Collision Detection Using Hierarchical Grid Spatial Partitioning on the GPU

by

Ryan R. Kroiss

B.S., University of Wisconsin - Madison, 2009

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Computer Science

2013

This thesis entitled:
Collision Detection Using Hierarchical Grid Spatial Partitioning on the GPU
written by Ryan R. Kroiss
has been approved for the Department of Computer Science

_____
Prof. Willem Schreüder

_____
Prof. Ken Anderson

_____
Prof. Roger King

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Kroiss, Ryan R. (M.S., Computer Science)

Collision Detection Using Hierarchical Grid Spatial Partitioning on the GPU

Thesis directed by Prof. Willem Schreüder

Collision detection is a popular topic in computer graphics. Any physics-based application (e.g. surgical simulations, robotics, video games, computer animations, etc.) necessitates some degree of collision detection and handling. Many techniques exist for performing collision detection. Until recently, the majority of them rely on CPU-based approaches. The increasing accessibilty of general purpose GPU (GPGPU) programming has marshalled in new possibilities for computing collision detection in virtual environments. However, GPGPU programming is certainly not a silver bullet. There are a variety of challenges to recognize and overcome, in particular, as it relates to collision detection. This thesis will explore existing methods for performing collision detection and present a novel GPU-based approach to a common technique - hierarchical grid spatial partitioning.

Dedication

To my wife Janna and the rest of my family for all of their patience and support.

Acknowledgements

I would like to thank my advisor, Professor Willem Schreüder, for introducing me to the field of computer graphics and for his support and guidance during this thesis. I would also like to thank my committee members, Professors Ken Anderson and Roger King, and everyone else who reviewed this thesis.

Contents

Chapter

Figures

Figure

Chapter 1

Introduction

In a physics-based virtual environment, whether it is robotics, surgical navigation, flight simulation, computer animation, or video games, accurate and efficient simulation of physical interactions is critical. A major part of simulating a physics-based environment is dealing with colliding bodies. The interaction between potentially colliding bodies is called collision handling. It consists of three main phases: collision detection, collision determination, and collision response [6]. Collision detection provides an indication of whether or not two objects are colliding. Collision determination calculates the actual points of intersection between the colliding objects. Collision response resolves the collision into a physical response (e.g. reflecting away from one another). This thesis will explore existing methods for performing collision detection and present a novel GPU-based approach to a common technique - hierarchical grid spatial partitioning.

1.1 Collision Detection Overview

Collision detection is a complex process, which is often broken down into phases: broad and narrow phase collision detection. Broad phase collision detection is a coarse pass over the objects in the scene. Rather than looking at the actual geometry of each object when performing the collision calculation, broad phase collision detection will look at a bounding volume that provides a crude approximation of each object's shape. The result of this phase is a list of objects that are potentially colliding - the *potentially colliding set* or *PCS*. By pruning the list of objects down to the PCS, the number of fine-grained collision calculations that must be performed on actual object

geometry in narrow phase can be significantly reduced.

If the broad phase collision detection steps determine that the bounding volumes of two objects are indeed colliding, narrow phase collision detection will then check the actual geometry of the objects and determine if the collision determination and response phases are necessary. Narrow phase collision detection is beyond the scope of this thesis. For further information, the Lin-Canny Closest Features algorithm [16] and the Gilbert-Johnson-Keerthi (GJK) algorithm [12] provide a good foundation of narrow phase collision detection.

Occasionally, mid phase collision detection is used between broad and narrow phase. This is an optional phase that is primarily used in scenes with complex geometry (e.g. non-convex and/or highly tessellated objects). In such scenes, computing tight fitting bounding volumes can be expensive, so loose fitting bounding volumes may be used during the broad phase. The mid phase then performs a more fine-grained check that further reduces the size of the PCS. [17]

1.2 Bounding Volumes

Broad phase collision detection relies heavily on the use of bounding volumes, which are convex shapes that are used to approximate the geometry of a more complex object. They range in geometric complexity from spheres or rectangular prisms to discrete-orientation polytopes. A survey of commonly used bounding volumes is presented by Ericson [11] and summarized here. See figure 1.1 for examples.

Spheres as bounding volumes are fairly self-explanatory. They are transformed in such a way that they contain the geometry of the target object. Bounding spheres are desirable because they are simple to compute, require very little storage, and have simple collision tests. However, they suffer from the fact that they are often loose fitting compared to other bounding volumes.

Axis-aligned bounding boxes (AABBs) are simply rectangular prisms (boxes or cuboids) formed around the contained object where each of the faces is parallel to one of the coordinate planes. AABBs share similar advantages and disadvantages with bounding spheres. However, they often offer a tighter fit, but require slightly more storage in memory.

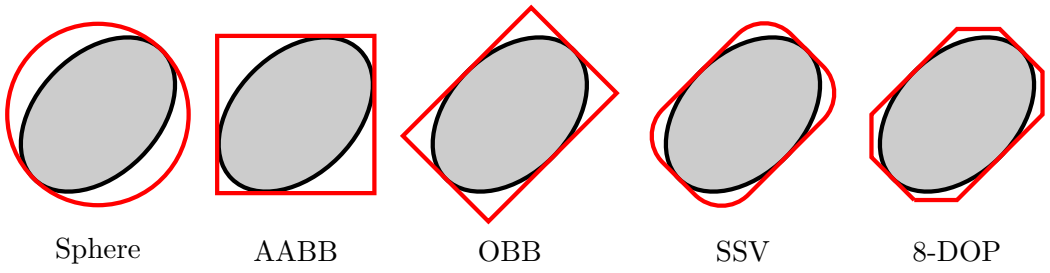

Figure 1.1: Spheres, axis-aligned bounding boxes (AABBs), oriented bounding boxes (OBBs), sphere-swept volumes (SSVs), and discrete-oriented polytopes (k-DOPs) are types of bounding volumes used to provide a crude approximation of object geometry. This figure illustrates a two dimensional example of each type.

Oriented bounding boxes (OBBs) are generalizations of AABBs in that they can be oriented in any arbitrary way. Usually, the orientation is chosen in such a way that it offers a tight bound to the contained geometry. OBBs are slightly more difficult to compute, require more storage space, and have more complex collision tests due to the arbitrary orientation but often offer a better bound than spheres and AABBs.

Sphere-swept volumes (SSVs) can take several forms: sphere-swept points, sphere-swept lines, and sphere-swept rectangles. Sphere-swept lines are often referred to as capsules and are formed by sweeping a sphere along a line segment. Sphere-swept rectangles (also known as lozenges) are formed by sweeping a sphere around a rectangle with an arbitrary orientation. They can offer tighter bounds than previous bounding volumes, but require more storage space, are more difficult to compute, and have more complex intersection tests.

Finally, discrete-orientation polytopes (k-DOPs) are generalizations of AABBs. They use k arbitrarily defined planes to compute a convex polytope as a bounding volume. Initially, each of the k planes is positioned at infinity and moved closer to the object until a tight fitting polytope is formed. An AABB is a 6-DOP with two planes defined to be parallel for each of the coordinate planes. Depending on the complexity of the k-DOP, they can be expensive to compute, store, and test for collisions. However, they often form a tighter bound around the object than other bounding volumes.

It is important to note that using bounding volumes alone in collision detection only offers a constant factor improvement. The true performance benefit is gained when bounding volumes are used in conjunction with an efficient broad phase collision detection technique that can reduce the size of the PCS.

1.3 Survey of Broad Phase Collision Detection Techniques

The goal of broad phase collision detection is to reduce the number of pairwise collision tests that must be performed. The methods for doing this can be broken up into three categories: object-centric, space-centric, and rendering-based. Object-centric and space-centric approaches attempt to partition the problem space into more managable pieces. They differ in the way that they subdivide the problem. Rendering-based approaches are quite different in that they attempt to perform collision detection in image-space using the rendering process.

1.3.1 Object-centric Approaches

Collision detection methods that fall into the object-centric category focus on organizing objects in such a way that they can efficiently perform collision queries. The most common object-centric approaches either use bounding volume hierarchies or the sort and sweep technique. Bounding volume hierarchies organize the bounding volumes of objects into a tree and use the resulting tree to perform collision queries. The sort and sweep technique sorts objects based on their position along various different axes and uses the result to find potential collisions.

1.3.1.1 Bounding Volume Hierarchies

Bounding volume hierarchies (BVHs) can reduce the number of pairwise collision tests performed by organizing bounding volumes into trees. The key idea with BVHs is to build a tree based on the bounding volumes of the objects in such a way that the leaf level contains the bounding volumes and the root level specifies a bounding volume that encapsulates all objects. It is worth noting that with BVHs a given non-leaf node does not necessarily need to contain the bounding

volumes of its children. That is to say, there is no guarantee that a non-leaf node will contain the bounding volumes of its children, only that it will contain the objects themselves.

There are two key operations with bounding volume hierarchies: construction and traversal. Constructing a BVH can happen in one of three ways: top-down, bottom-up, and insertion. In the top-down approach, the tree starts with a node that encapsulates all of the objects. From that point, the objects are split up and assigned to a child node. This happens recursively until all objects have been assigned as a leaf node. This method is relatively easy to implement. The disadvantage is that the resultant tree is often suboptimal.

The bottom-up construction method starts by grouping objects together. Again, all objects remain at the leaf level. When objects are grouped together, a bounding volume is created that contains all of the objects. This occurs recursively until all objects are contained in one bounding volume. The bottom-up approach really brings to light the possibility of whether non-leaf nodes should encapsulate the bounding volumes of its children or just the geometry of its children. The latter would likely require more computation but could result in more tight-fitting bounding volumes. In contrast to the top-down approach, the bottom-up approach is more difficult to implement, but generally results in better trees.

Finally, the insertion method iterates over all of the objects and inserts them into the tree one at a time. With each insertion the tree must grow in some fashion. The key to this method is to insert objects in such a way that tree growth is minimized. This can be done by using heuristics to estimate tree quality. This approach is less commonly used in practice. It offers the advantage of easy update. However, one consideration here is that insertion order does matter.

Since siblings within a BVH can potentially overlap, it is not clear how the tree should be traversed on a collision query. Breadth-first search (BFS) and depth-first search (DFS) are the most common graph traversal algorithms. Either can be used effectively with BVHs. However, DFS is more popular [11]. An alternative to DFS and BFS is a more informed search. In a well-constructed BVH, nodes that are near each other in the tree should contain objects that are near each other in space. This knowledge can be used to terminate searches more quickly than would be possible

with DFS or BFS.

There are many design considerations with BVHs. The BVH can be manually constructed as the objects are placed in the scene, or it can be automatically constructed using one of the methods above. As with any data structure, storage and performance considerations become a factor. If available memory is limited, that may influence the BVH. Further, the time required to construct, maintain, and traverse the tree must be relatively low for the BVH to be effective. The degree of the tree is also a major consideration. Binary trees are typically easiest and most commonly used, but a tree of a higher degree could be advantageous in certain situations. Some example implementations of BVHs include OBBTree [13], BoxTree [25], and the AABB tree described by Bergen [9].

1.3.1.2 Sort and Sweep

One major downside to BVHs is that there is a high cost of initialization in constructing the hierarchy. The sort and sweep method resolves that by simply using the list of bounding volumes. This method keeps sorted lists of bounding volumes based on their position. Bounding volumes are projected onto some axis. Each bounding volume's projection has a start and end position along that axis. Objects are then sorted based on their starting position along the axis. The sorted list then allows for efficient collision checking. Any particular object can be checked for collisions by examining its neighbors for overlap along the given axis. The disadvantage of this method is that it can perform poorly when objects are clustered along any particular axis.

1.3.2 Space-centric Approaches

Since objects will only collide if they share the same approximate space, the space-centric approaches attempt to reduce the problem by partitioning the scene in some way. This is often referred to as *spatial partitioning* or *spatial subdivision*. Objects in the scene are then associated with a particular partition or partitions. Objects that reside in the same partition are checked for collisions. This can greatly reduce the number of explicit collision checks that must be performed.

Space-centric approaches typically use grid or tree structures to represent the scene.

1.3.2.1 Uniform Grids

The uniform grid approach to spatial partitioning is simple, yet effective. The scene is subdivided into grid cells of equal size. After subdividing the scene, each bounding volume is then stored in all cells that it overlaps. Identifying which cells a bounding volume overlaps is done using spatial hashing. During the collision detection stage, all bounding volumes that reside in a particular cell are checked against each other. Any bounding volumes that are colliding are then added to the PCS for further examination in the narrow phase.

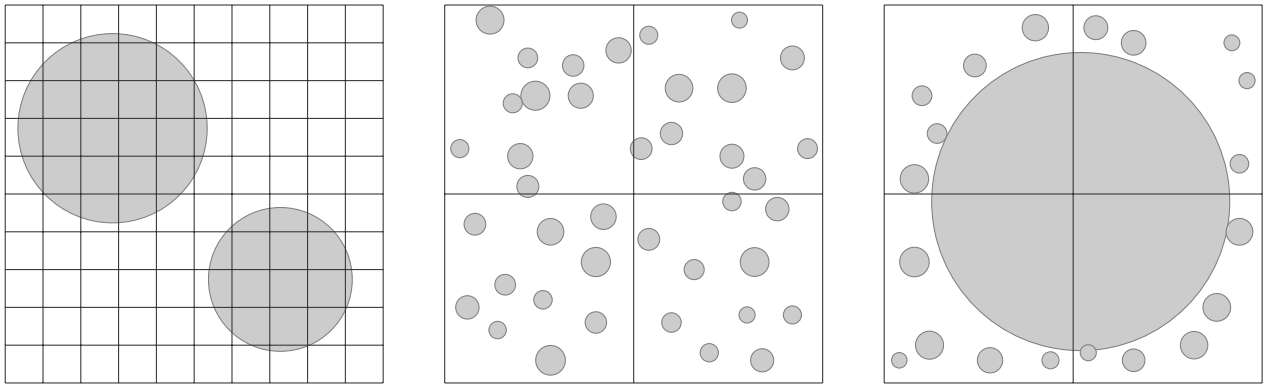

Figure 1.2: Using a uniform cell size for spatial partitioning can be problematic in certain situations. In (a) the grid is too small with respect to object size. In (b) the grid is too coarse with respect to object size. In (c) the grid is too coarse for the small objects but is too fine for the large objects.

Arguably the most critical part of uniform grid spatial partitioning is choosing an appropriate cell size. Ericson [11] presents some of the issues related to choosing cell size, some of which are represented in figure 1.2. When choosing a cell size that is too small, a potentially excessive amount of memory may need to be allocated to contain the grid. Also, large objects could overlap many cells, which could degrade performance. Conversely, if the cell size is too large, performance could

be negatively impacted because many objects could reside in each cell resulting in an unnecessary number of collision checks. Since real world problems typically contain a large range of object sizes, the uniform grid makes it difficult to select a near optimal grid size. Many methods attempt to alleviate these issues. One of the alternatives is the hierarchical grid.

1.3.2.2 Hierarchical Grids

Hierarchical grid (or *hgrid*) spatial partitioning schemes contain multiple overlapping grids with different cell sizes for each grid, rather than one grid with uniform cell sizes as is the case with uniform grids. The multiple overlapping grids make object tracking slightly more complicated. Objects can now overlap multiple cells within multiple grids. In a basic approach (see figure 1.3), each bounding volume is inserted into the grid level defining the smallest cell size in which the bounding volume can fit. The bounding volume is only inserted into one cell - the cell where its centroid resides (or one of the corners in the case of a box). By confining objects to one cell on one level, insert and delete operations are fast. However, collision checks then need to include neighboring cells on the same level and all potentially overlapping cells at all other levels since objects contained in those cells could overflow into adjacent cells.

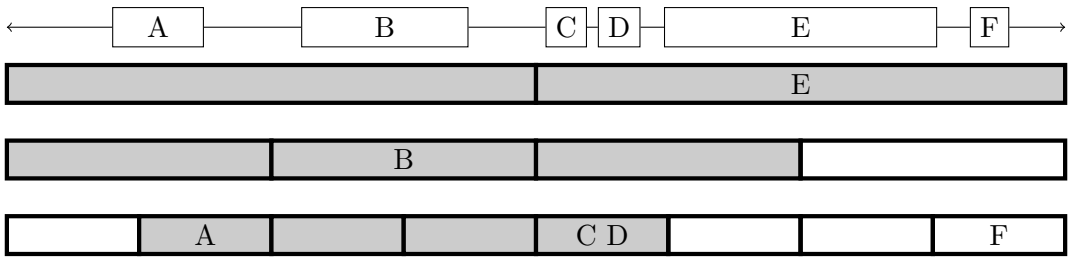

Figure 1.3: This figure demonstrates a one dimensional example of a basic implementation of hierarchical grids. The objects contained in the scene are represented on the top row. The hierarchical grid is represented by the bottom three rows. Each object is inserted on the level with the smallest cell size into which the object fits. Further, each object is inserted into only one cell - the cell where its centroid resides. This can be seen with object A which overlaps two cells on the level into which it is inserted but is only inserted into the cell containing its centroid. The cells that must be checked when a collision check is performed for object B are shaded.

Ericson [11] describes the relationship of object size to cell size and the relationship of cell sizes between levels. Often, the size of the largest object that is inserted into a level is not equal to the cell size at that level. Rather it is some fraction of that size. When that fraction is smaller, fewer neighboring cells need to be examined at a given level on average. As a consequence, it is likely that the average number of objects per cell also increases. When choosing cell sizes for each grid level, the cell sizes of adjacent grid levels often differ by some constant factor.

The basic implementation described so far has some major problems. When objects are tested one at a time, large objects may need to be checked against many cells. Also, depending on how the grid is structured, a large amount of memory can be consumed by the grid. Mirtich presents several solutions [18, 19] to these issues. These will be discussed in section 2.2.

1.3.2.3 Trees

Section 1.3.1.1 discussed how trees can be used to organize objects into bounding volume hierarchies. Trees can also be used for spatial partitioning. Spatial partitioning trees differ from BVHs in that BVHs only contain bounding volumes at the leaf level while spatial partitioning trees can contain bounding volumes in any node within the tree. Octrees and k-d trees are the most commonly used methods for spatial partitioning using trees.

Octrees (or quadtrees in two dimensions) start at the root level with a tight fitting bounding volume (generally axis-aligned) that contains all of the objects in the scene. That bounding box is then subdivided into eight evenly sized cubes (or four evenly sized squares in two dimensions). The subdivision continues until some stop criterion is met (e.g. a certain depth is reached or the cubes at the leaf level have exceeded a certain minimum threshold in size). [11] Typically objects are inserted into a node in the tree in which they entirely fit. This can be problematic as it can result in small objects residing in levels that far exceed their size. This happens when a small object falls right along the boundary of a subdivision. This problem can be avoided if the recursive subdivision step is relaxed to allow each of the subdivisions to overlap by some amount. An octree built in this way is often called a *loose octree* [24]. Loose octrees closely resemble hgrids. The benefit of the

hgrid here is that it does not contain the overhead of the higher levels of the tree.

Another approach to spatial subdivision using trees is the k-d tree [8]. The k-d tree is a generalization of the octree in that it subdivides in k dimensions. In octrees, each subdivision simultaneously splits the space evenly along the x, y, and z axes. In contrast, the k-d tree iteratively splits space along one of the k axes (e.g. x then y then z, etc.). Also, the split along any particular axis can be at an arbitrary position. From this it is easy to see how octrees are just a specific implementation of the k-d tree.

1.3.2.4 Binary Space Partitioning Trees

Binary space partitioning (BSP) trees are yet another tree-based method for collision detection. This method is similar to the k-d tree except that it is even more generalized. The scene is recursively divided into pairs of subspaces. The dividing planes can have arbitrary position and orientation. BSP trees can take many forms and be used for many different purposes, and are beyond the scope of this thesis.

1.3.3 Rendering-based Approaches

All of the collision detection methods discussed up to this point have dealt with object geometry or the associated bounding volumes. They attempt to structure the search space by either subdividing the scene or grouping the objects. Another set of approaches performs collision detection in image-space during the rendering process. The main advantage with this approach is that it utilizes the highly parallel nature of the GPU, whereas the majority of the other methods presented are generally performed on the CPU. Rendering-based collision detection is either done by using buffer read back calls or by performing occlusion queries.

In the buffer read back approach, the scene is rendered in an orthographic projection with depth testing turned on. After rendering, the depth buffer is then read back and used to determine if any objects are colliding. However, this method is rarely used anymore because of the large amount of data that must be transferred back and forth between CPU and GPU memory, which

is an expensive operation.

The more common approach is to use hardware occlusion queries, which are used to determine the visibility of an object. Typically, occlusion queries are used with a *less than or equal* depth test. In this way, the occlusion query counts the number of pixels (or, more precisely, fragments) that pass the depth test. If none of the fragments pass, then the object does not need to be rendered because it is fully occluded with respect to the contents of the depth buffer [6]. *CULLIDE* [14] uses a similar process for performing collision detection. In this approach, the depth test is set to *greater than or equal* and depth buffer updates are turned off. By reversing the depth test, occlusion queries reporting that no fragments pass the depth test indicate that the object is fully visible with respect to the depth buffer (i.e. it is not colliding with any other objects).

To use this technique for a broad phase collision detection algorithm, CULLIDE renders the scene using an orthographic projection looking down each axis: x, y, and z. For each axis, the scene must be rendered twice (for a total of at least six render passes). On the first pass, the objects are rendered in order from 1 to N. On the second pass, the objects are rendered in reverse order - from N to 1. After rendering each object, an occlusion query is performed. If the query determines that the object is not occluded, then it does not collide with any of the previously rendered objects.

This approach suffers from several factors. It is dependent on the resolution of the buffer used for rendering. If the resolution is low, more collisions may be reported than necessary. The accuracy of the depth buffer comes into play as well, in particular, when using buffer read back calls. Finally, the rasterizers can affect collision detection accuracy. Standard rasterizers may not fill a pixel if only a small portion of the object geometry is contained in the pixel. Many of these drawbacks can be mitigated by rendering the scene at an appropriate scale and using conservative rasterizers. [6]

1.3.4 Summary

A variety of approaches for broad-phase collision detection have been shown along with their advantages and disadvantages. The object-centric approaches seek to organize objects in such a

way that collision queries can be performed efficiently. The various derivations of bounding volume hierarchies and sort and sweep techniques take advantage of this approach. The space-centric group of approaches seek to partition space in such a way that collision queries can be performed efficiently. The various grid-based approaches (i.e. uniform and hierarchical) and spatial partitioning trees (e.g. octrees, k-d trees, BSP trees, etc.) utilize this method. Finally, collision detection can also be performed in image space using rendering-based techniques. This thesis seeks to combine hierarchical grid spatial partitioning with the parallelism gained in the rendering-based approaches by harnessing the GPU.

Chapter 2

Related Work

Of the three main areas of broad phase collision detection that have been discussed, this thesis focuses on spatial partitioning. More specifically, it focuses on performing broad phase collision detection on the GPU using hierarchical grid spatial partitioning. To achieve this goal, this thesis examines a variety of approaches from broad phase collision detection and tries to pull the promising aspects together.

2.1 Uniform Grids

Le Grand [15] presents a uniform grid spatial partitioning approach on the GPU using CUDA [3]. His approach uses a cell size for the uniform grid that is some small factor greater than the size of the largest object in the scene. With this prerequisite in hand, it can be inferred that an object can reside in at most 2^d cells. Further, it guarantees that two objects only require a pairwise collision check if they appear in the same cell *and* at least one or both of their centroids reside in that cell. In preparation for the collision detection algorithm, two separate arrays are created. One array contains identifiers that associate each element with an object - the *object array*. The other array contains information about each cell in which an object resides. Each array contains $2^d * N$ elements (an element per object for each cell in which it can reside). A key concept here is the cell ID, which is contained in the *cell ID array*. The cell ID for a cell is computed using a spatial hash of a given d-dimensional location. Le Grand also presents the concept of *cell type* and *control bits*. The *home cell* of an object is the cell in which the centroid is contained. A *phantom cell* is any

other cell in which an object resides that is not also the home cell. All of these concepts help to parallelize the collision detection and prevent multiple checks from being performed on the same objects. After all of the set-up work has been completed, a parallel radix sort [10] is performed on the arrays. A key-value sort is used where the keys are the elements of the cell ID array and values are the elements of the object array. After the cell ID array has been sorted, it is then scanned for transitions in cell IDs. Any section of the cell ID array where multiple consecutive cell identifiers occur indicates a potential *collision cell* - a cell where multiple objects reside. Finally, a thread can be launched per collision cell. Each of these threads checks for collisions amongst the objects in the cell. Any collisions are noted and become part of the PCS.

Pabst [20] presents several problems with the approach presented by Le Grand. The overall approach is a hybrid CPU/GPU collision detection system for rigid and deformable surfaces. The broad phase portion uses a uniform grid spatial partitioning much like Le Grand's implementation. After sorting the cell ID array, a calculation of the number of potential collisions is made, and a thread is launched for each of those potential collisions. Launching a thread per potential collision rather then per collision cell can drastically reduce the likelihood of divergence.[1] Further, a strict ordering of the collision checks is enforced to restrict the number of collision checks that must be performed. Even though the likelihood of divergence is reduced, global memory accesses can be a problem when checking for collisions between two objects. To get around this, a preprocessing step is performed to organize objects that will be used by a particular thread into a small data structure. This data structure is then loaded into shared memory and can be efficiently accessed when performing collision detection.

2.2 Hierarchical Grids

In section 1.3.2.2, a basic implementation of hierarchical grid spatial partitioning was presented. In that approach, each object was inserted into one cell on one level - the smallest level in which it would fit. Objects must then be checked against possibly overlapping cells at all other

[1] Divergence and global memory access issues are discussed in appendix A.

levels in the hierarchy. Two alternative approaches are presented by Mirtich [18, 19]. The first scheme presented by Mirtich [18] sorts objects in descending order based on size. As objects are inserted they only need to be tested against objects at their own level or higher. Lower levels do not need to be checked at this point because they will be empty due to the insertion order. A major problem with this is that it does not handle dynamic updates well.

The second scheme presented by Mirtich [19] inserts objects into multiple cells. An object is inserted into all cells that it overlaps in all levels in which it would fit. In this scheme, determining if two objects overlap can now be done by testing if they share a common cell at level $L = Max(Level(A), Level(B))$, where $Level(P)$ is the smallest level in which the bounding volume of object P will fit. This effectively tests if the bounding volumes overlap on the larger of the two initial insertion levels for each object. [11] This test requires that information about each pair of objects be tracked. When object A is placed into a cell that also contains object B, a counter corresponding to the A, B pair is incremented and the pair is added to a tracking data structure. When A moves out of the cell containing B, then the A, B counter is decremented. Narrow phase collision detection only needs to be computed on object pairs that are in the tracking data structure. In this scheme, moving an object is expensive, but it has the benefit that fewer cells need to be tested.

Chapter 3

Implementation

This thesis will explore using hierarchical grid spatial partitioning for broad phase collision detection. As explained in section 1.3.2.1, real world problems often contain objects of widely varying size. Such scenes can present problems for uniform grid spatial partitioning. Hgrids attempt to solve these problems using a set of overlapping grids with varying cell sizes. In addition, spatial partitioning by nature lends itself to GPGPU programming where data can be processed in parallel by many threads. This thesis will present an implementation that makes this possible.

3.1 Overview

The goal of broad phase collision detection is to prune the total list of objects in the scene down to some potentially colliding set of objects. The potentially colliding set of objects is then handed off to either the mid phase collision step where the PCS is further pruned or the narrow phase collision step where more fine-grained collision detection is performed. To facilitate efficient broad phase collision checks, bounding volumes are used. These bounding volumes are the main input to this algorithm, and the main output is the potentially colliding set of objects.

In hierarchical grid spatial partitioning, a set of overlapping uniform grids is created where each grid has a different cell size. Objects or their bounding volumes are then assigned to cells within the various different grids that they overlap. Objects that share a cell are determined to be *close*. Objects that are *close* need to be examined further for potential collisions. By structuring this problem correctly, the data can be processed in a highly parallel fashion. As such, it becomes

amenable to processing using GPGPU techniques. By harnessing the GPU in this way, many potential collisions can be examined in parallel. The algorithm for doing this will be discussed in the next section.

3.2 Algorithm

The algorithm begins by receiving a set of bounding volumes. It assumes that throughout the simulation, any object that could be involved in a collision keeps track of its bounding volume. The choice of bounding volume for this system is the axis-aligned bounding box (AABB). AABBs were chosen for their simplicity. The main advantage here is that AABBs require very little memory. This makes them a good candidate for the GPGPU environment where memory transfers are expensive. The algorithms could be easily extended to handle other bounding volume types as well. These bounding volumes are the basis for the algorithm presented here. Given the bounding volumes as input, the algorithm proceeds through four main stages: initialization, sorting, collision cell list construction, and collision cell list traversal.

3.2.1 Initialization Stage

The initialization stage begins by constructing two arrays - an *object array* and a *cell ID array*. These two data structures are central to the algorithm. Each array contains $N * G * 2^d$ elements where N is the number of bounding volumes in the scene, G is the number of overlapping grids, and d is the dimensionality of the scene.[1] The object array is filled with indices that refer back to the list of bounding volumes. Elements in the cell ID array contain identifiers that indicate which cells a bounding volume overlaps. Along with the cell IDs, cell ID array elements indicate the level of the hierarchy with which a cell ID is associated. Any sorting operations that occur on either of the arrays must retain relative ordering with respect to the other array.

Cell IDs are calculated using a simple spatial hash of the centroid of each bounding volume.

[1] If the size of a cell is always larger than the largest object inserted into it, then a bounding volume may overlap at most 2^d cells.

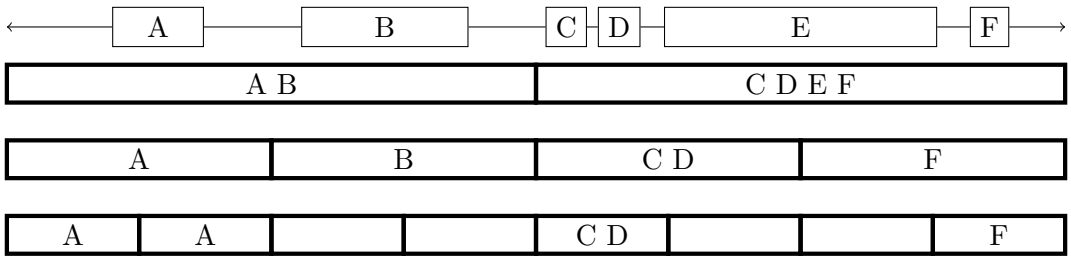

Figure 3.1: This figure demonstrates a one dimensional example of the implementation of hierarchical grids used in this thesis. The objects contained in the scene are represented on the top row. The hierarchical grid is represented by the bottom three rows. Each object is inserted into all cells that it overlaps on all levels in which the bounding volume fits.

The spatial hash of a particular point gives the approximate location in space where that point is located. Similarly to the hash described in [22], the spatial hash used here is computed by splitting up each axis uniformly from start to end and finding where the point in question falls within that partitioned space. This occurs for each x, y, and z component of the point. The three resulting values are then combined to compute a final hash. The size used to partition the space is related to cell size. When using this hashing method with hgrids, each hash must be tied to a particular level of the hierarchy since the cell size of each grid is different.

After allocating space for the two arrays, a CUDA kernel is launched with one thread per bounding volume. Each thread examines the size and location of the bounding volume assigned to it. A cell ID is generated for each cell in each grid level that the bounding volume overlaps. Since each bounding volume effectively has $G * 2^d$ elements associated with it, any elements that remain unused after initialization are assigned a value to indicate that they are invalid. Once all of the bounding volumes have been processed, the initialization stage is complete. Figure 3.1 illustrates how objects would be inserted into the hgrid discussed here.

3.2.2 Sorting Stage

After initialization is complete, the cell ID array is sorted. The sorting is done first with respect to the grid level and then with respect to the cell ID itself. This causes all cells from the

same level to be grouped together. As mentioned previously, the object and cell ID arrays must be sorted together. A key-value sort is used to achieve this. For maximum parallelism, the underlying algorithm is a parallel radix sort which has a theoretical run-time complexity of $O(N)$ where N is the number of fixed size keys [10, 15, 21].

3.2.3 Construction Stage

Once the arrays have been sorted, the collision cell list is constructed. The collision cell list is similar to the list presented in [15]. It is essentially the list of all cells that contain multiple bounding volumes. To calculate the collision cell list, a parallel sparse histogram is computed on the cell ID array. This is done by first finding all of the distinct cell ID/grid level combinations. At this point, any invalid cells are discarded. All of the distinct combinations are then counted.

Using this information, index offsets to each distinct element in the cell ID array can be computed. The count and index values are then paired together to form elements in the collision cell list. The final task of the construction stage is to initialize a matrix that is used to track collision checks between pairs of objects. The construction of the collision cell list in this way partitions the problem space and facilitates parallel data processing, which occurs in the traversal stage.

3.2.4 Traversal Stage

The collision cell list is processed during the traversal stage. A thread is launched for each element in the collision cell list. As such, that thread is assigned to a cell containing multiple potentially colliding bounding volumes. All of the bounding volumes in that cell are checked against each other for potential overlap. Any bounding volumes that are found to be overlapping are added to the PCS.

To prevent duplicate collision checks, a global list is kept that tracks each pair of bounding volumes. Before a collision check is performed between two bounding volumes, a look-up is performed to see if these bounding volumes have already been tested. If the pair of bounding volumes

has already been tested elsewhere (within a different cell in the same level or on a different level), the test is skipped.

3.3 Thrust

The back-end of the implementation for this thesis relied heavily on Thrust [7]. Thrust is a parallel algorithms library designed for presenting a high-level interface to GPGPU programming. It closely resembles the C++ Standard Template Library (*STL*) [23]. This facilitates enhanced developer productivity. Operations like sort, scan, and reduce can be quickly implemented. Listing 3.1 demonstrates how Thrust might be used to sort a vector of cell IDs.

Listing 3.1: Example Thrust Code

```
int main(void) {

  // generate 10000 random cell IDs on the host
  thrust::host_vector<int> hCellIds(10000);
  thrust::generate(hCellIds.begin(), hCellIds.end(), rand);

  // transfer data to device memory
  thrust::device_vector<int> dCellIds = hCellIds;

  // sort the cell IDs on the device
  thrust::sort(dCellIds.begin(), dCellIds.end());

  // copy the data back to the host
  thrust::copy(dCellIds.begin(), dCellIds.end(), hCellIds.begin());

  return 0;
}
```

Thrust is a powerful library that offers much functionality through a familiar interface. It abstracts away much of the heavy lifting that must be performed in GPGPU development. There is no need to specify the thread block or grid layout for CUDA. Most operations are performed on *STL*-like vectors of data. Memory transfers can be handled using standard vector operations. Many of the *STL* algorithms and containers are available for use. While Thrust greatly lowers the initial development costs, there is a certain amount of fine-grained control that is lost. Further, there is a certain amount of overhead that is assumed. Overall, it proved to be an effective way to implement the algorithm presented here.

3.4 Summary

A GPGPU implementation for hierarchical grid spatial partitioning has been presented. It consists of four main stages: initialization, sorting, collision cell list construction, and collision cell list traversal. The algorithm input is a set of bounding volumes (AABBs in this case). The output is the potentially colliding set of objects in the scene. The next section will analyze the performance of this system.

Chapter 4

Results

The goal of this thesis was to produce an effective real-time broad phase collision detection system. It has been demonstrated that uniform grid spatial partitioning can be used in a highly parallel way on the GPU to detect collisions in a broad phase [15]. As discussed previously, choosing an appropriate cell size can be difficult and presents issues in certain situations with uniform grids. In an attempt to alleviate these issues, the approach presented in this thesis used hierarchical grid spatial partitioning on the GPU to achieve real-time[1] broad phase collision detection. The results from testing this implementation will show that this hgrid system can be used to calculate collisions in real-time. However, in nearly all cases the overhead of using multiple grids offers no performance gains and, in many cases, even degrades performance.

To demonstrate the performance of the system, a variety of tests were performed. The test input parameters include the size of the scene, the number of objects in the scene, the object size disparity (the range in size from the smallest object to the largest object in the scene), and the number of grids included in the hierarchy. Each test starts by generating a scene using the specified parameters. Then, AABBs of a random size and location are generated. The size of the bounding volumes is constrained to be within the specified object size range. After the scene has been set-up, collision detection is performed against the objects in the scene. The performance metrics demonstrate the time spent in one step of the simulation. All of the tests were performed on a workstation with an Intel Core i7 processor at 2.5 GHz and 16 GB of available system memory.

[1] In this context, real-time is defined as 30 frames per second. As a result, any operations must complete in less than 33 ms to be considered real-time.

The workstation is also fitted with a NVIDIA GeForce GTX 580M with 2 GB of global memory and runs CUDA driver and run-time version 5.0.

As a basic demonstration of the system, a simple test was performed using a single grid in the hierarchy.[2] The number of objects included in the scene was varied to show how performance scales as the scene complexity grows. Figure 4.1 illustrates the results of this test. Even for as many as 30000 objects the algorithm can effectively detect broad phase collisions in real-time. For N objects, a brute force collision detection would require $N * (N-1)/2$ collision tests. For 30000 objects, that is nearly $5 * 10^8$ collision tests. By performing a parallel broad phase collision test using the algorithm presented above, the number of narrow phase or fine-grained collision checks that need to be performed can be drastically reduced in an efficient manner.

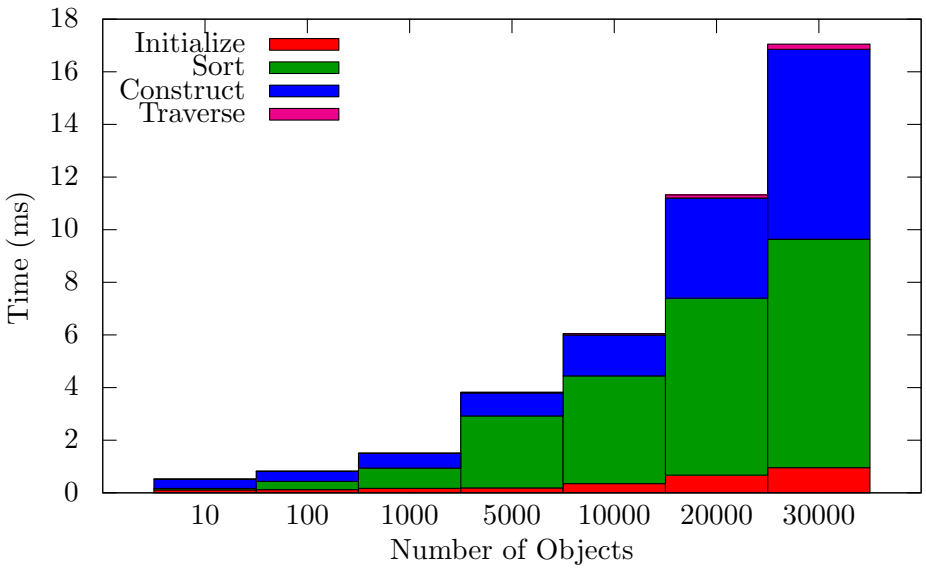

Figure 4.1: This figure primarily demonstrates the time spent in each stage of the collision detection algorithm and how the time spent in each stage scales with the number of objects. Only one grid was used in this test. The sort and construct stages are affected most heavily be scaling the number of objects.

Figure 4.1 also demonstrates the break down of each stage of the algorithm as described in

[2] Since uniform grids can be thought of as a special case of an *hgrid*, using one grid can effectively illustrate performance without loss of generality. The multiple grid scenario will be presented later.

section 3.2. As the figure shows, the sort and construct stages are highly affected by the number of objects in the scene. Since these stages act on the entire cell ID array, this follows logically. The traversal stage can also be highly affected by the number of objects in the scene. To be more precise, the time spent in the traversal stage is dependent on the number of potential collisions that must be checked. Object dispersal, grid size, and the range in object size will greatly affect the traversal stage.

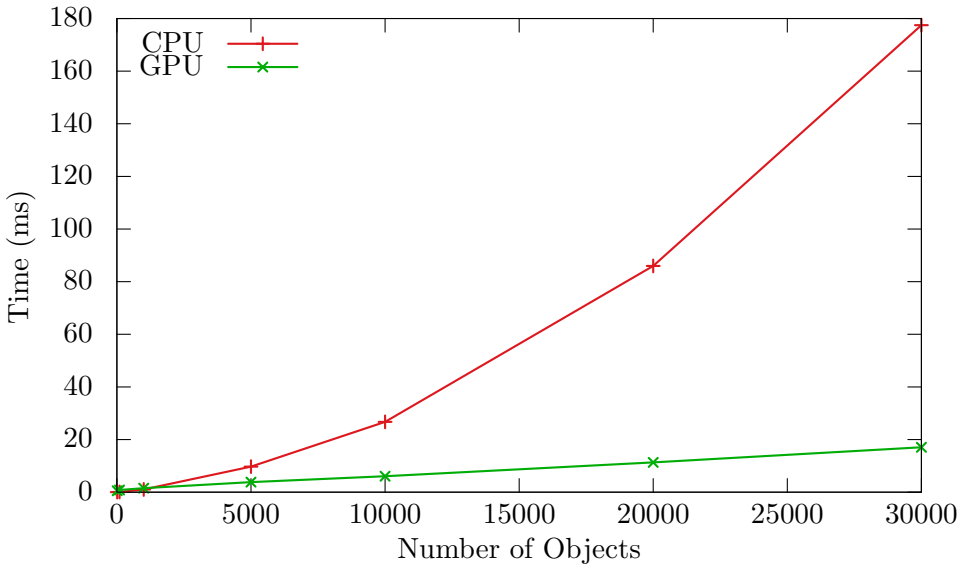

Figure 4.2: This figure illustrates a performance comparison between GPU and CPU implementation of the same collision detection algorithm. As the number of objects grows, time spent by the CPU implementation rises quickly while the GPU implementation remains relatively steady.

One of the main novelties of this thesis is that it runs on the GPU. However, since the underlying implementation uses Thrust, it is trivial to run the same algorithm on the CPU. Another test was performed to compare the CPU and GPU implementations. This test used the same parameters as the previous test. Figure 4.2 illustrates the performance boost that can be gained by using the GPU instead of the CPU. Harnessing the parallelism of the GPU often does not generally make a given problem solvable in $1/N$ time. Rather it allows a problem N times larger to be

solved in the same amount of time. This can be seen in figure 4.2. From 5000 objects onward, the CPU implementation performance starts rising rapidly while the GPU implementation performance remains relatively steady. Further, the CPU implementation breaks the real-time constraint near 10000 objects while the GPU implementation is still operating in real-time at as many as 30000 objects.

Hierarchical grids are designed to alleviate some of the issues that uniform grids have with cell size (see figure 1.2). By using multiple overlapping grids with different cell sizes, *hgrids* can attempt to handle widely varying object sizes. One case where uniform grids can perform poorly is when there is a large range of object sizes. In this scenario, uniform grids commonly choose a cell size that can contain the largest object in the scene. When this happens, each cell will contain many objects. Using an *hgrid* can help partition the objects into more reasonable cell sizes and thus reduce the number of objects contained in each cell in the collision cell list.

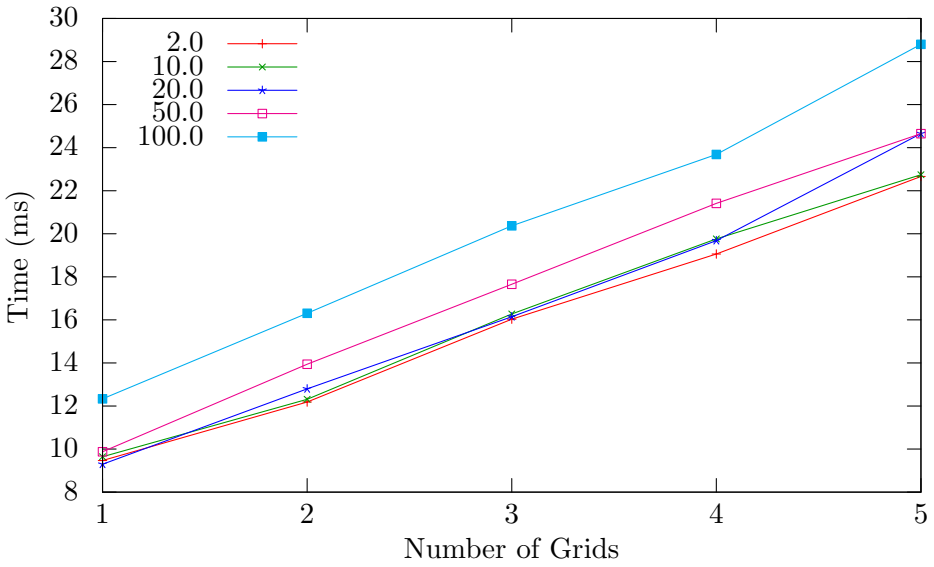

Figure 4.3: This figure represents the effect of the number of grids in the hierarchy and the size disparity of the objects on overall performance. Each line represents a scenario with different object size disparities (i.e. the ratio in size between the largest and smallest objects in the scene). Overall, performance scales similarly for the different disparities as the number of grids changes. The size of the scene and the number of objects were held constant.

To verify this hypothesis, a test was performed using a fixed size scene with 5000 objects. The number of grids used and the range in object size were varied. Figure 4.3 illustrates data collected from this test. Each line in the graph represents the ratio in size between the largest object in the scene and the smallest object in the scene. In all cases, performance scales linearly with the number of grids. Additionally, it appears that increasing the number of grids used in the hierarchy negatively impacts performance regardless of object size disparity.

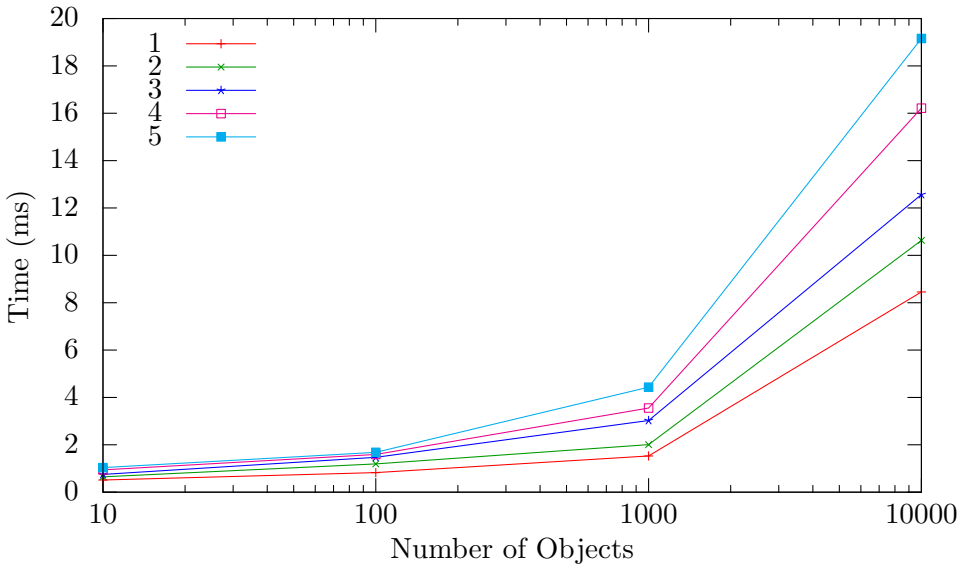

Figure 4.4: This figure illustrates how performance is affected by the number of objects in the scene and the number of grids used. The object size disparity was held at a constant level of 100. The different lines represent the different number of grids used. In scenes with many objects, a large number of grids can prove to be expensive.

Another test was conducted to examine the performance effect of varying the number of objects in the scene and the number of grids used. For this test, the disparity in object size was held constant at 100 (i.e. the largest object in the scene was 100 times larger than the smallest object in the scene). Figure 4.4 demonstrates the results of this test. Performance scales similarly for each of the hierarchy sizes. Regardless of the number of objects in the scene, using a larger hierarchy causes an increase in the run-time of the algorithm.

The results demonstrated in figures 4.3 and 4.4 contradict what one might expect to see in scenes with high object size disparity. To better understand the apparent poor performance when using multiple grids, figure 4.5 more closely examines the data from the previous test. It demonstrates the amount of time spent in each stage of the algorithm. In this figure, the object size disparity is 100 and the number of objects in the scene is 10000. These results are discussed in-depth in the next section.

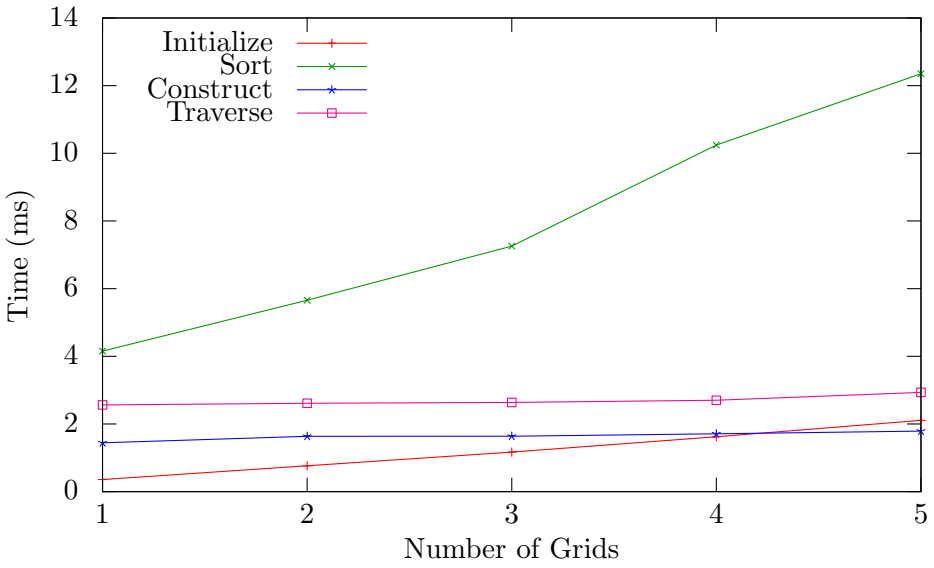

Figure 4.5: This figure illustrates the amount of time spent in each stage of the algorithm. In this case, the object size disparity and the number of objects were held constant at 100 and 10000 respectively while the number of grids was varied.

4.1 Discussion

Overall, the hierarchical grid spatial partitioning collision detection system presented here can successfully compute a potentially colliding set in real-time. This is demonstrated in 4.1 where 30000 objects are passed through broad phase collision detection in less than 20 ms. Further, figure 4.2 illustrates the benefits of performing the computation on the GPU. With respect to the number of objects in the scene, the GPU implementation scales much better than the CPU implementation.

One of the main contributions here is the hierarchical grid component of spatial partitioning. The two test results discussed above were performed using only one grid in the hierarchy. This effectively makes it a uniform grid system. In theory, the performance of uniform grid systems should be negatively impacted when the object size disparity is large. In the implementation presented here, the cell size will be equal to or greater than the size of the largest object in the scene when only one grid is used. This could lead to many objects residing in the same cell. Since a thread is launched for each collision cell, the amount of work required by each thread would be large. The expectation then would be that the traversal stage of the algorithm would take much time. By using multiple grids with different cell sizes, the number of objects in each collision cell could be reduced. However, figures 4.3 and 4.4 clearly show that using more grids negatively impacts performance regardless of the number of objects in the scene and the object size disparity.

Figure 4.5 provides some insight into why multiple grids fail to perform better than the single grid scenario. This figure shows the amount of time that is spent in each stage of the algorithm using a scene with 10000 objects and an object size disparity of 100. The data in figure 4.5 show that as the number of grids increases the amount of time spent in the initialization and sorting stages increases while the amount of time spent in the construction and traversal stages remains relatively constant. Since the initialization stage simply steps through the object and cell ID arrays, which are of size $N * G * 8$, its run-time complexity is $O(N * G)$ where N is the number of objects and G is the number of grids. The sort stage, which uses a parallel radix sort, also has a run-time complexity of $O(N * G)$ [10, 15, 21]. These run-time complexities match what the results show in terms of the performance scaling in the initialization and sorting stages.

To further understand performance, the algorithm was profiled using NVIDIA Visual Profiler [4]. This revealed a number of performance bottlenecks related to global memory usage, low computing occupancy, and branch divergence. Appendix A presents an overview of the CUDA computing architecture and what these issues mean. The implementation presented here stores the object and cell ID arrays in global memory. As such, each stage of the algorithm suffers to some degree from high global memory access latency. The sorting stage in particular suffers from

global memory efficiency problems. The traversal stage was found to suffer from global memory access and branch divergence issues. During the traversal stage, bounding volume look-ups are performed when testing for collisions. Due to the sorting of the object array with the cell ID array, these look-ups result in random access, which results in poor caching. Since the collision cell lists assigned to each thread during the traversal stage vary greatly depending on the scene, divergence is highly probable, which results in sequential thread execution within each warp.

As discussed in section 3, all bounding volumes are inserted into all cells into which they overlap and fit, which is illustrated in figure 3.1. As a consequence, the grid with the largest cell size in the multiple grid scenario will appear exactly the same as the sole grid in the single grid scenario. The global list of collision checks mentioned in section 3.2.4 is intended to prevent collision checks on any pair of objects for being performed multiple times. The goal is that if two small objects are processed during the traversal stage by a thread assigned to a cell in a grid with a smaller cell size, then they will not need to be processed by a thread assigned to any other cell containing those objects. This presents an issue with thread scheduling. If all of the cells in the grid with the largest cell size are processed first, then the traversal stage of the multiple grid case would perform no better than the single grid case.

Despite these performance issues, the algorithm presented here successfully performs broad phase collision detection in real-time with complex scenes. By addressing some of these issues, performance would likely be improved. Further, the multiple grid case could prove to be a more viable option.

Chapter 5

Conclusion

This thesis demonstrates a GPGPU implementation of broad phase collision detection using hierarchical grid spatial partitioning. Collision detection in general is one stage of the process of collision handling, which involves collision detection, collision determination, and collision response. Collision detection finds the set of objects in a scene that are colliding. Collision determination follows collision detection and identifies the actual points of intersection between two colliding objects. Collision response resolves the collision into a physical reaction. Broad phase collision detection is used as a preprocessing step to prune the number of fine-grained collision calculations that must be performed on the actual object geometry. To do so, the broad phase uses bounding volumes, which are an approximation of each object's geometry.

The method of broad phase collision detection presented here uses hierarchical grid spatial partitioning. This approach subdivides the scene into a set of overlapping grids where each grid contains a different cell size. The main advantage of using a set of overlapping grids is that the different grid sizes facilitate scenes with widely varying object sizes. The issue of widely varying object sizes can be a problem in uniform grid spatial partitioning where only one cell size is used. Hierarchical grids are an extension of uniform grids that were developed largely out of a need to resolve this problem.

The implementation of hierarchical grids presented here works simply by using a spatial hash to assign each bounding volume to each cell in each grid that it overlaps. Then, any cell that is found to contain multiple bounding volumes must examined. The bounding volumes in that

cell are checked against each other for overlap. Any overlapping objects are then added to the *potentially colliding set* or *PCS*. After all cells have been examined, the PCS can be processed by a narrow phase collision detection algorithm and subsequently processed for collision determination and response as needed.

The results show that this is an effective broad phase collision detection system. It can process as many as 30000 objects in real-time using modern hardware. The various stages of the algorithm were examined (figure 4.1) to demonstrate performance scaling. The sorting and construction stages are most highly affected. The performance of the traversal stage is dependent on the object dispersal, grid size, and range of object sizes. Figure 4.2 demonstrates that the GPU implementation does provide better scaling relative to a CPU implementation. Figures 4.3 and 4.4 examine the effectiveness of using multiple grids in this implementation. It appears that even for scenes with large object size disparity using multiple grids adds too much overhead to be effective. Figure 4.5 reveals that the overhead lies in the initialization and sorting stages, which scale poorly with the number of grids used in the hierarchy.

5.1 Future Work

While the broad phase collision detection system presented here does function well in pruning the number of objects that must be processed in the narrow phase, it fails to alleviate issues related to object size disparity that are intrinsic to uniform grids. Further investigation into this matter reveals that the main reason for no performance gains in the multiple grid scenario is caused by poor scalability of the initialization and sorting stages. The run-time of these stages is directly dependent on the size of the object and cell ID arrays. Storing the bounding volumes in a different way or performing some preprocessing steps to reduce the size of the arrays could potentially yield better scaling.

Section 4.1 discussed some of the bottlenecks found in this implementation. Some of these problems are the same as those found by Pabst, et al. [20] in the implementation presented by Le Grand [15]. To solve the divergence problem, Pabst, et al. suggests launching a thread per

potential collision rather than launching a thread per collision cell. Also, they suggest performing a preprocessing step that organizes objects into a shared memory data structure to alleviate the global memory access problem. Both of these techniques would likely yield performance gains.

The implementation presented here is currently restricted to using AABBs for bounding volumes. AABBs were chosen for their simplicity and low storage requirements. However, it is possible that the storage overhead of tighter fitting bounding volumes like an OBBs or a k-DOPs might be minimal and help reduce the number of objects placed in the PCS. Extending the implementation to use other types of bounding volumes could be a valuable addition.

Another future goal of this work is to integrate this system into a full-featured physics engine. The first step would be to couple this broad phase step with a narrow phase collision detection algorithm. This would provide the full collision detection step, which could then be expanded by adding collision determination and collision response.

Bibliography

[1] CUDA toolkit documentation - CUDA C programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[2] CUDA toolkit documentation - thrust. http://docs.nvidia.com/cuda/thrust/.

[3] NVIDIA CUDA. http://www.nvidia.com/object/cuda_home_new.html.

[4] NVIDIA visual profiler. https://developer.nvidia.com/nvidia-visual-profiler.

[5] SGI Sort. http://www.sgi.com/tech/stl/sort.html.

[6] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. Real-Time Rendering 3rd Edition. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[7] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for CUDA. GPU Computing Gems: Jade Edition, pages 359–372, 2011.

[8] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. Communications of the ACM, 18(9):509–517, 1975.

[9] Gino van den Bergen. Efficient collision detection of complex deformable models using AABB trees. Journal of Graphics Tools, 2(4):1–13, 1997.

[10] Guy E Blelloch. Prefix sums and their applications. 1990.

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

[12] Elmer G Gilbert, Daniel W Johnson, and S Sathiya Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. Robotics and Automation, IEEE Journal of, 4(2):193–203, 1988.

[13] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A hierarchical structure for rapid interference detection, 1996.

[14] Naga K Govindaraju, Stephane Redon, Ming C Lin, and Dinesh Manocha. CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, pages 25–32. Eurographics Association, 2003.

[15] Scott Le Grand. Broad-phase collision detection with CUDA. In Hubert Nguyen, editor, GPU Gems 3. Addison-Wesley Professional, first edition, 2007.

[16] Ming C Lin and John F Canny. A fast algorithm for incremental distance calculation. In Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on, pages 1008–1014. IEEE, 1991.

[17] Lin Loi. Fast GPU-based collision detection. Master's thesis, Chalmers University of Technology, 2010.

[18] Brian Mirtich. Impulse-based Dynamic Simulation of Rigid Body Systems. PhD thesis, University of California, Berkeley, December 1996.

[19] Brian Mirtich. Efficient algorithms for two-phase collision detection. Practical motion planning in robotics: current approaches and future directions, pages 203–223, 1997.

[20] Simon Pabst, Artur Koch, and Wolfgang Straßer. Fast and scalable CPU/GPU collision detection for rigid and deformable surfaces. Computer Graphics Forum, 29(5):1605–1612, 2010.

[21] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, pages 1–10. IEEE, 2009.

[22] Florian Schornbaum. Hierarchical hash grids for coarse collision detection. Student Thesis, University of Erlangen-Nuremberg, 2009.

[23] Alexander Stepanov and Meng Lee. The standard template library. Hewlett-Packard Laboratories, Technical Publications Department, 1995.

[24] Thatcher Ulrich. Loose Octrees, volume 1 of Charles River Media Graphics, pages 444–453. Charles River Media, 2000.

[25] Gabriel Zachmann. The boxtree: Exact and fast collision detection of arbitrary polyhedra. In First Workshop on Simulation and Interaction in Virtual Environments (SIVE 95), 1995.

Appendix A

CUDA Computing Architecture

To better understand GPGPU computing using CUDA, it is necessary to understand the CUDA computing architecture. An in-depth discussion of the CUDA computing architecture is discussed in [1] and summarized here.

GPGPU computing is achieved in CUDA by writing *kernels*. A kernel is a function that gets executed N times in parallel by N different CUDA *threads*. The CUDA thread layout must be specified when the kernel is launched. In the CUDA thread hierarchy, threads are grouped in *blocks* which are then organized into *grids*. The basic unit for resource scheduling is the thread block. *Streaming multiprocessors* contain multiple processing cores and can execute thread blocks concurrently. Further, threads within a thread block execute concurrently. To handle the large number of concurrently executing threads, multiprocessors use *Single-Instruction, Multiple-Thread* or *SIMT* execution.

The unit of execution within the SIMT architecture on a CUDA multiprocessor is a *warp* which is a group of 32 threads. Each thread within a warp executes concurrently. Each warp executes one common instruction at a time. If the various threads within a warp disagree on the next instruction to execute, computation diverges and the warp must execute each branch in sequence. When divergence occurs, execution of other threads within the warp must be paused while the current branch is executed. Divergence occurs when the warp must execute a data-dependent conditional statement. After all possible branches have been executed, the warp attempts to converge and execute instructions in parallel once again. Full warp efficiency is achieved when

all threads within the warp agree on the instructions to be executed. There is a maximum limit to the number of blocks and warps that can reside on a multiprocessor at a given time. When the number of actively executing warps is equal to the maximum number of active warps allowed, the system is said to be running at *full occupancy*. Within that maximum limit, the number of blocks and warps assigned to a multiprocessor is dependent on the available memory on the multiprocessor and the memory required by each thread block.

The CUDA computing architecture contains three basic memory spaces: local, shared, and global. Each thread has a small amount of memory available to it. This local memory is available for the lifetime of the thread. Shared memory is available to all threads within a thread block and remains active for the lifetime of the thread block. Finally, the global memory space is available to all threads. This memory space is managed at the application level and is available to the host. Global memory is the most abundant but has high access latency.