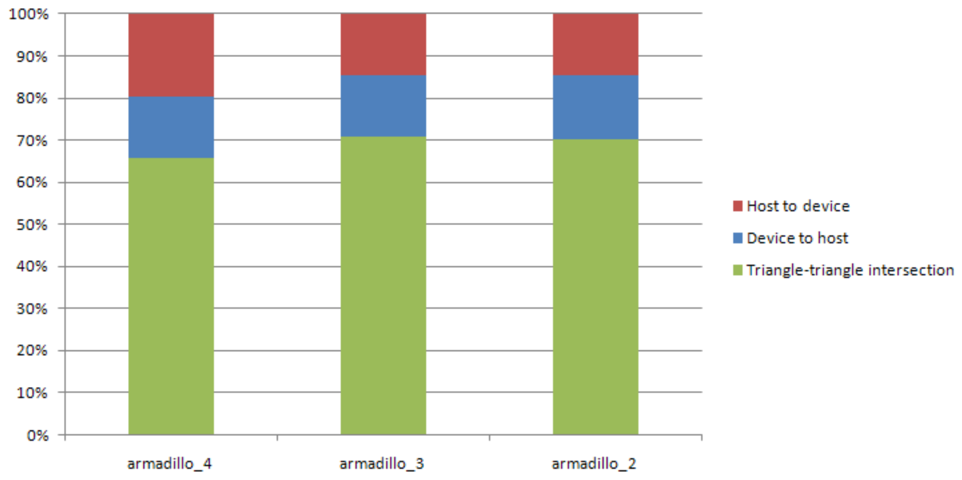

Figure 6.3: *Time consumption of the different parts of GPU near phase for various test scenes, shown as a 100% stacked column.*

Figure 6.3 shows how the execution of the intersection kernel is by far the largest influence on performance, but both sending triangles to the GPU and returning the resulting contacts play a significant role.

Parallelizing near phase of several triangle mesh objects together was not implemented, but is very simple. It only requires a pipeline accommodated to it by first executing middle phase for all intersections and then work on all the resulting triangle pairs. This will minimize the potential impact on performance of doing many kernel executions with smaller work sizes.

6.4 Importance of Middle Phase

Without a proper middle phase algorithm, the number of exact triangle-triangle tests would be $n_1 * n_2$ where n_1 and n_2 are the number of triangles for each object. For our tests scenes, this takes the number of exact triangle-triangle intersections down from $1.2*10^{11}$ to $1.2*10^4$ for the most detailed armadillo objects colliding and from $7.7*10^6$ to $1.4*10^3$ for the least detailed armadillos. This reduction is accomplished by only executing around $1.8*10^6$ and $1.9*10^4$ bounding volume intersection tests (see table 6.3) for the two scenes, a very good tradeoff.

As can be seen from inspection of tables 6.2 and 6.5, middle phase is by far the most time consuming phase. Thus, it is of high importance to have a quick and accurate middle phase algorithm, where accurate means returning as few false intersections as possible, as that will unnecessarily impact the time in near phase.

Chapter 7

Conclusions

7.1 Restrictions

Because graphics processors have not had a need for double precision floats historically, even though general programming on GPUs have grown much the last years, the hardware is not yet there in terms of performance for double precision floats. Because of this, the user is restricted to single precision floating points.

As for all primitives used in a discrete physics simulation, a small object of high speed can tunnel through another object between two discrete time steps, which leads to a correct impact collision being impossible to find. This can be a problem for the triangle mesh collision detection algorithm implemented in the thesis work, since a detailed mesh can consist of very small triangles. If such triangles tunnel through an object, there is a possibility that objects get stuck in each other, since there are triangles generating contacts on each side of the tunneled object, all pointing ”away” from the object.

Also, as mentioned in Section 5.3.1, manifold triangle meshes must be used for collision detection.

7.2 Limitations

The solution presented here and developed during the period of the thesis work had its focus on triangle meshes colliding with other triangle meshes. For a physics engine to be versatile and broad, one wants contact information in the case of triangle meshes colliding with any other primitive, such as plane, box or sphere. The limited time prevented development of such colliders. Before having triangle meshes fully usable in AgX, such colliders have to be done.

7.3 Future work

There are a number of things that can be further developed that were outside of the scope of the thesis work or that had to be dropped due to time limitations or priorities; the most important of these being the, in above section mentioned, lack of colliders between triangle meshes and other primitives.

Other areas where more work is possible is developing better interaction between the CPU and GPU solutions. For smaller problems where the parallel aspect of the GPU is not fully utilized, the program could decide whether to use the CPU or the GPU implementation. To fully utilize both the CPU and the GPU, support for using them both simultaneously, dividing the work between them in an efficient way, can also be investigated.

For the GPU implementation to be effective in its current state, large problem sets are needed. This is a concern especially for the middle phase GPU implementation, as there is a big difference between several collisions involving less detailed objects and one collision involving two higher detailed objects. Because each collision is solved sequentially, the GPU can not make full use of its parallel structure in the case of several collisions of less detailed objects. A segmented middle phase implementation, where several collisions are solved simultaneously, would therefore be interesting to evaluate. Extra data and more computations would be needed to keep track of what belongs to which segment, but it is very likely that a successful implementation of segmented middle phase would cut the computation times of many possible scenarios.

For the GPU near phase implementation, the main bottleneck is register shortage. A complete rewrite of the near phase implementation, with focus on lowering the register count, could prove extremely effective for performance. Dividing the implementation into several smaller kernels would also most likely lead to a more restricted usage of registers at the cost of kernel initialization time, a tradeoff that could prove to be very effective.

The time spent of the middle phase implementation in the prefix sum algorithm was quite extensive. Although the implementation of prefix sum done during this thesis work was optimized for parallel execution, it could most likely be even more effective with small corrections and improvements. By porting the implementation of prefix sum included in CudPP[13] (a C for CUDA library) to OpenCL, it is likely that a speedup during the prefix sum phase can be achieved.

Chapter 8

Acknowledgements

I would like to thank everyone at Algoryx for being very welcoming towards me and helpful in general. Especially I would like to thank Michael Brandl, my supervisor at Algoryx, for working with me on this project, for the daily discussions and for reading and commentating on the report. I would also like to thank my supervisor at Umeå University, Daniel Sjölie, for the help with structuring and reading the report.

References

[1] Blender. `http://www.blender.org` [October 10, 2010].

[2] Algoryx. Agx multiphysics. `http://www.algoryx.se/products.html` [October 9, 2010].

[3] Chanderjit L. Bajaj and Tamal K. Dey. Convex decomposition of polyhedra and robustness. *SIAM J. Comput.*, 21(2):339–364, 1992.

[4] E. Coumans. Bullet physics. `http://www.bulletphysics.com` [October 10, 2010].

[5] C. Ericson. *Real Time Collision Detection.* Morgan Kaufmann Publishers, 2005.

[6] S. Fisher and Ming C. Lin. Deformed distance fields for simulation of non-penetrating flexible bodies. In *Proceedings of the Eurographic workshop on Computer animation and simulation*, pages 99–111, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[7] A. Fournier and D. Y. Montuno. Triangulating simple polygons and equivalent problems. *ACM Trans. Graph.*, 3(2):153–174, 1984.

[8] E. Gilbert, D. Johnson, and S. Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics and Automation*, 4(2):193–203, 1988.

[9] S. Gottschalk, M. C. Lin, and D. Manocha. Obbtree: a hierarchical structure for rapid interference detection. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 171–180, New York, NY, USA, 1996. ACM.

[10] E. Greß, M. Guthe, and R. Klein. Gpu-based collision detection for deformable parameterized surfaces, 2006.

[11] Khronos Group. Opencl. `http://www.khronos.org/opencl/` [October 10, 2010].

[12] E. Guendelman, R. Bridson, and R. Fedkiw. Nonconvex rigid bodies with stacking. *ACM Trans. Graph.*, 22(3):871–878, 2003.

[13] M. Harris. Cudpp. `http://code.google.com/p/cudpp/` [October 10, 2010].

[14] W. Daniel Hillis, Jr. Steele, and Guy L. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

[15] Khronos. *The OpenCL Specification.* Khronos OpenCL Working Group, 1.0 edition.

[16] J. Klosowski et al. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, 1998.

[17] F. Leon. Gimpact collision detection. `http://gimpact.sourceforge.net` [October 10, 2010].

[18] Microsoft. Directcompute. `http://msdn.microsoft.com/en-us/directx/default.aspx` [October 10, 2010].

[19] T. Möller. A fast triangle-triangle intersection test. *J. Graph. Tools*, 2(2):25–30, 1997.

[20] A. Moravanszky and P. Terdiman. Fast contact reduction for dynamics simulations. *Game Programming Gems 4*, pages 253–263, 2004.

[21] K. Myszkowski, O. Okunev, and T. Kunii. Fast collision detection between complex solids using rasterizing graphics hardware. *Visual Computer*, 11(9):497–511, 1995.

[22] Nvidia. C for cuda. `http://developer.nvidia.com/object/gpucomputing.html` [October 10, 2010].

[23] Nvidia. *Nvidia OpenCL Best Practices Guide*. Nvidia, 2.3 edition.

[24] Nvidia. *Nvidia's Next Generation CUDA Compute Architecture: Fermi*.

[25] Nvidia. *OpenCL Programming Guide for the CUDA Architecture*. Nvidia, 2.3 edition.

[26] J. O'Rourke and K. Supowit. Some np-hard polygon decomposition problems. *IEEE Transactions on Information Theory*, 29(2):181–190, 1983.

[27] S. Redon, A. Kheddar, and S. Coquillart. Fast Continuous Collision Detection between Rigid Bodies. *Computer Graphics Forum*, 2002.

[28] L. Seiler et al. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[29] G. Snethen. Xenocollide: Complex collision made simple. In *Game Programming Gems 7*, pages 165–178. 2008.

[30] Stanford. Stanford armadillo. `http://graphics.stanford.edu/data/3Dscanrep/` [October 10, 2010].

[31] P. Terdiman. Opcode, optimized collision detection. `http://www.codercorner.com/Opcode.htm` [October 10, 2010].