Broadphase Collision Detection on the Cell Processor

by

Gavin Campbell,

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

September 2010

Declaration

Permission to Lend and/or Copy

Acknowledgments

I would like to thank my family for all their support throughout the year. I would also like to thank Michael Manzke and John O'Kane for their helpful advice on this project.

Gavin Campbell

University of Dublin, Trinity College
September 2010

Broadphase Collision Detection on the Cell Processor

Gavin Campbell
University of Dublin, Trinity College, 2010

Supervisor: Michael Manzke

Fundamental to computer games, physics simulations, molecular modelling and also robot motion planning, collision detection is considered to be a generally well established field. Many different solutions exist already, some more suitable to particular applications than others. However, in recent years, research into collision detection techniques has been re-invigorated by the onset of the parallel opportunities that came with the emergence of multi-core architectures and possibilities for general purpose processing on GPUs. The demand for more efficient techniques is evident as the importance of physical simulation in computer games has risen dramatically. It is the aim of this project to investigate and evaluate the applicability of broadphase collision detection, to the unique architecture of the Cell processor. The broadphase algorithm to be implemented is a variation of the popular "Sweep and Prune", also known as

“Sort and Sweep”, first documented by David Baraff and also implemented in the I-Collide system. The technique is supported by most current physics middleware solutions including the open source physics library, “Bullet”. This work will describe the development of a unique parallel sweep and prune algorithm designed to harness the power of the Cell. The development of an x86 based implementation will also be described, to provide a basis for comparison with the Cell implementation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Collision detection is the process of determining if objects are colliding or intersecting. It is one of the most important processes involved in creating believable, interactive virtual environments and is fundamental to a wide array of computing fields. Computer games, physics based simulations (including computer animation), molecular modelling and also robot motion planning, all rely heavily on a working collision detection system, to allow them to achieve their higher level goals. It is becoming increasingly important in the field of computer games and animation, as realistic physics simulation is often expected. The performance requirements of collision detection algorithms varies from field to field.

The Cell processor or “Cell Broadband Engine” is the result of an ambitious project between Sony, Toshiba and IBM, who formed the “STI” alliance with the intention of developing a groundbreaking processor architecture. What is unique about the processor, is its heterogeneous cores. It consists of one PowerPC Processor Element, known as the PPE, and eight Synergistic Processor Elements (SPE). It is the primary job of the PPE to initiate and manage the SPEs which are 128-bit vector processing units, designed for Single Instruction Multiple Data (SIMD) processing. To get the most out of the architecture, programmers must really take advantage of its parallel nature and SIMD processing capabilities.

1.1 Motivation

Despite the importance of collision detection and the number of algorithms and techniques that it has spawned, it will always remain a computationally expensive task. Any expensive calculations will always have an impact on interactive applications, such as games. This is because there are many other tasks that require processor resources, such as rendering, animation, artificial intelligence and audio. All of this must be carried out at least thirty times a second to generate a smooth interactive world. Some games even strive for 60 frames per second, which means even less processing time is allocated to the aforementioned tasks. For this reason, even if an algorithm performs admirably, there is always room for improvement when it comes to speed.

The sweep and prune algorithm is suited to generally static scenes, where only a small number of objects are moving. It does this by maintaining nearly sorted lists, which will be explained in more detail in chapter 2. However, it does suffer from a very time consuming initial sort all of the objects in the scene. The speed of this initial sort is a primary motivation for this project.

1.2 Goals

This aim of this dissertation is to investigate the applicability of the sweep and prune broadphase algorithm to the Cell processor. The primary goals are:

- To investigate and explain the background of collision detection and broadphase collision detection in particular, to discuss related work and provide an overview of the Cell architecture.
- To develop a simple rigid body based physics simulation environment that will provide the target for the broadphase algorithm. This implementation will be the same for both x86 and Cell.
- To develop a standard broadphase algorithm for an x86 processor.
- To identify opportunities for parallelism in the broadphase algorithm.
- To implement a parallelised broadphase algorithm that exhibits SIMD processing to exploit the capabilities of the Cell processor.

- To compare and contrast the two implementations.

1.3 Dissertation Layout

The State of the Art chapter will investigate the most pertinent techniques of collision detection and will present a detailed overview of the Cell processor. An overview will be provided of the area of collision detection also. It will present a more detailed assessment of broadphase collision detection in particular, examining some of the more seminal papers and some recent papers in the field. The merits and trade-offs of different broadphase algorithms will be discussed.

The Design chapter will deal with the reasoning behind important design decisions. It will also detail the design for rigid body based physics simulation, which is basically the testbed for the broadphase algorithm. The plan for the project implementation will also be presented.

The purpose of the Implementation chapter is to describe the development of both the final x86 and Cell implementations. The chapter will also describe some issues that arose during the implementation. Nuances with the algorithm implementation and issues encountered specific to Cell programming will be examined, as well as the methods by which these issues were overcome.

The Evaluation chapter is concerned with providing an accurate analysis of the project. The different scenarios under which both implementations of the algorithm will be assessed, will be detailed. The results will compared and contrasted, highlighting the possible strengths and weaknesses of each implementation of the algorithm.

The final chapter will draw conclusions based on the analysis of the project performed in the preceding chapter. A discussion of the merits of the project will be provided. Lastly, ideas for future work and any improvements that could be made to the system, will be put forward.

Chapter 2

Background and State of the Art

2.1 Collision Detection Overview

Collision detection is a vast topic, centred around the primary notion of detecting if two (or possibly more) objects are intersecting. Delving deeper into the problem, some more important questions that must be addressed are not just if objects have collided, but how have they collided, and when? Often the answers to these questions can have a dramatic impact on the the next stage in the collision pipeline. The topic that follows on directly from collision detection, is collision response. Collision response however, is an entirely different topic in itself and will not be discussed as it is not the concern of this project. Still, it is the duty of the collision detection algorithms to gather the relevant information about a collision.

In virtual worlds, such as in computer games, collision detection plays an essential role in creating and sustaining the illusion of a solid world. Without collision detection, objects would pass through each other and game characters would fall through floors, walk through walls and not be able to interact with the game world. It is also utilised as part of the artificial intelligence of game entities, to help them make decisions. An example of this could be an enemy soldier that starts shooting at the player only when he is visible to the enemy. Line-of-sight queries from the collision detection system would be used to check if the player is obscured by structures and obstacles in the game world.

Tying in to the world of computer games, computer animation has become more advanced, using expensive physics calculations to re-create realistic behaviours of materials, clothing, hair and even limbs. Cloth simulation is very popular in current game animation, for mimicking the behaviour of clothes. Effects such as clothing [1], [2] and hair [3], would require collision detection with the body of the character.

Collision detection is prominent in the world of robotics, used for motion planning and obstacle avoidance [4]. Other applications include the study of molecular interaction, education and training applications such as virtual surgery simulators [5] and also crash testing and other engineering simulations.

The computation cost of collision detection varies with scene complexity and the performance requirements differ with each application. Some applications such as computer games require real-time performance of their collision systems, others such as offline animation rendering and motion planning do not. In computer games, collision detection can account for a large amount of the time taken to render a frame. A well designed collision detection system can be vital to the performance and frame rate of a game. For scenes containing n objects, a brute force collision test between each pair of objects results in a complexity of $O(n^2)$. This may be acceptable for a relatively simple scene, with just a couple of objects. However, start introducing hundreds and thousands of objects into a scene and as you can imagine, this brute force approach is just not feasible as it is far too computationally expensive.

To overcome this issue, techniques have been developed that break up the problem of collision detection into two phases. These phases are known as the broadphase and narrowphase. Their primary goal is to reduce the computational cost of calculating collisions between large amounts of objects in a scene. The broadphase is concerned with quickly and efficiently calculating pairs of objects that could potentially be colliding, usually based on an approximation of their positions in 3D space. These potentially colliding pairs are then passed to the narrowphase algorithm for further processing. This narrowphase should be capable of determining exactly if two objects are colliding or not. An important aspect of the broadphase algorithm's job, is therefore to reduce the workload for the more computationally expensive narrowphase. Much of the information presented in this chapter and its structure is based on the excellent book "Real-Time Collision Detection" by Christer Ericson [6], which is a staple of any collision algorithm programmer's library.

2.2 Collision Algorithm Design Issues

When it comes to choices for designing a collision detection system, many different factors must be taken into consideration. How will virtual objects and the scene be represented from a geometrical standpoint? What type of queries is the system expected to handle? Is the system expected to run in real-time? These are the issues that will be expanded upon, in this section.

2.2.1 Application Environment Representation

This section will discuss how objects can be represented and why it is useful for simplified geometry to be substituted for modelling geometry, when it comes to collision detection.

The triangle is the de facto standard when it comes to rendering primitives. As a result, a polygonal representation of objects and scene geometry is favoured, as polygons can be broken up into many smaller triangles. *Polygon Soup* is a term used to describe a group polygons that are not ordered in any particular manner. In a polygon soup, it is no trivial task to determine if one polygon lies inside the bounds of another, as there is no information pertaining to the "inside" of a polygon. Additional information is required about the *features* of a polygon. A feature of a polygon is either a vertex, an edge or a face. We need to know which features are connected to each other, to determine the convexity or concavity of an object. A larger polygonal surface called a *mesh* can be constructed by connecting smaller polygons together by their edges. From these polygonal meshes, larger objects are built. There are different types of polygonal object representation. The preceding method is known as *explicit* representation, where objects are defined in terms of their features. The other type of representation is *implicit*. This refers to other geometric primitives that are defined by a mathematical equation, rather than by their explicit features. Examples of implicit geometric objects would be spheres, cylinders and tori. The implicit description of an object can allow for quick intersection tests and is often used to approximate explicitly defined objects in a scene, enabling them to be efficiently processed by a collision detection system.

Though possible to perform the collision detection directly on the polygonal meshes of models and scene objects, it is a far better idea to pass separate geometry to the collision testing system. Modern day applications exhibit such graphical prowess, that the associated rendering geometry has become almost too complex for collision detection and physics systems to handle efficiently. Therefore it is common practice to use far simpler geometric objects as approximations for the collision detection. This less complicated *proxy geometry* frequently consists of simple boxes and spheres to represent the more complex objects in a scene. If the proxy object collides, then it is safe to assume that the complex objects being represented are colliding as well, or are at least in very close proximity to one another. These simplistic shapes are more commonly known as *bounding volumes* and will be described in more detail in section 2.3. There are some drawbacks to replacing rendering geometry with separate collision geometry, that include the extra work required to produce and maintain two sets of similar geometry. However, it is generally considered to be worth the effort as the positives outweigh the negatives.

2.2.2 Types of Queries

Different types of collision queries can be handled by a collision detection system, depending on the requirements of the system. The most fundamental and straightforward query would be the problem of *intersection testing*. An intersection test returns an answer in terms of a simple true or false, as to whether two objects are overlapping, given their current positions and orientations. These boolean queries are commonplace as they are fast to calculate and are easily implementable. A slightly more difficult type of query to implement is *intersection finding*. It is often necessary to find the exact parts of the objects that are intersecting. The result is no longer a simple boolean expression, as multiple contact points can be found. In rigid body simulations, these multiple contact points or *contact manifold* may need to be ascertained. In other situations, a single point in space, common to the colliding objects may suffice as a single point of contact.

Computation of the contact manifold is a difficult problem, which inspired the use of *approximate queries*, which are easier to compute. Approximate queries simply impose a limit to which the contact manifold is computed. These types of queries are commonly used in games as they are easier to deal with than exact queries and any loss in accuracy would not be remarkably perceptible to the naked eye. Most applications would also require the determination of the *penetration depth* of two colliding objects. The penetration depth is defined as the minimum translation vector required to translate the objects, so that they become separated. *Closest points* calculation between two objects, is another advanced query required by some narrowphase algorithms, such as GJK [7].

2.2.3 Simulation Parameters

The parameters of a simulation must be taken into account before making any important decisions, when it comes to the design of a collision detection system. Parameters may include the amount of objects that the simulation is expected to handle, whether collisions are handled in a sequential or simultaneous manner and if motion is considered to be discrete or continuous.

Probably the most important parameter that concerns this research and broadphase collision detection, is the number of objects present in a simulation. As mentioned previously, in the introduction to this entire section 2.1, a simulation with n objects requires $O(n^2)$ pairwise tests, due to the fact that any one object can potentially collide with any other object. If one were to focus on trimming the cost of performing the pairwise tests, runtime would only be affected linearly. What is needed is a method of reducing the amount of pairwise tests to be carried out, without affecting the integrity of the system.

The reduction as explained briefly in section 2.1, is performed by using a two-phase *hybrid* approach to handling the collisions, the phases being the *broadphase* and the *narrowphase*. The broadphase identifies groups of objects that have the potential to collide and the narrowphase performs the exact collision detection on these sub-groups. The broadphase is sometimes referred to as *n-body processing* and the narrowphase as *pair processing*. Broadphase algorithms will be discussed in detail in section 2.4. As well as the number of objects, the size of the objects and how they differ can affect how many pairs must be processed. This makes sense as, if you think about a group of boxes in a confined space, there will be far more collisions between them if they all have large dimensions, where as there may be no two objects colliding if their dimensions are sufficiently small.

Another parameter to be considered is whether sequential or simultaneous motion is catered for by the system. *Simultaneous motion*, is what we observe every day in real life. This is where all objects are moving at the same time. In terms of a simulation, this type of motion would be akin to the movement of all objects taking place during the same time step. Any collisions are handled during this time step. For this type of motion to be accurately re-created by a computer simulation, the time of earliest contact is required. The simulation would have to be moved to this point in time, as well as moving every other object to its position and orientation at this point in time. This process would have to be repeated for every collision, which becomes quite expensive. There are alternatives to handling simultaneous motion, such as rolling back time to the point of collision, but they all remain quite an expensive process. It is often considered overkill for some applications to have such a high degree of accuracy, when it comes to simulated motion. For these applications, such as some games, the option of *sequential motion* might be a more attractive proposition. This sequential motion involves moving objects, one at a time and collisions are handled before the next object is processed. Sequential movement can lead to collisions that would not occur with simultaneous movement. This can occur sequentially because one object is moved then the second object detects a collision with it, however if both are moved simultaneously there may be no collision at all.

Similar to sequential versus simultaneous motion is the notion of *discrete* versus *continuous* motion. While the former deals with the motion that lead objects to possible collision at a point in time, the latter concerns whether or not the objects are actually in motion at the time of testing. Choosing discrete or continuous motion can have a significant impact on both the processing effort required to generate the result and the efficacy of the system. Discrete versus continous is really *static* versus *dynamic* collision detection. Static collision detection is testing objects against each other at different points in time to see if they intersect. At each point in time during this process, each object is treated as a stationary object. Dynamic detection differs by considering the full continuous motion of an object over a given time period. This is achieved via a *swept volume*. A swept volume is basically the concatenation of an object's volume at each point in time, along a set time interval, e.g., the volume of a cube at position $timestep = 1+$ the volume of a cube at position $timestep = 2 + ...$ etc. If there is no intersection between two swept volumes then an intersection has not occurred and will not occur in the near future. Even if the swept volumes do intersect, it does not necessarily mean the two objects will collide when in motion. Dynamic collision tests can be difficult to compute and to deal with in general, but can usually report the exact time of collision and the point of first contact. Static collision test are far cheaper computationally but must be performed often enough (with a sufficiently small time step) so that collisions are not missed altogether. This is known as *tunnelling* and can occur if the time steps between collision test are so large that two objects have passed through each other during the gap in the time steps. It is called tunnelling, as when running at thirty or sixty frames per second, it appears as though the objects have passed through each other without any consequence. A good solution is to subtract the motion of one object from another, leaving the second relatively static. Sometimes instead of a swept volume, an approximate simplified object would be used, such as a *speedbox*, which is basically and elongated box, that accounts for an object's full range of motion.

2.2.4 Performance

Consider again computer games, as they tend to be more demanding in terms of collision detection performance requirements. For the best visuals and smoothness, all games would ideally like to run at 60 frames per second (fps). This means that only 16.7 milliseconds (ms) are allocated to calculate each frame. Different types of applications have different requirements, but some games could allocate up to 30% of frame creation to collision detection, giving us around 5 ms for 60 fps or 10 ms for 30 fps. This is not a lot of time, if you consider that a scene in a game could contain hundreds of objects all requiring collision handling. Any possible reductions in processing time, must be sought after. Collision algorithms sometimes vary in their processing cost from a frame to frame basis. Care must be taken to make sure that there is no significant drop in frame rate at any stage, as this can be visually jarring and disorientating to the user. The worst case scenario for an algorithm should not take excessively longer than the average case, to ensure that the frame rate remains constant. A slower consistent frame rate (such as 30 fps) is often favourable to a varying frame rate that can occasionally reach 60 fps.

2.3 Bounding Volumes

Most broadphase collision algorithms require objects to be enclosed in some sort of simple proxy object, to allow them to work efficiently. These proxy objects are commonly known as *bounding volumes.* An object can be enclosed by many different geometric shapes, but not all make for sufficient bounding volumes. There are a set of favourable qualities that all bounding volumes should strive for. Objects should be bound as tightly as possible, eliminating as much empty space between the bounding volume and the object itself. The intersection tests between two bounding volumes should not be costly to compute. There is a trade-off between tightness and intersection testing as the tightest-fitting volumes are usually more complex geometrical shapes, but efficient intersection testing requires the geometry to be a simple as possible. The other important characteristics of a good bounding volume is that it should be inexpensive to calculate, transformations of the volume such as rotations should also be processor friendly and it should use as little memory as possible to store the structure. In this section we will explore some of the more popular bounding volumes used today.

2.3.1 Spheres

A sphere is one of the simplest bounding volumes. It is very popular, and with good reason. The cost of an intersection test between two spheres is very low. It is simply a matter of calculating the euclidean distance (or distance in 3D space) between the two spheres and comparing it to the sum of the radii of the spheres. If the distance is less than the sum of the radii, then the spheres must be intersecting. Another favourable feature of sphere bounding volumes is that they are rotationally invariant, meaning that the orientation of the sphere and that of the enclosed object make no difference. The benefit of this is that updating the sphere as the enclosed object moves, is computationally inexpensive. The update basically consists of a simple translation, which is preferable to other update calculations that we'll encounter later in this section. With only four components to its structure (x,y,z coordinates and the radius size) the bounding sphere has the smallest memory footprint of all bounding volumes. One drawback is that it may not enclose an object as tightly as one might wish and the calculation of an optimal bounding sphere is more difficult to compute, than say that of an axis aligned bounding box.

2.3.2 Axis Aligned Bounding Boxes

Along with bounding spheres, the *axis aligned bounding box (AABB)* is one of most common type of bounding volume. An example of its usage is provided by Baraff [8]. Basically a six sided rectangular box (or a cuboid), its faces are always aligned to the corresponding axes, such that the normals of each face are parallel to the axes in question. A simple way to visualise an AABB, is to imagine one of the bottom left corners of the box (either the nearest face or furthest face) being positioned at the origin of an *x,y,z coordinate system.* The three lines emanating from this position, will be aligned exactly to the x, y and z planes respectively. This box can move anywhere in 3D space, but its orientation will never change. AABBs can be *fixed* or *dynamic.* Fixed boxes must be large enough to house an object and accommodate for all possible orientations, e.g., an oblong object will require different bounding box dimensions depending on its rotation. Therefore, fixed AABBs result in simpler update procedures as they don't need to be recalculated, but the downside is that they will probably not fit the object very tightly. The alternative to fixed AABBs is to dynamically recalculate the bounding box as the object moves and rotates. As you can imagine, dynamically generated AABBs are more computationally intensive, but result in a more ideal, tightly-fitting bounding volume.

The most attractive feature of AABBs is the speed of their intersection test. The test is very quick because it is just a matter of comparing the coordinate values of the boxes. The test consists of checking to see if there is an overlap between the boxes separately on each axis, for instance, along the x-axis, check if the minimum extent of box A less than the minimum extent of box B and is the maximum extent of box A greater than the minimum extent of box B. An intersection exists only if this comparison returns true for each axis. An AABB can be expressed in three different ways. The first way is to store the minimum and maximum coordinates along each axis, which will require two data structures capable of holding three values each. The second representation is to keep track of the minimum corner again, but just store the diameter or width extents from this corner. The third approach is to store the centre point of the box and the radii or half-widths that extend from it, much like a sphere. The latter two approaches are considered to be more efficient as the diameters and radii can be stored in fewer bits, the first minimum and maximum approach, requires all values to be stored to the same precision. For simple translation update operations, the second two approaches are again favourable as they only require three values to be updated. The minimum and maximum point approach can be useful for situations where the minimum and maximum AABB information is required by an algorithm, such as sweep and prune. It is also relatively simple to calculate for applications that desire dynamically computed bounding boxes.

2.3.3 Oriented Bounding Boxes

An *oriented bounding box (OBB)* is a rectangular shape, essentially the same as an AABB except rotated arbitrarily. Implemented by Eberly in [9]. The advantage of OBBs over AABBs is that they can enclose an object more tightly, as they are oriented to the lie of the object. The disadvantage is that the intersection tests are somewhat more complicated. Adding orientation into the equation allows for a much greater variety in the possible representations of an OBB. The predominantly used approach is to store a corner vertex of the box and three mutually orthogonal vectors that define the local coordinate system of the box, as well as three half-width extents. Some other plausible techniques are to represent the OBB as a collection of vertices, a collection of planes and as three pairs of parallel planes. However, the corner vertex and local axes approach is favoured because it leads to less expensive overlap tests between two OBBs.

3D intersection tests between two OBBs are quite complicated. The test can be performed by using the *separating axis test*. The important idea behind this approach is to find a plane that can be placed between two OBBs, so that they lie on either side. This plane will be orthogonal to a separation axis. With respect to this separating axis, the OBBs are considered separated if, the sum of their projected radii is less than the distance between the projection of their center points. See figure 2.1

It is possible to show that a maximum of fifteen different separating axes must be tested in order to accurately deduce whether two OBBs are colliding or not. The amount of processing for these tests can be reduced by transforming one OBB into the local coordinate system of the other box. It is still a costly process when compared to the simpler bounding sphere and AABB overlap tests.

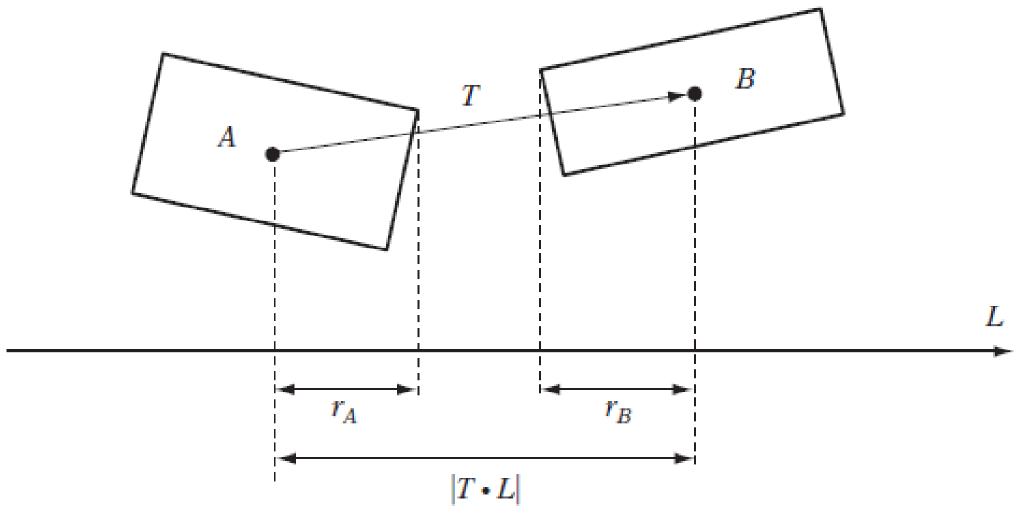


Figure 2.1: Separating Axis Test: The two OBBs are considered separated if the sum of their projected radii on to separating axis L, is less than the distance between their projected centers.

2.3.4 Swept Volumes

As mentioned in section 2.2.3 a *swept volume* can be used to aid in dynamic collision detection, accounting for an object's movement as part of the process. Swept volumes can be used as stand-alone encapsulated volumes for objects in a static collision detection system. They are an alternative to using more complicated geometric shapes, like cylinders. A cylinder has been proposed as a bounding volume and might be an ideal fit for a certain object, but intersection testing between cylinders proves to be quite costly, from a processing point of view. However, if one was to use multiple simpler objects, such as a sphere, to make a cylindrical shape (with rounded edges), the resulting bounding volume becomes a far more attractive proposition in terms of computational expense. This can be implemented by sweeping a sphere along the line between the centres of the cylinder's end-points. The resulting volume is known as a *sphere-swept line (SSL)* which belongs to a group of bounding volumes which are referred to conjointly as *sphere-swept volumes (SSV)*, utilised for fast proximity queries by Larsen et. al in [10]. This SSL or *capsule* can be stored in a data structure by a start point, an end point and a radius. This overlap test for SSVs is relatively simple and follows on from the bounding sphere test, by comparing the distance between the two inner shapes (e.g. line or rectangle) to the sum of their radii.

2.3.5 Slabs and Discrete Orientation Polytopes

Discrete Orientation Polytopes are commonly referred to as *k-DOPs*. Their effectiveness has been demonstrated in [11]. These k-DOPs are based on the idea of *slab based volumes*. A *slab* is defined as the region of space between two parallel planes. Before a bounding volume can be created, a number of normals have to be decided upon. A pair of parallel planes are positioned according to each normal, bounding the object on both its sides along the direction of the normal. At least three slabs are required to create an enclosing volume in three dimensions. AABBs and OBBs are examples of an intersection of three slabs. For AABBs in particular you can visualise the parallel planes on each of the three cardinal axes. As mentioned, k-DOPs utilise the idea of slabs. The "k" in k-DOPs simply refers to the number of discrete orientation polytopes in the bounding volume, for example AABBs and OBBs are described as 6-DOP. k-DOPs are convex polytopes that are basically slab based volumes, except for the fact their normals are defined as a fixed set of axes. Slab based volumes were designed to allow for fast ray intersection tests with objects. Two slab based volumes are not painlessly used for intersection tests between two objects. To overcome this hindrance, k-DOPs allows fast intersection tests by sharing the normals across all k-DOPs objects. As the normals are defined as a fixed set of axes and are being shared between all volumes, intersection tests are quite straightforward and similar to that of AABBs. It simply involves checking $k/2$ intervals for an overlap and if any pair do not overlap, there is no collision.

2.3.6 Other Volumes

As well as these more commonly used bounding volumes that have been described, there have been plenty of other implementations. There are a variety of geometric shapes that have been proposed as bounding volumes, such as cylinders (as mentioned previously) [12], [13], cones [14], [13] and some other interesting ideas in the form of "spherical shells" [15] and the strangely named "zonotopes" [16], which are essentially centrally symmetric polytopes. Most of these approaches incur quite the cost when it comes to intersection testing and for that reason alone they are not frequently used as bounding volumes.

2.4 Broadphase Algorithms

The primary function of broadphase algorithms are to quickly and efficiently remove pairs of objects from being tested by the more costly collision detection algorithms in the narrowphase. There are many different ways to implement this functionality and the focus of this section will be to explore these methods. Most broadphase algorithms are based on the idea of logically grouping objects together so that they can be either tested as a group or to divide space into disjoint regions and check if objects occupy this space. The first set of algorithms use the bounding volumes of objects to build recursive structures known as *bounding volume hierarchies.* The latter set of broadphase algorithms are referred to as *spatial partitioning* techniques.

2.4.1 Bounding Volume Hierarchies

The bounding volumes discussed in section 2.3 are viable options as substitutes for complex objects, when it comes to collision detection. Through this approach, large performance gains can be achieved by simplifying the testing process. A problem remains however, in that the same number of pairwise tests must still be performed between the bounding volumes as would have been for the objects themselves. This problem can be overcome by arranging the volumes into a hierarchical tree structure called a *bounding volume hierarchy (BVH)*, which will effectively cull objects from the list of those to be tested further. The bounding volume hierarchy will consist of leaf nodes, made up of the original bounding volumes in a scene. These leaf nodes are recursively grouped and surrounded by a larger bounding volume. The process repeats until one bounding volume, stored at the root node of the tree, represents the largest bounding volume and the one that encloses all others. The idea behind this hierarchical configuration is that, if a parent node does not intersect with a testing volume, any child node of that parent, does not have to be checked for collision, as it cannot possibly be colliding. These BVHs can be made up of many of the bounding volumes described in section 2.3, but the most popular BVHs consist of the simpler volumes, such as

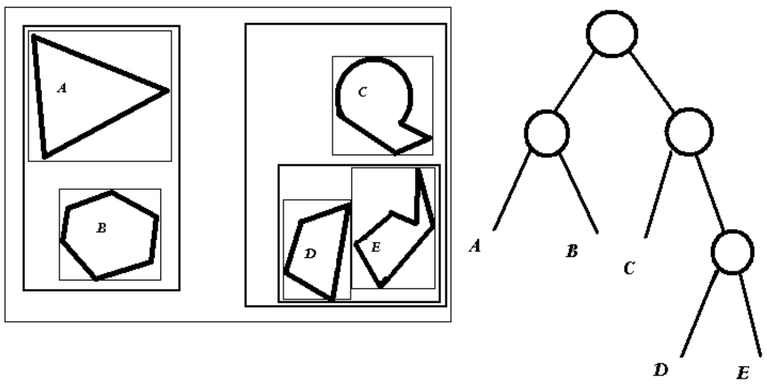


Figure 2.2: A simple BVH in the form of an AABB tree.

axis aligned bounding boxes, oriented bounding boxes and spheres. Rocha evaluates a collision system for rigid body animation in [17]. See figure 2.2

The bounding volume hierarchies built up of these volumes, tend to exhibit the same properties, advantages and disadvantages as the volumes themselves. For example, OBB trees may fit a collection of objects more tightly than say a sphere tree, but it they are more costly to update as they must take orientation into account, whereas a sphere tree does not have to. Like bounding volumes themselves, bounding volume hierarchies have desirable traits of their own. Some desirable characteristics have been discussed by [18] and [19]. They describe how nodes in any subtree should be near each other and that as you progress down the levels of the tree, the objects should be nearer to each other. The volume of nodes should be minimal. There should not be excessive overlapping of sibling nodes and the tree should be well balanced to allow as much of the hierarchy to be pruned as possible. BVH techniques are often difficult to classify under the term broadphase, as they can sometimes take place after an initial broadphase algorithm has completed, sometimes being implemented a separate dedicated stage, that executes between the broadphase and the narrowphase.

2.4.2 Spatial Partitioning

As mentioned in section 2.4, *spatial partitioning* schemes attempt to cull the number of pairwise tests required, by dividing 3D space up into distinct regions. The testing is performed by checking if objects overlap the same region of space. The regions of spatial partitioning techniques are disjointed and are generally allowed to house only one or two objects, in comparison to bounding volume hierarchies, where there is no rule against two or more bounding volumes covering the same regions and objects are usually enclosed by just one volume. Spatial partitioning is a broad topic within itself, with several different techniques being used, such as *grids*, *trees*, and the most important approach of all, from the point of view of this project, *spatial sorting* techniques.

Grids

The imposition of a grid is a very effective method of subdividing space. Space is partitioned into a number of equally-sized cells, by the grid. The closer objects are to each other in 3D space, the more likely they are to occupy the same cells in the grid. Collision detection is performed by checking if two objects share any similar grid cells. If two objects are spaced very far apart, there is little chance of them colliding and this is expressed in the grid as the two objects occupying cells that are very different, as there is a large distance between them. There are two common methods of grid implementation, *uniform grids* and *hierarchical grids.*

The type of grid that has just been described, is a uniform grid. The uniformity allows cell look-ups to be performed with ease. This fast access is a result of a direct correlation with the world space and it is usually just a matter of dividing the world space coordinates to find the corresponding grid cell. If you know the coordinates of a particular cell, it is quite a simple task to find its neighbouring cells. An example of this might be to find neighbouring cells to the left and right of a certain cell, these will simply be located in the grid at the $cellposition + 1$ and $cellposition - 1$ respectively. Conceptually simple and powerful in implementation, uniform grids have found much favour as a method of spatial partitioning.

There are some issues with the size of cells, that must be considered when using grids. Choosing an appropriate cell size can have a large impact on the efficiency with which a grid performs. If the cell size is too small, forming a very fine mesh, a large amount of cells will have to be constantly updated with object information, which takes up more processing time and more space in memory. If on the other hand, the cell size is too large, and the objects are small, there will be far more pair-wise tests, effectively impeding the grid's performance, with respect to the purpose for which it

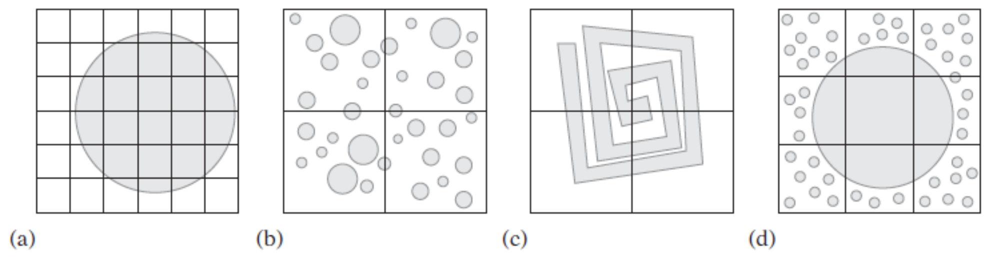


Figure 2.3: Grid cell size issues. (a) Cells that are too small. (b) Cells that are too large for the object. (c) Cells that are too large with respect to the complexity of the object. (d) Cells that are both too small and too large for a varied scene.

was implemented in the first place. If the cell size is too large and the objects are very complex, the larger cells might fit the objects well, but there will still be a large number of pair-wise tests to carry out. Objects should really be broken down into smaller pieces to fit smaller cells. A grid's cells can be considered to be at the same time too small and too large, if a scene contains many objects of different sizes. The grid cells may be too small for some objects and not big enough for others. Figure 2.3 depicts these issues. A more elaborate grid based technique, the hierarchical grid, attempts to address this issue of varying objects sizes.

A hierarchical grid is made of many smaller grids, each with different cell sizes. These sub-grids all overlap to cover the same space. A level must be set for the hierarchy, to determine how many grids it will have. The idea is that the cell sizes increase as the levels increase, with level one having the smallest cell size. As there is no fixed size that cells should adhere to, an effective approach is to select sizes with respect to sizes of objects in a scene. A suitable hierarchical grid can be constructed by setting the lowest level in the hierarchy to house the smallest objects and then doubling the cell size at the next level in the hierarchy, so that the cells are twice as wide at level two as they were at level one. This approach helps to limit the amount of cells being overlapped by a single object. Though these types of grids improve over the non-hierarchical approach, they are not without problems. One such problem is that costly collision tests can be incurred, if objects are tested one at a time. This is due to the fact that they could be covering a large number of cells at the lower levels, with smaller cell sizes. Techniques to combat this problem are presented in [20], see figure 2.4. Here, instead of insertion at only a single cell, objects are inserted in

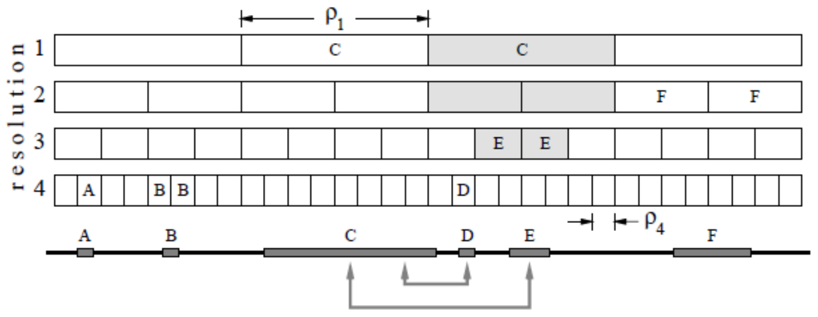


Figure 2.4: Mirtich's 1D example of a hierarchical spatial hash table, with four resolutions. The one-dimensional boxes are shown at the bottom, labeled A-F. The value Pi is the size of the tiles at resolution i. Boxes are stored in the hash table in order of increasing resolution. The cells which must be checked when box E is stored in the table are shaded. This hash table verifies that all boxes are disjoint except for the pairs (C,D) and (C,E).

the hierarchical grid at all cells that they overlap, starting at the lowest level, where the cells can accommodate the size of an object's bounding volume, and working up the levels of the hierarchy. The advantage of this approach is that no other tests are required, if the two objects overlap at the higher of the two insertion levels of the objects, i.e.(at the level of whichever object was inserted higher, in the hierarchy). [21] compares some of these approaches as well as some alternative grid implementations that exist.

Trees

Trees are found in abundance in different broadphase algorithms. They are not only useful for bounding volume hierarchies as discussed in section 2.4.1, but their structure also lends itself quite well to spatial partitioning. Some common trees for spatial subdivision are *octrees*, *quadtrees* and *k-d trees.*

An octree as mentioned in section 2.4.1, is an axis-aligned subdivision of a volume of 3D space, in a hierarchical manner. In an octree, each parent node has eight children, known as *octants* (or cells). The root node represents the smallest AABB that can enclose the entire 3D world. This initial world volume is partitioned into equally-sized octants. This subdivision is achieved by dividing the initial cube in half, along each of the three primary axes. These octants become the child nodes of the root node.

The rest of the tree is constructed by recursively subdividing each of the octants in the same way, until a desired level of sub-division is attained. After a certain number of iterations, sub-dividing the cubes will cease to be beneficial as they will become too small to be of any use. An important property of the tree, is that all descendant nodes are contained in the associated volume of any of their ancestor's nodes. While an octree is designed to divide up 3D space, its 2D analogue is the quadtree. As we move from 3D to 2D, we move from cubes to squares, when talking about quadtrees. The world is enclosed in an axis aligned bounding square and recursively sub-divided into smaller quadrants. The recursive nature of octrees and quadtrees, allows them to be automatically generated with little difficulty. It is difficult for the size of trees to grow and shrink dynamically as they must be pre-allocated to a specific level of depth. With this mind, implementations of trees for generally static scenes tend to favour arrays to store the tree, whereas for more dynamic scenes, pointer based implementations are preferred as the tree must be constantly updated.

A less strict tree based subdivision scheme is the k-dimensional tree or *k-d tree*. Presented by [22], the k represents the number of dimensions to be sub-divided. Unlike quadrees and octrees which are bound to the realm of 2D and 3D respectively, the dimensionality of k-d trees are not required to be the same as the number of dimensions in the world space. Quadtrees and octrees sub-divide space in to two and three dimensions at a time, whereas k-d trees differ by only dividing space by one dimension at a time. Typically, they will cyclically sub-divide along the x-axis, then the y and the z. This is not a requirement however and splits can occur at arbitrary positions along an axis, not necessarily dividing in half each time. The variable nature of this approach demands that both the axis being divided and the point along that axis have to be stored in the nodes of a k-d tree. In terms of collision detection, a k-d tree is directly substitutable for any situation where a quadtree or octree is being used. It has other uses that include finding the region that a point in space occupies, locating the closest neighbouring point of a given point and for finding all other points in the same region as a given point.

Spatial sorting - Sweep and Prune

Spatial partitioning methods that feature grids and trees come with inherent complications that involve objects occupying multiple regions of the structures. However, spatial partitioning techniques are not limited to algorithms involving grids and trees. An alternative approach is to spatially sort the objects in a scene. The most prevalent technique of achieving this spatial sorting is commonly known as the *sweep and prune (SAP)* method, as referred to by Cohen et al. in 1995 [23], although originally presented as *sort and sweep* by Baraff in 1992 [8].

The goal of the SAP algorithm is to determine pairs of overlapping objects. The entire algorithm hinges on the use of axis aligned bounding boxes (AABB), as discussed in section 2.3.2, as each object in the SAP is defined by an AABB. Given a set number objects, the algorithms attempts to find a subset of overlapping pairs.

A SAP implementation can provide *direct* or *incremental* results. A direct SAP provides the full set of overlapping pairs each frame. An incremental SAP provides lists of new and deleted pairs for the current frame. A related but different concept is the way the SAP operates internally: by starting from scratch each time, or by updating internal structures, persistent over the SAPs lifetime. We will call the first type *brute-force* and the second type *persistent*. This definitive explanation of the different ways that a sweep and prune algorithm can be implemented is presented by Pierre Terdiman, in an excellent sweep and prune resource [24]. There is often confusion between the terms "incremental" and "persistent", as sometimes persistent SAPs are referred to as incremental ones. This problem arises usually because of a mis-interpretation between the type of results that a SAP produces and the way the inner workings of the SAP operate. An example of this segregation is evident in [25], which is the successor to [23], where the SAP used is persistent, but provides direct results.

The algorithm works primarily on the basis of *intervals* and *end-points*. An interval is defined by two end-points, which are the minimum and maximum extents of an AABB, orthogonally projected onto one of the cardinal axes. The idea is to handle the x, y and z axes separately, maintaining a list of end-points for each axis. Each of these three lists are sorted at any given time. The main idea of the algorithm is that two AABBs are considered to be completely overlapping, only if they are overlapping on each of the three axes, at the same time. Only by sorting the lists can overlaps be

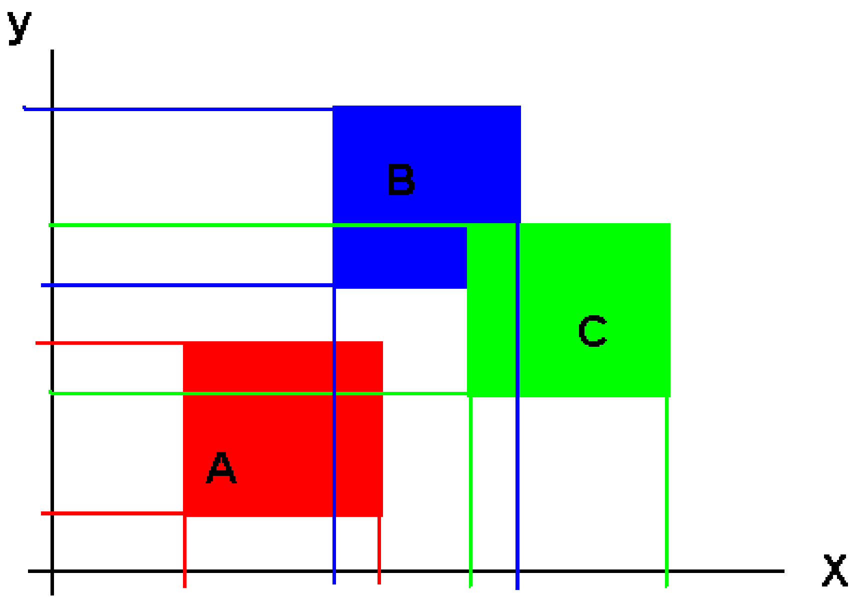


Figure 2.5: 2D Sweep and Prune: a simple diagram showing overlaps on different axes.

determined. One approach to calculate overlapping pairs, is to start at the minimum endpoint of an object's AABB and iterate through the list until the corresponding end-point has been found; the endpoints of any objects encountered along the way, are considered to be overlapping. A simple 2D example is presented in figure 2.5. Notice how box A overlaps with box B on the x-axis but not on y-axis. Box A overlaps with box C on the y-axis but not the x-axis. Only box B and box C overlap on both axes, therefore they are colliding.

The strength of the algorithm lies in its exploitation of *temporal coherence*. After an initial sorting pass of each list, the lists will remain almost sorted, this is because objects do not tend to move very far, on a frame-to-frame basis. To take advantage of this temporal coherence, an insertion sort can be carried out on the list, which is an ideal sorting algorithm for nearly ordered lists. The sweep and prune algorithm can be implemented in a multitude of ways, usually dependant on the type data structures being used. The two most important considerations when designing a SAP are the representations of the sorted lists and some sort of pair management functionality;

implemented as a separate structure that is required to track the overlap status of different pairs. Should the sorted lists be stored as linked lists or arrays? Should the pair manager be implemented as 2D array, array of linked lists or some sort map structure?

Using linked lists allows for a large number of objects, including their dynamic addition and removal, and very little work, if any, is required to maintain the structure between frames. However linked list implementations of the sorted end-point lists can lead to a problem known as *clustering*; this is where coherence breaks down and even small movements in the scene can cause large positional shifts in the list. This clustering of objects can be quite common. An example of where clustering may occur could be along the y-axis (or whichever is the vertical axis), when gravity in a simulation forces all objects towards a floor, objects cannot pass through the floor and they become clustered around this area. In such cases sorting can break down to $O(n^2)$. An acceptable work-around is simply to just use two sorting axes, namely the x and z axes. Arrays are less flexible than linked lists but they also use less memory. An entire sorting pass of array will have to be performed if even a single object moves. However, the clustering of objects issue becomes virtually non-existent and the worst case performance, when it comes to sorting, lies solely on the worst case performance of the sorting algorithm. Arrays also tend to be easier to deal with. Similarly to the sorted endpoint lists, the data structure used to represent the pair manager must also be considered. 2D arrays for managing collision pairs can take up a lot of memory and can be particularly tricky to parse, for determining collisions on all three axes. The original V-Collide system [25] utilised an array of linked lists approach, but this was not without some performance issues. Hashing of the id's of objects is a good idea as less memory will be required to store them and look up times can be drastically reduced. The id's can be stored in a map structure and or even a hash based linear array, which is used to good effect in Terdiman's ICE system [24], which is incorporated in OPCODE[26]. The choice of sweep and prune will be explained in chapter 3 and the implementation details of the sweep and prune algorithm will be discussed in chapter 4.

2.5 Cell Broadband Engine Overview

As mentioned in section 1, The creation of the "Cell Broadband Engine" or "Cell Processor" as it is commonly known, was a collaborative effort between Sony, Toshiba and IBM, who formed the "STI" alliance. The goal of the STI alliance was to develop a groundbreaking processor architecture, one that would be integrated into Sony's then next-generation gaming console, the *Playstation 3*. The processor has a unique design, consisting of heterogeneous cores. It consists of one PowerPC Processor Element (PPE), and eight Synergistic Processor Elements (SPE). It is the primary job of the PPE to initiate and manage the SPEs which are 128-bit vector processing units, designed for Single Instruction Multiple Data (SIMD) processing. To get the most out of the architecture, programmers must really take advantage of its parallel nature and SIMD processing capabilities.

2.5.1 Structure

Other than the control-intensive processor core (PPE) and eight compute-intensive processor cores (SPEs) an important building block of the Cell is a high-speed bus called the Element Interconnect Bus (EIB). The EIB is used for connecting the processor cores within the Cell. The EIB also provides connections to main memory and external I/O devices, making it possible for processor cores to access data. See figure 2.6 for a structural overview of the Cell. [27]

2.5.2 Power Processing Element

The *Power Processing Element (PPE)* can be considered to be the control centre of the Cell. It is responsible for running the operating system, responding to interrupts, and it contains and manages its 512KB L2 cache. It also distributes the processing workload among the SPEs and coordinates their operation. The PPE consists of two operational blocks. The first is the *PowerPC Processor Unit*, or *PPU* and the second is the *PowerPC Processor Storage Subsystem*, or *PPSS*, see figure 2.7 [28]. The instruction set of the PPU is based on the 64-bit PowerPC 970 architecture. The PPU primarily executes PPC 970 instructions, as well as some Cell-specific commands. The PPU is a general-purpose processor and as it is the only such processing unit in the Cell,

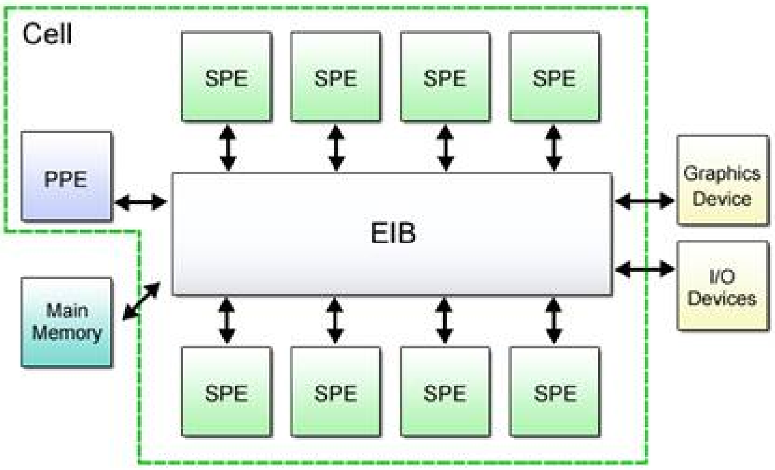


Figure 2.6: Structural diagram of the Cell processor

it is the only thing capable of running the Linux operating system. The PPU has more advanced capabilities also, including *Single Instruction, Multiple Data (SIMD)* processing and *symmetric multithreading (SMT)*, allowing two threads to execute at the same time. The PPSS contains the L2 cache along with registers and queues for reading and writing data [28].

2.5.3 Synergistic Processing Element

The SPEs are not control intensive processors like the PPE and as a result, have a less complex design. The power of the Cell really lies in the efficient use of the *Synergistic Processor Unit (SPU)*, in each SPE, see figure 2.8. The sole purpose of the SPU is for high-speed SIMD operations. Each SPU contains two parallel pipelines that execute instructions at 3.1GHz. In only a handful of cycles, one pipeline can multiply and accumulate 128-bit vectors while the other loads more vectors from memory [28]. Both the SPUs instructions and data, are stored in a unified 256KB local store (LS). The SPU can only access shared memory via its *direct memory access (DMA)* units.

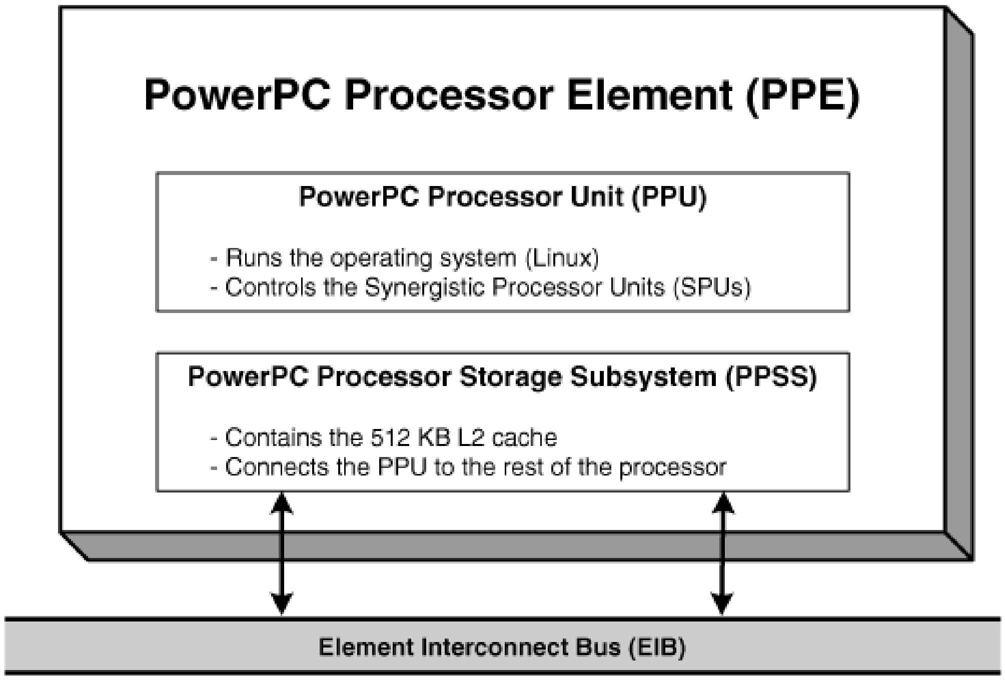

PowerPC Processor Element (PPE)
PowerPC Processor Unit (PPU)
- Runs the operating system (Linux)
- Controls the Synergistic Processor Units (SPUs)
PowerPC Processor Storage Subsystem (PPSS)
- Contains the 512 KB L2 cache
- Connects the PPU to the rest of the processor
Element Interconnect Bus (EIB)

Figure 2.7: Cell Power Processing Element (PPE) layout

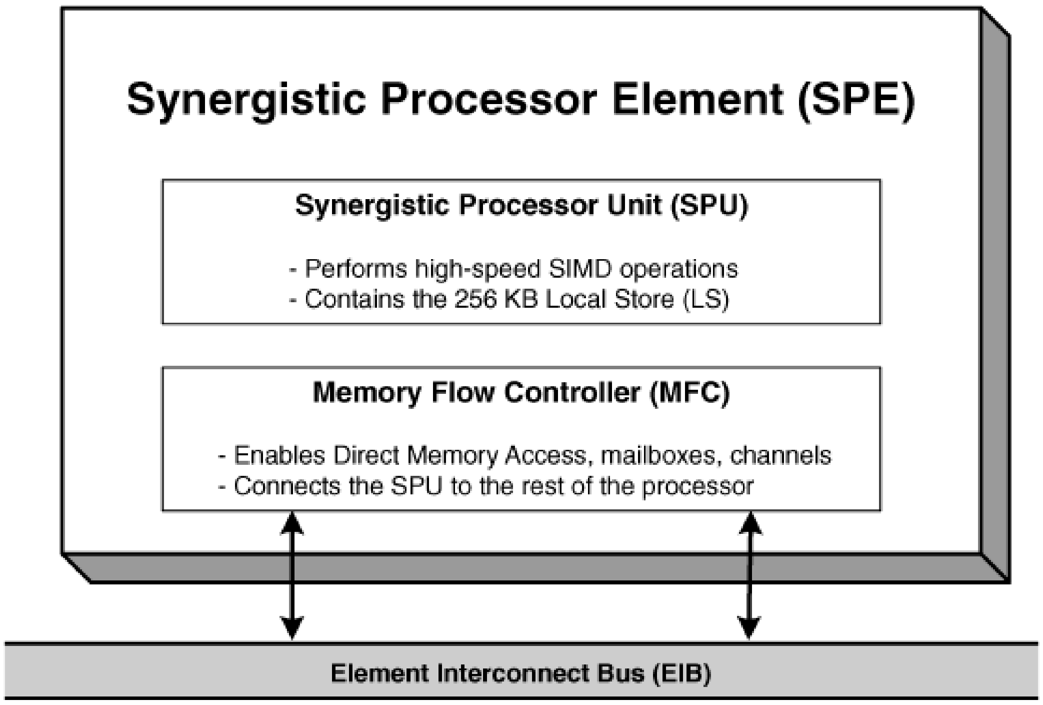

Synergistic Processor Element (SPE)
Synergistic Processor Unit (SPU)
- Performs high-speed SIMD operations
- Contains the 256 KB Local Store (LS)
Memory Flow Controller (MFC)
- Enables Direct Memory Access, mailboxes, channels
- Connects the SPU to the rest of the processor
Element Interconnect Bus (EIB)

Figure 2.8: Cell Synergistic Processing Element (SPE) layout

2.6 Recent Work

To the best of my knowledge, there has been little work published that details broadphase collision detection implementations for the Cell processor. The purpose of this section is to briefly describe some relevant work that has taken place recently with regards to collision detection. Some recent Cell processor projects will also be highlighted.

2.6.1 Broadphase Collision Detection

It appears that much of current research is not concerned with trying to find opportunities for parallelism in existing broadphase algorithms and implementing multi-threaded versions of the algorithms to run on multi-core CPU architectures. Rather, recent work has been more interested in coming up with new methods of implementing collision detection, usually on the GPU, in an attempt to make the algorithms highly parallel. There has been some interest in parallelising other collision detection techniques such as bounding volume hierarchies (BVH), which are hard to classify. As mentioned in previously in section 2.4.1 The BVH algorithms lie somewhere in between the banners of broadphase and narrowphase, sometimes considered as a third intermediary stage altogether.

An interesting GPU based approach is evident in such systems as “CULLIDE” [29], which uses visibility processing to determine colliding pairs of triangles and prune non-overlapping pairs. Another example is some research from Rahul Sathe and Adam Lake of Intel [30], which proposes a novel approach of using cubemaps to store the distances from the centroid (centre) of a rigid body to it boundaries, by casting rays outward from the centre and storing the distance in a cubemap texture. It compares the distance lookup from the texture to the distance between the centroid and a vertex of the rigid body that it is being tested against. Via this method, they can determine intersections.

Some of the most recent work has dealt with bounding volume hierarchies and attempts to parallelise their implementations. Lawlor explored the parallelisation of grid based techniques in his master’s thesis [31] and then went on to work on a parallel voxel based approach soon after [32]. This work presented a scalable high-level parallel solution to a large subclass of collision detection problems. Their approach was to partition space into a sparse grid of regular axis-aligned voxels that would be distributed across a parallel machine. Information regarding objects in the scene that are intersecting the voxels are stored in the voxels themselves. Each voxel becomes a self-contained subproblem, which is then solved using standard serial collision detection approaches. [33] proposed a parallel bounding volume hierarchy based algorithm that created three AABB BVHs for each model, where the idea was to limit the construction of a huge tree containing many leaves. The more recent research has continued in this vein, concentrating on BVHs. In 2007, [34] suggest a dynamic load balancing approach, which is able to handle heterogeneous processors with less difficulty than previous approaches. They claim several advantages over the known parallel collision detection simulations. The main idea of the suggested simulation is keeping the scalability principle while not abandoning the locality principle and the load balancing of the system. Techniques to perform continuous collision detection in parallel over homogeneous processors has been investigated to good effect by [35] and [36], exhibiting vast improvements in performance.

The area of more traditional broadphase collision detection is generally well established with the seminal papers including explanations of the different techniques, such as the aforementioned Baraff's "sort and sweep" [8] and other spatial subdivisions approaches such as grids and octrees [37]. The "Sweep and Prune" approach has been successfully implemented in the "I-Collide" system [23], and that is often used an argument in favour of its usage. Ericson's book [6], which proved a valuable resource as part of this research, is a very popular book which ties together the most important aspects and techniques of collision detection as whole and not just broadphase related techniques. There have of course been some recent forays into parallel broadphase collision detection, most notably by Avril et. al [38]. This paper presents a parallel implementation of sweep and prune that favours the brute-force approach to the algorithm, rather than the persistent method. The basis of the paper is a two stage parallelisation process, where the first stage involves updating the AABBs of all the objects in parallel, then a synchronisation is performed before all of the overlapping pairs are calculated concurrently in the second stage. This work is the most closely related to this project, but the implementation targets up to eight 3.20 GHz Intel Xeon CPUs X5482 homogeneous cores, so it is not directly relatable.

2.6.2 Cell Processor

As mentioned in the introduction to this section 2.6, there does not appear to be a great deal of published work concerning collision detection implementations that target the Cell, let alone specific broadphase related work. There has been one relevant effort that involves porting the Bullet Physics Library to the Cell [39]. This paper does not go into great detail, although it does describe a parallel broadphase algorithm that was developed. The algorithm uses AABBs and is focused on performing brute force collision tests on the SPUs. SIMD is exploited and some performance gains resulted. However, the performance was only compared to a PPU only implementation, when really it should be compared with a standard PC implementation.

Some other collision detection related work was presented by Nils Hjelte in his master's thesis [40]. The focus of this work is not on collision detection however and is primarily concerned with looking at ways to improve the performance of a "Smoothed Particle Hydrodynamics (SPH)" fluid simulation by designing algorithms for parallel execution. The target platform was the Cell processor. A grid based spatial hashing technique is described, as is common for fluid dynamics simulations. He states that SPH is inherently parallel due to the local independent force calculations. Much of the work is concerned with calculating hash values, searching for neighbouring cells in the grid and then calculating interactions in parallel via the SPUs by iterating over hash buckets. The Cell implementation provided some notable improvements, but the project was hindered by lack of access to actual Cell hardware and was limited to working with a simulator, so the results cannot be taken as absolute performance numbers.

Chapter 3

Design

The purpose of this chapter is to explain the motivation behind some of the important design choices made for the project. The choice of broadphase algorithm will be discussed. The chapter will explain the design of an experiment to best test the effectiveness of the implementation. The chapter will also detail the design for a rigid body based physics simulation environment, which is basically a framework that will allow for the broadphase algorithm to implemented. In addition, the chapter will also present the plan for the project implementation.

3.1 Goals

As mentioned previously in section 1.2, the aim of this project is to investigate the applicability of the sweep and prune broadphase algorithm to the Cell processor. In order to effectively satisfy this aim, there are multiple goals that must be met. The goals to be considered in this design stage are:

- To develop a simple rigid body based physics simulation environment that will provide the target for the broadphase algorithm. This implementation will be the same for both x86 and Cell.
- To develop a standard sweep and prune broadphase algorithm for an x86 processor.
- To identify opportunities for parallelism in the broadphase algorithm.

- To implement a parallelised version of the algorithm that exhibits SIMD processing to exploit the capabilities of the Cell processor.
- To compare and contrast the two implementations.

3.2 Experiment

To test the implementation of the sweep and prune broadphase algorithm for the Cell processor, an experiment would need to be designed. As alluded to in the previous section 3.1, there are many facets to the design of this experiment and many crucial goals that must be met. A crux of this experiment and project as whole, is the co-development of an implementation for both an x86 processor and the Cell processor. The reasoning behind this, is to create some sort of control implementation as a basis for evaluation.

3.2.1 Two Implementations

The x86 implementation represents the control in the experiment. Using an x86 processor as the target processor, represents an example of a standard PC implementation. This version is also a single-threaded implementation of the sweep and prune algorithm, adhering closely to principles of a standard version of the algorithm. The x86 part of the experiment provides a solid set of results which can be compared and contrasted to the Cell implementation, which is the focus of the investigation.

The Cell implementation, which is the primary concern of this project, will require some changes to be made to the standard sweep and prune algorithm, if it is to execute effectively. The structure of the algorithm will differ from the x86 version of the algorithm. To make any sort of meaningful contribution and to derive any sort of noteworthy results, it is believed that the best approach is to the test the Cell version of the algorithm against a highly effective sweep and prune implementation that can execute on an x86 processor. Due to the changes in the core structure of the algorithm for the Cell, the experiment will not be a direct comparison of identical code, running on the two target platforms. The purpose of the experiment is not to compare the exact code efficiency, function by function, but rather to compare the manner in which the algorithm operates, construct the two best possible versions of the algorithm (each specific to their target processor) and compare how they deal with different situations and workloads. Only via this approach, can the true applicability of the algorithm be investigated. There is no merit to simply reproducing a carbon copy of a single threaded standard algorithm, testing it on the Cell and stating that it is not applicable because it is too slow. The algorithm has to be designed with the target platform in mind, if the desire is the best possible performance on that platform. Conclusions can later be drawn from the results and the possible strengths and weakness of the two implementations, highlighted.

3.2.2 Simulation Framework

Before any broadphase algorithm can even be considered, there must first exist a scene complete with objects; a virtual environment that requires collision detection. To facilitate this need, a simulation framework would have to be constructed, capable of creating and moving these objects. For the purpose of this experiment, a simple rigid body based system was created that uses simple physics calculations to move the objects. Crucially, it provides the data structures and basic functionality necessary to create a scene to which a broadphase algorithm could be applied. The design of this framework will be examined in more detail in section 3.3.

3.2.3 Test Scenes

The efficacy of the two sweep and prune algorithm implementations was tested on a couple of different aspects. In typical broadphase collision detection, performance is evaluated on a varying scales. The primary elements that are present in these tests are *scene complexity* and whether the scene is generally static or dynamic. One way that a scene can be considered to be complex is if there are many objects present, usually requiring clever methods to mask the associated computation costs such as updating all of the objects, rendering and collision detection. Whether a scene is static or dynamic can have a drastic impact on the performance of a broadphase algorithm and as a result, broadphase algorithms are usually tested in both situations to see how they compare. Pierre Terdiman who wrote a valuable sweep and prune resource [24] as mentioned previously in section 2.4.2, has documented some tests on sweep and prune in this manner [41]. The idea behind the experiment for this project is not much different. The performance of both implementations of the algorithm will be measured in the same finite volume of space, or bounds of the world, and under varying circumstances of motion; a set of initial sorting tests, one static set of tests, where none of the objects move, and one dynamic set of tests, where the scene will be composed of mainly static objects and some dynamic objects. This third set of tests, setting only a portion of the objects as dynamic, is likely to be the most accurate approach to simulating a common scenario found in an actual application, particularly in the example of a game. An example of this assortment, of primarily static objects and only a small amount of dynamic objects in game, could be imagined easily: the player character and all enemies in the scene are dynamic and moving about, but all the obstacles and platforms are static. Each sets of these tests (initial sort, static and dynamic) will be run multiple times with different amounts of objects present in the scene. For the purpose of this experiment, the addition and deletion of objects to the scene will not be considered. The amount of objects in each scene will vary from a relatively low amount *64 objects*, doubling each time to a large amount *1024 objects*. By varying the number of objects in the scene, it can show how well the performance of the algorithm scales, in other words, do the two versions of the algorithm perform as well with a thousand objects as it they do with a hundred?

3.3 Simulation Framework

A framework was required to generate a scene for the broadphase algorithm to work on. This simulation framework would need to allow for both static and dynamic objects. For the purpose of this experiment, the requirements of this system were not too demanding, and as a result, it was felt that it would be a waste of time and energy to integrate an existing open source physics engine into the project. The inclusion of an open source physics engine can be a time consuming process and would require extensive studying of the documentation, before it could actually be used properly. In terms of what would be required for the experiment, it was sufficient to develop a small, sequential, physical simulation framework as part of the project. This framework was first developed for the x86 processor, built with the OpenGL (Open Graphics Library) API.

The idea behind the design of this framework was to separate all the relevant functionality into one large data structure called the "Physics World". Data structures for bounding volumes and for rigid bodies were created also.

The only bounding volume implemented in the framework was the axis aligned bounding box (AABB), as this is what is required by the sweep and prune algorithm. The only type of rigid body implemented was a cube (or box). The type of rigid body used was not actually important as the sweep and prune algorithm works purely on the axis aligned bounding boxes of the rigid bodies anyway. A rigid body structure was implemented nonetheless, as IDs needed to be stored that correspond to the bounding boxes, so that we know what object they are enclosing. Also, it makes far more sense conceptually to include actual rigid bodies in the whole process, also we want to be updating the movement of rigid bodies and have the AABBs get updated accordingly, rather than apply the movement calculations directly to the AABBs.

The *physics world* is responsible for all of the rigid bodies in the scene. Each rigid body has an associated bounding box. The physics world determines how many rigid bodies are in the scene, applies the basic physics for movement calculations to the rigid bodies and renders. As the objects may be dynamic, something was required to stop them from moving too far. As part of the movement calculation (or update process) the rigid body is responsible for updating its own AABB. The default functionality of the system is to dynamically generate the AABBs. This means that although they are the same dimensions as the cube they are enclosing initially, if the cube is to rotate in any direction, the AABB will grow to make sure that it still encloses the rigid body cube. For the actual experiment the orientation of the cubes remain the same, so the dynamic generation of AABB has no effect. It was included as it may provide a future avenue of work as a possible opportunity for parallelism, for the Cell version of the framework. As well as setting up the number of objects in the scene, the physics world is responsible for setting the speed at which the objects move. As mentioned in the previous section 3.2.3, a requirement of the experiment is that all tests take place with a set area of space. The physics world is also responsible for setting these world bounds. The bounds are implemented using six planes, effectively enclosing the virtual world in a cube. The physics world provides some very simple collision detection by checking the positions of each object; if they have passed beyond the bounds of the world, their velocity (which is how far the object moves in a certain direction) is inverted on the corresponding axis, basically changing the object's direction so that it will stay within bounds. To allow for some testing of the broadphase algorithm after it had been implemented, the physics world was also capable of changing the colour of objects that had been marked as colliding, by the broadphase algorithm. This is just a simple collision feedback mechanism as there is no narrowphase collision detection implemented in the framework.

3.4 Broadphase Algorithm

When investigating the applicability of broadphase collision detection to the Cell, many different algorithms had to be considered, as detailed in section 2.4. Some of these algorithms are based on the use of the different types of bounding volumes discussed in section 2.3. This section discusses the thought process that led to sweep and prune becoming the algorithm of choice for the project, as well as the use of axis aligned bounding boxes (AABB).

Typically the choice of broadphase algorithm depends on the type of application and the requirements of that application. The varying facets to a collision detection system, as explained in section 2.2.3, can also have a large bearing on deciding which type of broadphase algorithm may be the most suitable for a particular situation. An example of this could be: if a large number of complex objects of different sizes populate the scene, will a grid based algorithm provide suitable performance and accuracy when partitioning space to track these objects' positions? On the contrary, when choosing a broadphase algorithm for this investigation, the primary consideration is not the requirements of a single application, but rather the requirements of the target platform. The important questions are: what are the strengths and weaknesses of the platform and how can they be respectively exploited and avoided? [28] states that for optimal performance, you need to know how your code runs on the target processor and that nowhere is this truer than when programming the Cell.

The goal of this project is investigate if broadphase collision detection is suitable to the Cell architecture. To achieve this, the algorithm with the best chance at being implemented efficiently on the Cell had to be chosen. The strength of the Cell processor lies in its synergistic processor elements, the SPUs. As mentioned in section 2.5, the multiple SPUs are designed for Single Instruction Multiple Data (SIMD) processing, which means that these vector processing units can perform multiple operations of the same type, in one go. With this in mind, the aim should be to let the SPUs handle as much of the work load as possible, with the PPU acting as a controlling processor, directing the work to and from the SPUs. The largest impediment to this ideal of off-loading work on to the SPUs, is the process of splitting up the work load into separate, independent chunks, so that they can be effectively processed in parallel. This is a notoriously difficult task in nearly all fields of computing as different sets of data often rely on each other, due to the sequential nature of standard single-threaded algorithms, that preceded parallel architectures. To get the best performance out of the Cell processor as a whole, advantage must be taken of its parallel nature and SIMD processing capabilities. The most widely used broadphase methods are all based on sequential single threaded implementations. A key component to this project is the identification of broadphase algorithm that can be parallelised. This remains a difficult problem in the area broadphase collision detection today.

The sweep and prune algorithm was chosen ahead of other spatial partitioning schemes because it exemplifies some traits amenable to parallelisation. While other broadphase algorithms may well be conducive to a parallel-oriented restructuring, the means of creating new parallel versions of the algorithms, were not immediately obvious. Luckily, the characteristics identified in sweep and prune as potentially parallelisable, stood out, as they are embedded in the core functionality of the algorithm. As described in section 2.4.2, the sweep and prune technique uses only axis aligned bounding boxes (AABB) and operates fundamentally on three separate lists of sorted values. Each list represents an axis and as AABBs are the algorithm's bounding volume of choice, the axes in question are the cardinal x, y and z axes. Each list stores the endpoints of every AABB along either the x,y or z axis, for example, the x-axis list will store the minimum and maximum x-coordinates of each AABB. All three lists are kept sorted at all times, via an insertion sort algorithm, which excels at sorting almost sorted lists. The important point to note is that these three lists are entirely separate and operate independently of each other. When different components of an algorithm operate autonomously and do not depend on each other, they are good candidates for parallel processing. If one component does not need to wait for another component to execute fully before it can, then the two components can be processed in parallel, lowering the overall execution time.

The primary reason that the sweep and prune algorithm was chosen for this project was because of its potential for parallelisation; by sorting each of its three lists concurrently on the SPUs. Though the SPUs work more efficiently without excessive branches in code that are inherent to the standard insertion sort algorithm of the sweep and prune method, the work load is still favourable to the recursive nature of other spatial partitioning algorithms such as hierarchical grids and octrees. Compiler options for the SPU can help eliminate branches in code, to optimise execution speeds, but it is harder for compilers to circumvent recursivity. Recursion and branching in code can result in slower execution on the SPUs, due to its streamlined design for powerful vector processing. Another reason for the choice of sweep and prune was its use of AABBs. Although not the focus of this project, there may be some merit to investigating the dynamic generation of AABBs on the SPU, as the work could be split up along the x, y and z axes in a similar manner to the sweep and prune algorithm. In this regard, an alley could be left open for future work and extension to this project.

To the best of my knowledge, AABB based approaches seem to be the most popular when it comes to multi-threaded broadphase implementations. As mentioned previously in section 2.6, of the relevant research found, both [38] and [39] both utilise AABBs. Avril et. al [38] also favoured sweep and prune for their parallel implementation, though a brute force version of the algorithm. They state that "it is one of the most used methods in the broad-phase algorithms because it provides an efficient and quick pairs removal and it does not depend on the objects complexity" and that is one of reasons that they chose it.

3.5 Project Plan

A rough outline of the implementation plan for the project was to:

- Create the simulation framework, using C++ and OpenGL.
- Implement an efficient version of the sweep and prune algorithm, to run on a standard PC with an x86 processor.
- Port this codebase of the simulation framework and standard sweep and prune algorithm to the Cell processor, by running it solely on the PPU of the Cell. No rendering code necessary as the Cell server is without a graphics device.

- Restructure the algorithm with parallel processing in mind, by offloading the sorting of lists on to three separate SPUs. Utilise SIMD capabilities to improve execution speed.

Chapter 4

Implementation

The purpose of this chapter is to give a detailed account of the constructing process for the two different implementations of the sweep and prune algorithm (SAP). The x86 implementation will be described first, not just because this was the order of construction, but also because much of the code is also present in the Cell implementation. The x86 version lays the ground work for the Cell version. The purpose of the Cell adaptation is to restructure some parts of the x86 version in an attempt to make it more parallel friendly, whereas the overall structure of the SAP; what type of data it operates and how it provides results, remains the same as the x86 algorithm.

4.1 x86 SAP

The implementation of the x86 SAP was a very important stage of this project. It was crucial that the standard algorithm worked properly and ran to a satisfactory level. This was essential so that the x86 version could provide a strong foundation for the inevitable Cell adaptation and also to provide a competent alternative when it came time to compare with the Cell version. This section will detail the implementation of the key components of the SAP that was developed for the x86, namely the pair manager, the interval lists and how the insertion sort algorithm operates as part of sweep and prune.

4.1.1 Pair Manager

One of the key aspects of any sweep and prune implementation is the type of pair management functionality employed. This *pair manager* is usually implemented as a separate structure that is required to track the overlap status of different pairs of objects. There are different approaches to the representation of the pair manager, as mentioned in section 2.4.2. For this project a map structure was chosen as it can provide a mapping between keys and values, in this case between the objects in the scene, identified via an ID, and their current collision status. This type of functionality is harder to replicate with arrays. Maps also provide a faster general search time of $O(log\ n)$ than the linear $O(n)$ arrays, due to the fact that maps are sorted internally and are akin to searching through binary search trees. Maps also do not allow duplicate entries by default, so time does not have to be wasted checking if entries exist before adding them to the map. Some different uses of the map based pair manager were tested for feasibility. The pair manager map was implemented to provide incremental, persistent results.

Initially, the map structure stored a pair of IDs, combined into a small data structure, as the keys, and the value for each key was an array of three boolean variables, to denote the collision status for those two objects on the three axes. During the insertion sort process, if a swap occurred, the two objects were tested for an overlap on the current axis being sorted. If an overlap was detected between the two, the pair of objects and their new collision status would be added to pair manager, if they did not already exist. If the pair already existed in the map, their collision status would simply updated. If the overlap test failed and the pair did not exist, they would simply be ignored. There was no problem with this other than that the pair manager map had to be searched for these pairs each time a swap occurred during the sorting process.

As discussed in 2.4.2, this $O(log\ n)$ search time could be improved by somehow hashing the IDs, which could radically lower search times by basically reducing the operation to a look up, rather than a search. This look up operation can be performed in constant time $O(n)$ and could be used to speed up the entire sorting / overlap testing procedure. For the final implementation, it was decided to use a hashing approach to storing keys in the map. The idea was that during the initialisation stage of execution, the map could be populated in full, by adding all possible pairs of objects to the pair

```
int GetHashKey(unsigned int id0, unsigned int id1)
{
  return (id0&0xffff)|(id1<<16);
}
```

Listing 4.1: Hash function to generate a single key from two integers. Works on the lower 16 bits of the numbers

manager map and initialising the collision status of each pair to false on all three axes. Where this differs from the previous approach of storing a pair structure, containing two integers for the IDs, is that the pairs of objects are now stored in the map by generating a single hash value, given two object IDs, which are integers. The hash function used to generate a single hash value from two integers is based on one used in Terdiman's ICE library [24], which basically works by shifting the upper and lower 16 bits of the two integers and merging them into one, shown in listing 4.1. This method of hashing two numbers guarantees that all combinations of two ID values under 2^16 hash to a different key, which is more than enough for the purposes of this project, which deals with a maximum of 1024 objects. Another difference to the previous approach, is that the actual IDs are now stored as part of the value structure, that corresponds to each hash key, where as before the IDs where the key itself. The layout of the key and value structures for the final implementation are shown in figure 4.1, where you can see that the value structure now contains both the two IDs and the collision status arrays. This decision was made so that when it came time to iterate over the entire map, to see which objects had their collision status set to true on all three axes, i.e., are colliding, the actual IDs of these objects could be procured. Another reason for this decision was because it was faster to just pluck out the two IDs from the value structure, rather than to reverse engineer the IDs from the hash key, due to the nature of the hashing function. The code representation for the pair manager, called "axisCollisions" and the value structure, named "PairInfo" is shown in listing 4.2; the key value is simply a regular integer, to store the hash value.

4.1.2 Endpoint Lists

As explained in section 2.4.2 The algorithm works primarily on the concept of *intervals* and *end-points*. An interval in terms of sweep and prune, is a region of one dimensional space defined by two end-points. The two end-points are the minimum and maximum

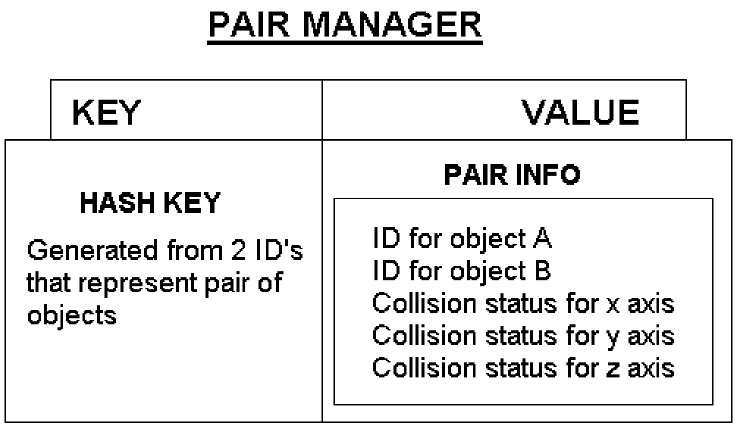

PAIR MANAGER
KEY
VALUE
HASH KEY
Generated from 2 ID's that represent pair of objects
PAIR INFO
ID for object A
ID for object B
Collision status for x axis
Collision status for y axis
Collision status for z axis

Figure 4.1: Pair manager conceptual diagram: mapping a pair info structure containing collision status, to a hash key generated using the two IDs of the pair of objects

```
struct PairInfo
{
  int aID;
  int bID;
  bool value[3];
};

map<int, PairInfo> axisCollisions;
```

Listing 4.2: Pair Info Value structure and Pair Manager Map

extents of an AABB, orthogonally projected onto one of the cardinal axes. The algorithm is fundamentally based on maintaining three lists of end-points, one list for each of the x, y and z axes. These end-point lists are kept sorted every frame by the use of an insertion sort algorithm, for which the implementation will be explained in section 4.1.3. As discussed previously in section 2.4.2 the way that these lists are represented in the code can have different effects on the performance and behaviour of the algorithm. Section 2.4.2 details some of the trade-offs between linked list implementations and array based implementations. For this project the end-point lists are implemented as arrays, primarily because the number of objects / end-points in our scene are fixed so the lists do not need to grow and shrink dynamically, which is what linked lists are good at. Another important reason for using arrays is that they consume less memory than linked lists and they are easier to deal with, especially when it comes to the Cell implementation as we will see in section 4.2.

It may appear as though these end-points can just be implemented as floating point variables, but there is a little more information needed to provide something useful. As shown in listing 4.3, the structure must also contain information about its parent object, represented by the "ownerID" variable. The end-point's parent will not change, but its value will. Its value represents the end-point's position in 1D space. To save processing time that would be wasted copying the new positional values of the minimum and maximum extents of all AABBs into each list each frame, as the AABB moves through the scene, the object can store a pointer to the minimum or maximum extent. By storing a pointer to variable, we are basically storing the address of the variable in memory. As that variable changes, the pointer is privy to this new value that the variable holds. In terms of the array of end-points, it means that the values of the end-points will get updated automatically without the need to copy the new positions of the AABBs' extents. Though no real savings in memory, as a pointer and float both take up the same amount of space, usually 4 bytes, the use of pointers here is a far more elegant approach as end-point structures are updated automatically as the AABBs move. A pointer is also used so that the end-point has direct access to the other half of the interval that it is a part of. Take an example of one end-point representing a minimum extent of an AABB: if this end-point gets swapped in the array when it is being sorted, it can query its parent AABB to receive the maximum extent of the AABB, so that we now have the complete interval and an overlap test

```
struct Endpoint
{
  int ownerID;
  AABB* pBox;
  float* pValue;
};
```

Listing 4.3: Endpoint structure

can be performed with the end-point that it was swapped with, and its full interval. Unlike the pointer to the value, this pointer to the parent AABB is practical from a performance and memory saving perspective, as the pointer takes up less space than an AABB and it maintains a dynamic reference to the AABB as it moves, so that we don't have to keep copying the AABB's other extent information into the end-point structure. The size of each array is equal to the *number of objects in the scene* $x2$, because each object has an AABB and each bounding box has two extents, a minimum and maximum.

4.1.3 Insertion Sort

Although the algorithm can technically work without the use of an insertion sort, by simply performing a regular standard sort of the arrays of endpoints, this method is not very efficient as the arrays are being sorted in full each time, using no previous information to help. A standard sorting algorithm does not take advantage of the temporal coherence in a scene, in the way that insertion sort does. As explained in section 2.4.2, The true strength of the sweep and prune algorithm lies in its exploitation of *temporal coherence*, or *frame-to-frame coherence* as it is sometimes known. This is achieved by retaining the sorted arrays of end-points from the previous frame and performing an insertion sort on each one. After an initial sorting pass of each end-point array, the array will remain almost sorted, this is because objects in the scene do not tend to move very far, on a frame-to-frame basis. The insertion sort is an ideal algorithm for sorting a nearly ordered collection of values. This means that if no objects move in the scene for a certain period of time, the insertion sort will execute very fast, with little or no work to do. The premise behind the insertion sort is similar to the way one might physically sort a deck of cards. For example, they might take the first card from the top of the deck to start with, and start a new pile, then for every subsequent card they take from the original deck, they compare that to cards in the

```
insertionSort(array A)

  for j := 1 to length[A]-1 do
  begin
      key := A[j];
      i := j - 1;
      while A[i] > key and i >= 0 then
          begin
              (Perform Swap)
              temp = A[i+1];
              A[i+1] = A[i];
              A[i] = temp;

              i := i - 1;

              if i < 0 then
                  break
          end
  end;
```

Listing 4.4: Insertion Sort pseudocode

new pile and insert it in its proper place. The pseudocode for the basic insertion sort algorithm used in the project, is presented in listing 4.4 and an example run through of the sorting process is given in figure 4.2.

Of course, for the purposes of the sweep and prune algorithm, some more work is required when performing a swap during the sorting process. Sorting the structure and determining overlapping pairs of objects occur simultaneously, this is important to realise because we want to fully capture the desired behaviour for the algorithm, of not performing any overlap tests if the objects in the scene have not moved. In the implementation, a function was created to insert sort a single end-point array along one axis and perform the overlap tests as part of the sort. To allow the same code to be reused for the sorting of the end-point arrays for all three axis, two parameters were used: the array to be sorted and an integer to represent the axis to sort on, i.e., 0 = x, 1 = y and 2 = z. The actual code implementation of the insertion sort algorithm in listing 4.4 may differ slightly from the basic one, from listing 4.4, but the overall structure is clearly identifiable. Notice that before the swap is performed between the two elements in the list, their IDs are stored in a collision pair structure, which is just a convenient way of storing two IDs together in the same structure. After the swap is performed, the current axis being sorted is checked and depending on whether the x, y or z is the current axis, the appropriate AABB box extents are stored, to be used for an overlap test. If the overlap test proves true or false, then the collision status of that pair is updated accordingly in the pair manager map. The pair can be referenced in the map easily by generating their hash key and simply performing a look up in the

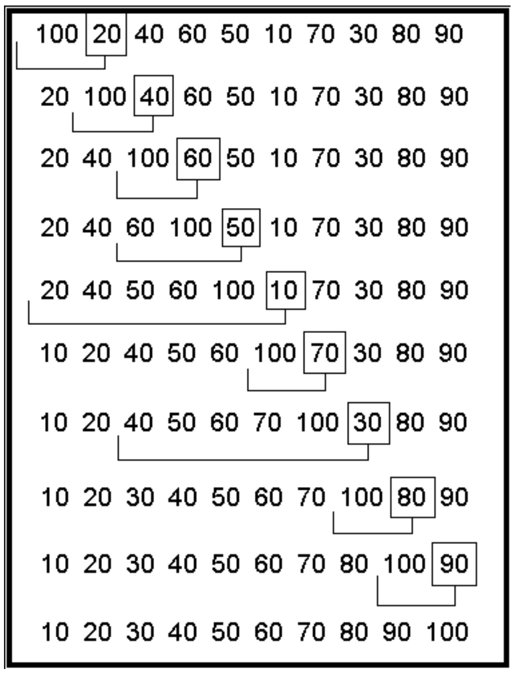


Figure 4.2: Insertion Sort process of sorting set of 10 numbers.

map, based on this key. Notice how the object IDs are always sorted before the hash key generation, this insures that duplicate ID pairs will not exist in the map, as the hash key generated for say, *id:1* and *id:2* will be different from the hash key generated for *id:2* and *id:1*, due to the nature of the hash function.

```
void SweepAndPrune::SortAndTest(Endpoint* pAxisList, int axis)
{
  // Perform insertion sort and update active collisions map (pair manager)
  // Iterate for NUM_BOXES*2 because for every n boxes there is a min and max stored
  for(j = 1; j < NUM_BOXES*2; j++)
  {
    // The key is always equal to one index ahead of i
    current = pAxisList[j];
    i = j-1;

    // Compare the values at index of key and of i
    // If value at i is greater, we need to swap
    while(*(pAxisList[i].pValue) > *(current.pValue) && i >= 0)
    {
      // Create a collision pair
      Pair collisionPair;

      // Store IDs of owners (rigid bodies)
      collisionPair.aID = pAxisList[i].ownerID;
      collisionPair.bID = pAxisList[i+1].ownerID;

      // Perform the swap in the list //
      temp = pAxisList[i+1];
      pAxisList[i+1] = pAxisList[i];
      pAxisList[i] = temp;
```

```
        // Determine min and max extents for the two recently swapped AABB's
        // based on the current axis being sorted
        switch(axis)
        {
        case 0:
          {
            min = pAxisList[i].pBox->min.x;
            max = pAxisList[i].pBox->max.x;
            minNext = pAxisList[i+1].pBox->min.x;
            maxNext = pAxisList[i+1].pBox->max.x;
          }
          break;
        case 1:
          {
            min = pAxisList[i].pBox->min.y;
            max = pAxisList[i].pBox->max.y;
            minNext = pAxisList[i+1].pBox->min.y;
            maxNext = pAxisList[i+1].pBox->max.y;
          }
          break;
        case 2:
          {
            min = pAxisList[i].pBox->min.z;
            max = pAxisList[i].pBox->max.z;
            minNext = pAxisList[i+1].pBox->min.z;
            maxNext = pAxisList[i+1].pBox->max.z;
          }
          break;
        }

        // Check for overlap
        if((min <= minNext && max >= minNext) || (min <= maxNext && max >= maxNext))
        {
          // Make sure IDs are always stored / referenced in the same order
          Sort2(collisionPair.aID, collisionPair.bID);
          key = GetHashKey(collisionPair.aID, collisionPair.bID);
          axisCollisions[key].value[axis] = true;
        }

        else
        {
          // Make sure IDs are always stored / referenced in the same order
          Sort2(collisionPair.aID, collisionPair.bID);
          key = GetHashKey(collisionPair.aID, collisionPair.bID);
          axisCollisions[key].value[axis] = false;
        }

        i--;

        if(i < 0)
          break;
      }
    }
}
```

Listing 4.5: SortAndTest function responsible for performing the insertion sort and overlap tests for a single end-point array

4.2 Cell SAP

While the development of the x86 version of the sweep and prune algorithm, as explained in section 4.1, was very important as a basis for comparison with the Cell implementation in the eventual experiment, it also formed the basis of the Cell implementation itself. It was a necessary stage in the implementation of this project as a whole. However, the development of the Cell code was of paramount importance, as the title of this dissertation is "Broadphase Collision Detection on the Cell Processor", after all. Before the project, there had been no previous experience gained, in working with the Cell processor and developing code for it. As a result of this, the implementation could only be designed as a high level overview without any hands-on time with the processor. The concise plan presented in section 3.5 was really all that could be afforded, before more knowledge and experience could be acquired. From this stage on, the design and implementation became intertwined, in a constant process of testing out ideas and determining their feasibility. The premise behind this section is to detail the most pertinent segments of functionality, developed during the course of this project, that became the final version of the Cell sweep and prune algorithm. An overview of the unique algorithm developed, will be presented and the steps of the algorithm will be explained in more detail in the sections that follow.

4.2.1 Algorithm overview

An attempt was made initially, to test the feasibility of simply taking the existing insertion sort and overlap test code from the x86 version and implementing it on the SPUs of the Cell. The idea being that by running this code simultaneously, a speed up over the x86 implementation would be achievable. This idea was not without its problems. In short, this approach was hindered by lots of time consuming copying of data and impeded by a lack of SIMD functionality. With these issues in mind, it made sense that this approach would not yield great performance results and eventually led to the inception of a completely new approach. The programming concept used throughout the Cell implementation was a simple looping, that involved the PPU running the simulation and when it came time to execute the sweep and prune, it would assign the correct arrays to be passed to the SPUs, start three SPU thread (one for sorting on each axis) and then wait for them to complete their execution and receive the results back via *Direct Memory Access (DMA)* transfers.

The aim of this new algorithm was to mimic the functionality and strengths of the sweep and prune algorithm, but in a way that was better suited to the architectural constraints of the Cell processor, namely exploiting the powerful SIMD nature of the SPUs and giving more consideration to the DMA requirements, which are necessary to transfer data to and from the SPUs. The new algorithm discards some of the ideas of the x86 implementation, namely, storing end-point values in large data structures, that contain the ID of the parent object and pointers to the position values and to parent bounding boxes. Instead, the values are just stored as floating point variables in three standard arrays, one for each axis, the same as before. The most serious change to the previous algorithm, concerns the sorting process, disposing of the old sequential insertion sort method described in section 4.1.3. The algorithm is now centred around the idea of a vectorised insertion sort, i.e., one that can be performed using SIMD. There are a few steps involved in extending a SIMD insertion sort algorithm to make it useful as part of the sweep and prune process. A key factor in recreating the behaviour of sweep and prune is the introduction of three arrays of object IDs, that correspond to values in the end-point values arrays. The correspondence lies in the position of each value in the end-point arrays and the position of each ID in the ID arrays, e.g., an endpoint value at array index = 10 in the end-point array for the x-axis, will belong to the object ID at index = 10 in the ID array for the x-axis. The idea is to sort the values array but apply the same swaps in array position to IDs array, so that the link between the two sets of arrays is maintained. This idea will be explained more clearly in section 4.2.2.

By maintaining the value and ID arrays from the previous frame, we can perform now perform a comparison between the new sorted values array and the old value array, to see which values have changed. This is performed in SIMD also, to speed up the process. In this stage we can identify the IDs of the objects that have moved since the last frame, recreating the persistent sweep and prune behaviour. The SIMD nature of the comparison function gives us an array containing the correct IDs that have changed, but in an awkward layout; dispersed among many zeroes. Some further array manipulation stages are required to turn the array of changed IDs into an easily parsable group. From here, we basically want to perform the overlap tests on these IDs that have changed since the last frame. Two methods for achieving this were implemented.

The first involves finding the minimum value of that ID in the sorted value array and iterating over it until the maximum value is found, with all IDs encountered along the way, considered overlapping objects. The second approach is to perform a brute force collision check of these objects that have changed, against all other objects in the scene. A high level overview of the algorithm is appears as follows:

- Parallelised over 3 SPUs 1 per axis
- Sort an endpoint (values) array and a corresponding IDs array for each axis based on the SIMD Insertion Sort algorithm
  - Maintain previous ID and value arrays and compare (SIMD)
- Reverse sort this ID changes array
  - e.g. after SIMD comparison array is of the form 030434010
  - After reverse sort it becomes - 443310000 (zeroes at the end)
- Remove Duplicates
  - Example array becomes 4310
- Now we can:
  - Find IDs between min and max value of each ID that has changed
    - Iterate over sorted IDs array until ID is found (minimum)
    - Continue and store any encountered ID until test ID is met again which would be the maximum
    - All of these encountered IDs will be overlapping with the test ID
  - Or
  - Perform brute force overlap tests with each ID that has changed
    - Test each changed ID with every other object

Each stage of the algorithm will be explained in more detail in the subsequent sections. As the values and IDs are stored in separate arrays and not in the one structure, as in the x86 implementation, their values do not get updated automatically as the objects (and their AABBs) move through the scene. A necessary step of this algorithm is that the PPU performs an update of all of the values, each frame. This can be achieved after receiving the array of IDs from the SPUs, the IDs that got manipulated as part of the sorting process, i.e., the IDs that represent the sorted values. Once SPU execution has finished, we can use this updated IDs array to index into a list of objects in the scene. The code implementation for this update functionality is present in listing 4.6. The *rigidBodies* array represents the array of objects in the scene. Notice how the ID array received from the SPU *out_IDs_X* is being used to find the corresponding bounding box in the rigid bodies / objects array. A simple flag called *met* is used to determine if the value is a minimum end-point or a maximum end-point. The process must be applied on all three axes. The arrays *inX, inY, and inZ* store the values and these arrays are the input arrays for the next execution of the sweep and prune algorithm. Data was transferred between SPUs and the PPU via Direct Memory Access (DMA) commands. An example of a DMA transfer of an array of data from the SPU, to the PPU is presented in listing 4.7. The important function is *spu_mfcdma64*, where the array *in_values_copy* is being sent to an address in the *effective address* space (EA) (or main memory) that only the PPU has access to. The upper and lower parts of this address in EA must be split into its upper and lower bits, using *mfc_ea2h* and *mfc_ea2l* respectively. The number bytes that we are sending is equal to the number of end-points in the scene by the size of a floating point variable. The amount of bytes being transferred must equal a power of sixteen. The two function calls that follow, simply ensure that the DMA transfer has completed.

4.2.2 Vectorised Insertion Sort

The SIMD based insertion sort algorithm, that formed the backbone of the Cell sweep and prune implementation is described by Matthew Scarpino in his Cell programming book, [28]. The book describes the use of SIMD operations for the PPU, using the "vec_" prefix. The SPU differs slightly and SIMD function calls on SPU require "spu_" instead of "vec_". However this Cell implementation section explains ideas in terms of "vec_" as that is what was used. This was made possible by including a header file for SPU code called *vmx2spu.h*, which basically performs the conversion automatically. Though performing an insertion sort in a similar conceptual manner to the standard

```
float val;
AABB* temp;

  // We need to update the new positions of the AABB's
  for(int i = 0; i < EP_LIST_SIZE; i++)
  {
    ////////////////////////    X    /////////////////////////

    temp = rigidBodies[out_ids_X[i]]->boundingBox;

    if(temp->met[0] == false)
    {
      val = temp->min.x;
      temp->met[0] = true;
    }

    else
    {
      val = temp->max.x;
      temp->met[0] = false;
    }

    inX[i] = val;

    ////////////////////////    Y    ////////////////////////

    temp = rigidBodies[out_ids_Y[i]]->boundingBox;

    if(temp->met[1] == false)
    {
            val = temp->min.y;
            temp->met[1] = true;
    }

    else
    {
            val = temp->max.y;
            temp->met[1] = false;
    }

    inY[i] = val;

    ////////////////////////    Z    ////////////////////////

    temp = rigidBodies[out_ids_Z[i]]->boundingBox;

    if(temp->met[2] == false)
    {
            val = temp->min.z;
            temp->met[2] = true;
    }

    else
    {
            val = temp->max.z;
            temp->met[2] = false;
    }

    inZ[i] = val;
  }
```

Listing 4.6: Updating the values array, based on IDs array after sorting.

```
spu_mfcdma64(in_values_copy, mfc_ea2h(abs_params.ea_out), mfc_ea2l(abs_params.ea_out),
             EP_LIST_SIZE * sizeof(float), tag, MFC_PUT_CMD);
spu_writech(MFC_WrTagMask, 1 << tag);
spu_mfcstat(MFC_TAG_UPDATE_ALL);
```

Listing 4.7: Example DMA operation

insertion sort algorithm, that can be likened to a person sorting a deck of cards, the vectorised insertion sort is actually quite a different algorithm to the standard sort. As we are using SIMD functionality to sort numbers in sets of four simultaneously, the regular insertion sort algorithm is simply not up to the task; some additional steps are required. The basis of Scarpino's SIMD insertion sort, is a two stage process, that consists of what he describes as an *intervector sort* and an *intravector sort*. The two sorting passes are necessary to sort an array in full. A vector in terms of SIMD, can store 128 bits of data. Remember from section 2.5 that the SPU's are purpose built for processing these 128 bit vectors. A "char" variable takes 1 byte (8 bits) of memory to store, while integers and floating point variables typically take 4 bytes (32 bits). From this knowledge, it is evident that we can store either 16 characters, 4 integers or 4 floats in one vector. For the purpose of this section, we can think of a vector as a structure that contains four floating point values, so for example, an array of eight values can be stored in two vectors.

Intervector Sort

The purpose of the intervector sort stage, is to order the vectors in the array. The four values within each vector will not be sorted after this stage has completed its execution, which is why a second intravector sort stage is necessary. A goal of this stage is to limit the amount of branch statements, as they can be costly to execute. Scarpino [28] explains that the SPU is designed to process instructions in a sequential manner and that branching instructions disrupt the instruction pipeline. Branch prediction is possible for SPU code, but a wrong prediction can result in a penalty of 18 wasted clock cycles. To avoid branches, a continuous process of finding the maximum and minimum values of two vectors is used extensively. During the intervector sort, the vectors will be ordered from lowest to highest based on their contents, i.e., the four elements that they contain. It is important to note that these vectors will not contain the values of the array as if the array was parsed from beginning to end, assigning the first four float values to the first vector, the second four values in the array to the second vector, and so on. The intervector sort uses a similar insertion sort looping mechanism to that described in section 4.1.3, but a new method of comparison is needed for vectors. This is achieved via the SIMD operations *vec_max* and *vec_min*, which return vectors

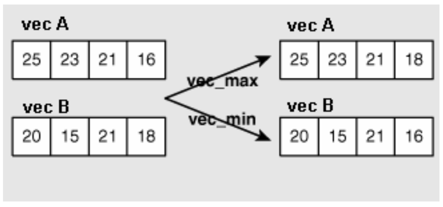


Figure 4.3: Intervector sort splits two vectors into two new vectors: one containing the maximum values of the original two and one containing the minimum values of the original two.

containing minimum and maximum elements of the two input vectors, see figure 4.3. As you can see from the previous figure, *vec_min* and *vec_max* can only perform the comparison between corresponding numbers in the arrays, e.g., between the value at index 0 in vec A, which is 25 and the value at index 0 in vec B, which is 20. In order to correctly ascertain the minimum and maximum values of two vectors fully, all max elements must be compared against all of the minimum elements, to make sure that no value in the final minimum vector is higher than any value in the final maximum vector. For this reason, a second SIMD operation, *vec_perm*, is used. The function allows us to rotate the elements of the vector (or permute them). An example of a permutation could be a vector that has the values *1,2,3,4*, that after permutation by one element becomes, *4,1,2,3*. By permuting the vector by one element, four times, we can compare all the necessary elements to each other. This might be made more clear by figure 4.4, in which a full run through of the process is shown. With regard to the figure, if you compare the two input vectors on the left of step 1 to the final vectors on the right of step 4, you can see that we find the proper maximum and minimum values from the original two vectors. Again, as Scarpino describes, this method produces two clearly ordered vectors, but it doesn't order the elements inside the vectors. A second stage is needed.

Intravector Sort

At this stage we have a set of ordered vectors, but the elements within those vectors are not sorted. It is the job of the intravector sort stage, to sort the values inside each vector. This is a somewhat more complicated process than the previous intervector sort stage, but conceptually it is quite simple. The intravector sort takes a similar approach

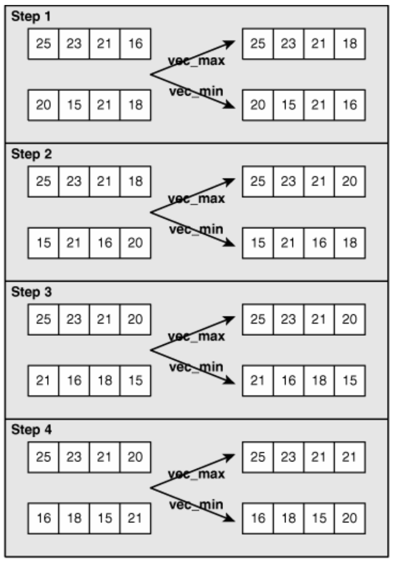


Figure 4.4: Intervector sort process in full, performing four permutations and min / max splits.

of comparing and rearranging the vectors to make sure that all necessary elements are compared to each other. Instead of using *vec_max* and *vec_min* however, it uses a different SIMD operation, *vec_cmpgt* or "compare greater than or equal to". This function compares the elements in two vectors and when the corresponding elements have a greater-than relationship, the function returns a vector whose elements are all ones, and will return all zeroes if they are less than or equal. An example of this could between the vector A 1,2,3,4 and vector B 5,1,6,3, a *vec_cmpgt* of these vectors will return a vector containing 0,1,0,1. This resulting vector is known as a *mask*, which basically provides information about an operation that just occurred, with the respect the vectors involved. Using this mask vector, we can perform a SIMD bit-wise operation, *vec_and* to pluck out elements from a separate array of values. With this in mind, the intravector sort works by in three stages of comparing and rearranging vectors, as shown in figure 4.5. In order to follow the correct sequence and compare the correct elements at each step, a number of pattern vectors are constructed. The pattern vector is defined in terms of 16 characters as opposed to four integers or floats. A pattern describes a specific order in which we want the elements of a vector to occur., for example the pattern 12,13,14,15,8,9,10,11,4,5,6,7,0,1,2,3 describes a reverse ordering of a vector. The pattern vectors are what are used to change a

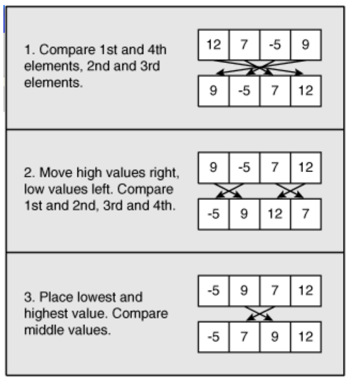

1. Compare 1st and 4th elements, 2nd and 3rd elements.
12
7
-5
9
9
-5
7
12
2. Move high values right, low values left. Compare 1st and 2nd, 3rd and 4th.
9
-5
7
12
-5
9
12
7
3. Place lowest and highest value. Compare middle values.
-5
9
7
12
-5
7
9
12

Figure 4.5: Intravector sort stages of comparing different elements to each other.

```
vector unsigned char perm_reverse =
   {12,13,14,15,8,9,10,11,4,5,6,7,0,1,2,3};
vector unsigned char add_index1 =
   {12,12,12,12,4,4,4,4,4,4,4,4,12,12,12,12};
vector unsigned char basis_index1 =
   {0,1,2,3,4,5,6,7,4,5,6,7,0,1,2,3};

   mask = vec_cmpgt(vec, vec_perm(vec, vec, perm_reverse));
   pattern = vec_and((vector unsigned char)mask, add_index1);
   pattern = vec_add(pattern, basis_index1);
   vec = vec_perm(vec, vec, pattern);
```

Listing 4.8: Intravector Sort comparison example

vector in order for a comparison. In step 1 we want to use *vec_cmpgt* to compare a vector to its own reverse, so permute (rotate) the vector by this reverse pattern 12,13,14,15,8,9,10,11,4,5,6,7,0,1,2,3. In code, this appears as in listing 4.8. The result of the *vec_cmpgt* operation determines how the vector should be rearranged for the next comparison. We use the mask from this comparison operation to pluck the specific values out of another pattern called "add_index1" and create a new pattern, then we add this new pattern to a third pattern called "basis_index1", this all ensures that the correct elements are compared and swapped if necessary and the vector is suitably rotated for the next set of comparisons in step two, where a different set of patterns are required. The code implementation is given in listing 4.9 and is from [28].

4.2.3 Extending the Sort Process

This is all well and good, being able to perform an insertion sort on an array of values, using SIMD, but how can this be used to perform the functionality required by the sorting process of the sweep and prune algorithm? Remember from the x86

```
/* Perform intra-vector sort */
void sort_elements(int* array) {
   int i, numVec = N/4;            /* Number of vectors */

   /* Vectors used in the first compare-arrange */
   vector unsigned char perm_reverse =
      {12,13,14,15,8,9,10,11,4,5,6,7,0,1,2,3};
   vector unsigned char add_index1 =
      {12,12,12,12,4,4,4,4,4,4,4,4,12,12,12,12};
   vector unsigned char basis_index1 =
      {0,1,2,3,4,5,6,7,4,5,6,7,0,1,2,3};

   /* Vectors used in the second compare-arrange */
   vector unsigned char perm_in_out =
      {4,5,6,7,0,1,2,3,12,13,14,15,8,9,10,11};
   vector unsigned char add_index2 =
      {4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4};
   vector unsigned char basis_index2 =
      {0,1,2,3,0,1,2,3,8,9,10,11,8,9,10,11};

   /* Vectors used in the third compare-arrange */
   vector unsigned char perm_middle =
      {0,1,2,3,8,9,10,11,4,5,6,7,12,13,14,15};
   vector unsigned char add_index3 =
      {0,0,0,0,4,4,4,4,4,4,4,4,0,0,0,0};
   vector unsigned char basis_index3 =
      {0,1,2,3,4,5,6,7,4,5,6,7,12,13,14,15};

   vector bool int mask;
   vector unsigned char pattern;

   /* The intra-vector sort */
   for(i=0; i<numVec; i++) {
      vector signed int vec =
         vec_ld(i*sizeof(vec), array);

      /* First compare and rearrange */
      mask = vec_cmpgt(vec, vec_perm(vec, vec, perm_reverse));
      pattern = vec_and((vector unsigned char)mask, add_index1);
      pattern = vec_add(pattern, basis_index1);
      vec = vec_perm(vec, vec, pattern);

      /* Second compare and rearrange */
      mask = vec_cmpgt(vec, vec_perm(vec, vec, perm_in_out));
      pattern = vec_and((vector unsigned char)mask, add_index2);
      pattern = vec_add(pattern, basis_index2);
      vec = vec_perm(vec, vec, pattern);

      /* Third compare and rearrange */
      mask = vec_cmpgt(vec, vec_perm(vec, vec, perm_middle));
      pattern = vec_and((vector unsigned char)mask, add_index3);
      pattern = vec_add(pattern, basis_index3);
      vec = vec_perm(vec, vec, pattern);

      vec_st(vec, i*sizeof(vec), array);
   }
}
```

Listing 4.9: Intravector Sort comparison example

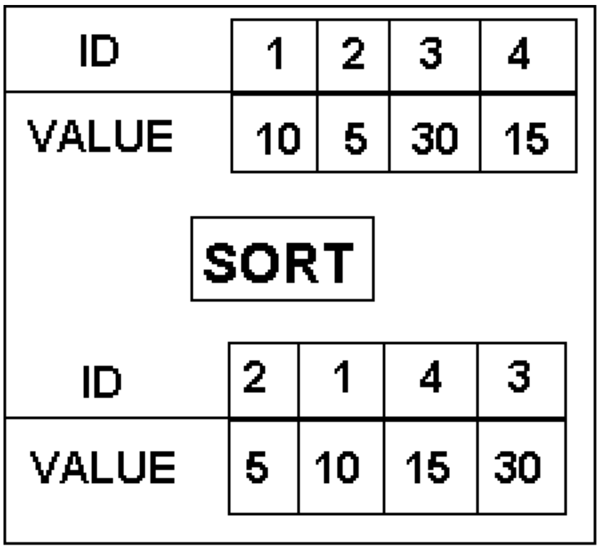


Figure 4.6: ID array and Value array. Value is sorted, but the same positional swaps applied to ID array.

implementation in section 4.1.3, how the insertion sort was sorting arrays of end-point data structures. This SIMD version can only operate on a single array of values. Sorting an array of values by themselves has no effect on our sweep and prune algorithm, we need some way to make it meaningful. To overcome this problem a separate set of ID arrays can be created, as explained in section 4.2.1. The idea is that these ID arrays correspond directly to the value arrays, based solely on their position in the arrays, using the same example as before: an endpoint value at array index = 10 in the end-point array for the x-axis, will belong to the object ID at index = 10 in the ID array for the x-axis. The idea is to sort the values array but apply the same swaps in array position to IDs array, so that the link between the two sets of arrays is not corrupted. This idea is perhaps shown more clearly in figure 4.6, notice that after the sort, the values have changed position, but the relative array index in the ID array, still hold the correct ID for that value.

Applying the changes that the sort triggers, to the ID array, is no trivial task and must be performed during both the intervector and intravector sorting stages. Starting with the intervector sort, the idea to obtain the correct IDs, consists of a two part process, that must be performed after all four min / max vector splitting operations, described in section 4.2.2. The first step is to find the new *max IDs*, or the IDs that correspond to the new max vector after each min / max operation. We can then use some of this information to find the *min IDs* at each of the four min / max operations. The process will be described for one min / max operation, as it is repeated in kind for the following three. Four vectors are maintained during the process: *min_values*, *max_values*, *min_IDs* and *max_IDs*.

To find the max IDs, we first perform the min / max operation on the value vectors and then compare the new *max_values* vector to the original *max_values* vector, using *vec_cmpeq*, which is exactly the same as *vec_cmpgt* as described in section 4.2.2, except that the operation being performed is "is equal to" rather than "is greater than". We store the result in a mask called *maxMask*. By performing a SIMD bitwise *AND* between maxMask and the original *max_IDs* vector, we can obtain the IDs that have changed since the min / max split operation. We store these changed IDs in a new vector *max_IDs_diff*. We also create a *minMask* and set it to have the opposite values of *maxMask*. This *minMask* can now be used to pluck out the IDs from *min_IDs* and store them in a new vector *min_IDs_diff*, via a bitwise *AND*. By performing a bitwise *OR* between the *max_IDs_diff* and *min_IDs_diff*, we get the new set of IDs for *max_IDs*. The same process is reversed to obtain the new *min_IDs*. The process for the first step of finding the maximum values of two vectors and finding the corresponding IDs is shown in figure 4.7, to find the min values, the process is reversed.

This is all just for the intervector sort. The intravector sort is a degree less complex. For this we can simply apply the derived permutation vector as described in section 4.2.2, to the IDs array. This method causes an issue if there are two of the same values in the different arrays, however. The problem is that, the algorithm does not count this as a swap and so the IDs do not get updated as they should. This can be overcome by effectively forcing the two identical values to swap. To force this swap, two masks are required. One mask stores the result of a *vec_cmpgt* and the second stores the result of a *vec_cmpeq* operation. Depending on the position of elements being compared in the vector at that time, for one of the masks, some of their values are set to zero. Performing a bitwise *AND* of the two masks now, will force a swap and will overcome the issue. This is shown in listing 4.10.

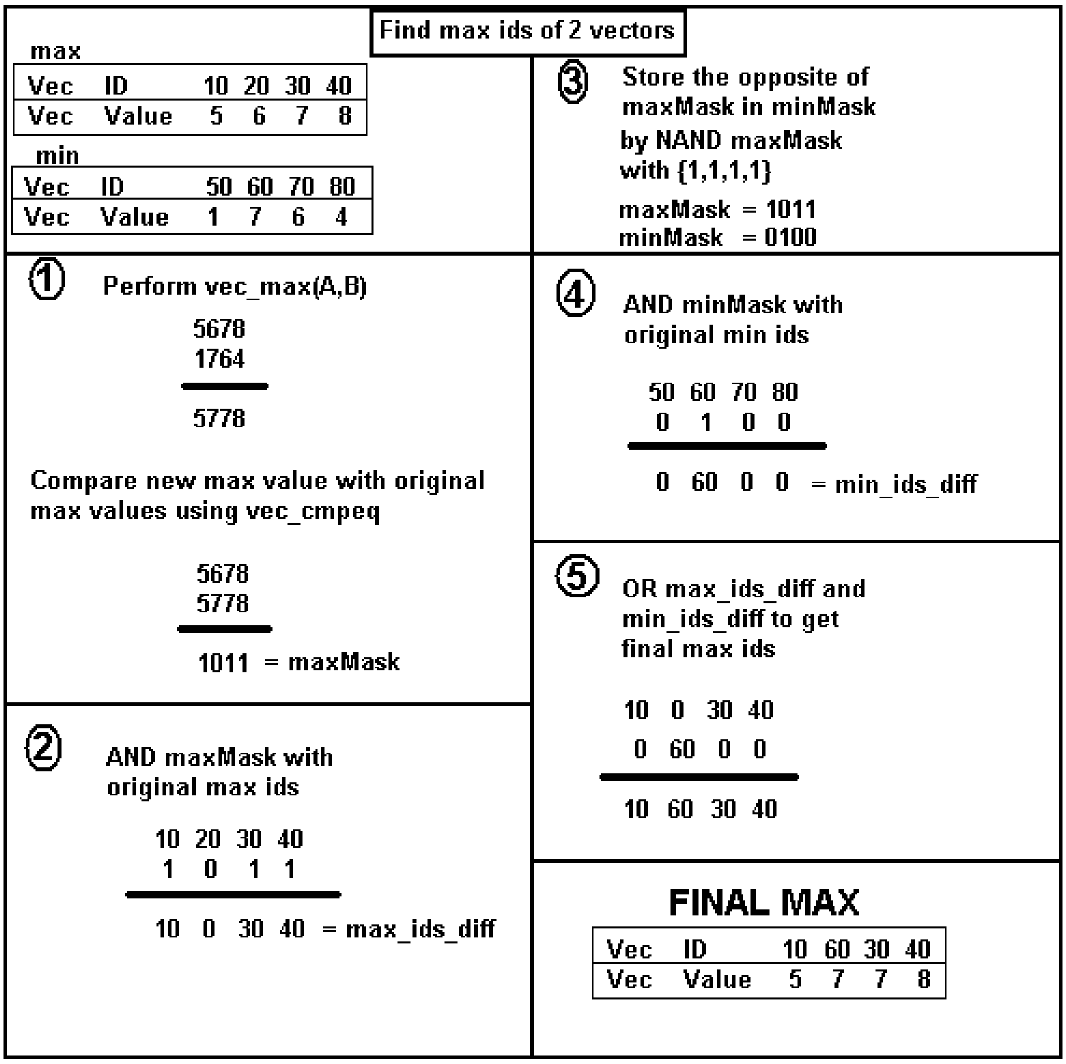

Find max ids of 2 vectors
max
Vec ID 10 20 30 40
Vec Value 5 6 7 8
min
Vec ID 50 60 70 80
Vec Value 1 7 6 4
1 Perform vec_max(A,B)
5678
1764
5778
Compare new max value with original max values using vec_cmpeq
5678
5778
1011 = maxMask
2 AND maxMask with original max ids
10 20 30 40
1 0 1 1
10 0 30 40 = max_ids_diff
3 Store the opposite of maxMask in minMask by NAND maxMask with {1,1,1,1}
maxMask = 1011
minMask = 0100
4 AND minMask with original min ids
50 60 70 80
0 1 0 0
0 60 0 0 = min_ids_diff
5 OR max_ids_diff and min_ids_diff to get final max ids
10 0 30 40
0 60 0 0
10 60 30 40
FINAL MAX
Vec ID 10 60 30 40
Vec Value 5 7 7 8

Figure 4.7: Finding the corresponding IDs after splitting a vector into min and max values (MAX example).

```
vector unsigned char perm_reverse = {12,13,14,15,8,9,10,11,4,5,6,7,0,1,2,3};
vector unsigned char add_index1 = {12,12,12,12,4,4,4,4,4,4,4,4,12,12,12,12};
vector unsigned char basis_index1 = {0,1,2,3,4,5,6,7,4,5,6,7,0,1,2,3};
vector unsigned char toggleOneOfTheEquals = {-1,-1,-1,-1,-1,-1,-1,-1,0,0,0,0,0,0,0,0}; //The -1's
    should be equal to 8 1 bits.

// Find which values are greater than
mask1.vec = (vector unsigned char)vec_cmpgt(vec.vec, vec_perm(vec.vec, vec.vec, perm_reverse));

// Find which values are equal
mask2.vec = (vector unsigned char)vec_cmpeq(vec.vec, vec_perm(vec.vec, vec.vec, perm_reverse));

// Set one of each pair to 0
mask2.vec = vec_and(mask2.vec, toggleOneOfTheEquals);

// OR with the first mask to indicate the value we want to swap
mask1.vec = vec_or(mask1.vec, mask2.vec);

pattern.vec = vec_and(mask1.vec, add_index1);
pattern.vec = vec_add(pattern.vec, basis_index1);

vec.vec = vec_perm(vec.vec, vec.vec, pattern.vec);
```

Listing 4.10: Intravector Sort code snippet

4.2.4 Detecting Changes

With the most complex sorting stage out of the way, there are still some further operations to be performed in order to detect the objects that have moved since the last frame. We can identify the IDs of objects that have changed, by storing the values and IDs array before the sorting process. This is achieved in code via a simple *memcpy* function. We then perform the intervector and intravector sorting and then compare the resulting sorted values array against the original array (before sorting). Exploiting SIMD once again, a comparison of two the value arrays is performed using *vec_cmpeq*, as was used in the previous stage, section 4.2.3. This comparison for equality gives a vector mask detailing which corresponding values were equal and which ones were not. As *vec_cmpeq* returns a mask of *1,1,1,1* if all elements are equal, we need to focus on any zeroes in the mask, as these represent elements that were not equal. Whereas before we would bitwise *AND* with a mask to pick out values we want, here we perform a bitwise *OR* because we desire the elements in the original array that correspond to the zeroes in the mask. We perform the bitwise *OR* between the mask and the IDs array. It is important to note, that we perform the comparison between the value arrays, and the *OR* between the mask of that operation and the IDs array. Though possible to perform the comparison between the original IDs array and the one after the sort process, this can provide unnecessary extra information on IDs that may have swapped, but have the same values, so it is not necessary to take them in to account.

```
void compare_changes(float* values, float* values_copy, int* ids, int* ids_copy)
{
  int i, j, numVec = EP_LIST_SIZE/4;     /* Number of vectors */
  vector signed int ids_vec, ids_copy_vec;
  floatVec oldValues, newValues;
  vector unsigned int comparison_vec;

  for(i = 0; i < numVec; i++)
  {
    // Load in the vectors
    ids_vec = vec_ld(i*sizeof(ids_vec), ids);
    ids_copy_vec = vec_ld(i*sizeof(ids_copy_vec), ids_copy);

    oldValues.vec =  vec_ld(i*sizeof(oldValues.vec), values);
    newValues.vec =  vec_ld(i*sizeof(newValues.vec), values_copy);

    // Compare the old values to the new values and see which are equal
    comparison_vec = vec_cmpeq(oldValues.vec, newValues.vec);

    // Or comparison vec with ids_copy_vec (the sorted one) to get ids that have changed
    ids_vec = vec_or(ids_copy_vec, comparison_vec);

    // Store the result back in ids_vec
    vec_st(ids_vec, i*sizeof(ids_vec), ids);
  }
}
```

Listing 4.11: Comparing and finding IDs of objects that have changed

The code to perform this functionality is presented in listing 4.11. In this code example, the arrays with the "_copy" tag on the end represent the arrays that underwent the sorting stages, "EP_LIST_SIZE" refers to the number of end-points in the scene, also equal to *number of objects x 2* .

4.2.5 Obtaining a Manageable Array

After the detection of IDs that have changed in the array, we are left with an array that contains these IDs, albeit in a slightly dispersed manner. As the comparison function from section 4.2.4, uses SIMD operations and mask vectors, it stores the changes in IDs in groups of four, e.g. a comparison of the vectors *25,30,40,55* and *20, 30, 35, 55* would result in the ID changes array as: *25, 0, 40, 0* or *changed, no change, changed, no change.* As you can see, the IDs that we want in the array are separated by many zeroes. The purpose of this stage is manipulate this *changes_ID* array into something more manageable and amenable to parsing. The first step is to sort the array in reverse order, so that all of the lowest values will be at the end. This reverse sorting is performed by a standard non-SIMD *quicksort* although altered slightly to sort in reverse. This will give us an array where all of the zeroes are at the end. Now when we want to process the array, we can start at the first index of the array and iterate until we find value zero, to save us iterating over the entire array and skipping zeroes as we go. An additional manipulation is performed to remove duplicates in the array. This is necessary because for every ID that represents a minimum end-point of an object's AABB, in the original IDs array, there is a second appearance of this ID to represent the maximum end-point of the same object. When an object moves, both end-points of its AABB will move and these two changes will be detected by the comparison process of section 4.2.4. Basically every ID in the *changes_ID* array occurs twice. Removing the duplicates is performed by a function that operates on a sorted array. As the array is sorted, the remove duplicates function can simply compare neighbouring objects to each other. A step-by-step example of this whole sequence of events was previously shown in section 4.2.1.

4.2.6 Overlap Testing

At this stage we have successfully detected the IDs of the objects that have moved significantly enough in the scene, to warrant retesting their collision status. By isolating only the objects that have changed in the arrays, we have exploited the temporal coherence as the original sweep and prune algorithm does. All that remains is to retest these objects for overlaps. As mentioned in section 4.2.1, we have two choices: we can determine overlaps based on the intermediary IDs in between the minimum and maximum end-points of an object or we can perform a isolated brute force overlap test between these changed objects and all other objects in the scene. The first method was the primary approach implemented in the project, while the second approach was investigated also.

Overlap Test Method 1

This testing method involves iterating over our array of changed IDs, referred to as *changes_ID* in section 4.2.5. We iterate over this array until we reach element zero, as explained previously. We store the element at the current index in the *changed_IDs* array, in a temporary variable called the *target ID*. The first step is to find the array index of this target ID, in the IDs array that underwent the sorting procedure. This search was implemented as a basic linear search, but could be optimised in future via a binary search. Once we find this array index, we break out of the search and this index represents where the minimum endpoint value of the target ID occurs in the IDs array.

It represents the minimum because, the values array is sorted by value and all relevant swaps are applied to the IDs array, during the sorting process. We now are at the index of the minimum endpoint and we want to iterate over the IDs array, starting at this index, until we reach the target ID again. As long as we are iterating and the ID at the current index is not equal to the target ID, we mark this ID as overlapping for the target ID. It must be overlapping because at least one of its end-points is positioned between the minimum and maximum end-points of the target ID. It is important to note that these overlaps are being discovered purely by encountering IDs in a list and there is not floating point comparison operation of bounding box extents taking place. To set the target ID and the current ID as a collision pair, they are sorted and a single hash key to represent the two is generated, as explained in section 4.1.3. This hash key is stored in an array on the SPU called *collision_keys*. After all the relevant collision keys have been generated through this approach, we then send them back to the PPU through a DMA transfer. The collision keys for the pairs of objects that are now colliding along that axis are then looped through and applied to the pair manager, updating the collision status to true. This process is visible in listing 4.12.

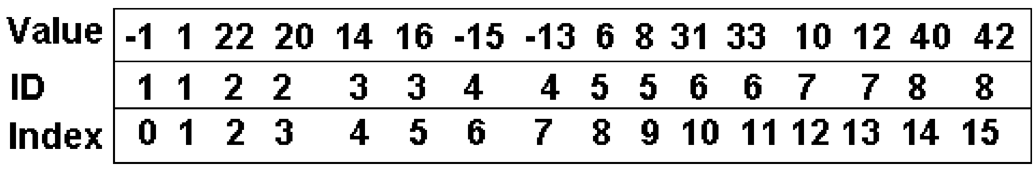
This is an effective method of detecting new collisions. However, it is unable to detect if two objects are no longer in collision. Perhaps a specifically designed narrow-phase algorithm could compensate for this, but that could prove difficult. In order to overcome this problem, on the PPU side, a set of previous collisions can be maintained for each ID in the changed_IDs array. During each frame, the previous collisions of all new IDs in the changed_ID array are set to false. This is a can have a major impact with the regards to an interactive frame rate, as will be discussed in chapter 5.

Overlap Test Method 2

An alternative to the previous overlap test method is to simply perform overlap tests between all of the IDs that have changed and the IDs of every other object in the scene. Then, instead of just generating and storing hash keys for overlapping pairs, we generate and store the keys for both colliding and non-colliding pairs and keep these in two separate arrays: colliding pairs and non-colliding pairs. By taking this approach, we eliminate the need to track a set of previous collisions, like we had to in the previous method. The downside is that a far greater volume of hash keys have to be sent back

```
// Process each element of changes_id array
while(in_ids[i] != -1)
{
  target_id = in_ids[i];

  // Search through the sorted list until we find the target ID
  for(j = 0; j < EP_LIST_SIZE; j++)
  {
    if(in_ids_copy[j] == target_id)
      break;
  }

  // Should have found the target_id, so iterate from j until
  // until we meet the target_id again
  // Skip ahead to the next element
  j++;
  while(in_ids_copy[j] != target_id)
  {
    tempTargetID = target_id;
    tempOtherID = in_ids_copy[j];

    // Make sure ID's are always stored / referenced in the same order
    Sort2(tempTargetID, tempOtherID);

    // Get the hash key
    key = GetHashKey(tempTargetID, tempOtherID);

    collisionKeys[numKeys].aID = tempTargetID;
    collisionKeys[numKeys].bID = tempOtherID;
    collisionKeys[numKeys].hashKey = key;
    numKeys++;

    j++;
  }
  i++;
}
```

Listing 4.12: Determining overlaps based on the intermediary elements between the minimum and maximum endpoints of objects that have moved significantly since the last frame



Value	-1	1	22	20	14	16	-15	-13	6	8	31	33	10	12	40	42
ID	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Figure 4.8: How values, IDs and indices correspond.

to the PPU, which in turn have to all be applied to the pair manager, to update the collision status. Whereas the first approach just sent the keys of objects that were colliding, this approach sends back all of the keys for pairs that are colliding and all of the keys for every other pair that is currently not colliding. To allow us to perform actual overlap testing with the values of bounding boxes, we can pass in the original input arrays for the values and IDs. These two arrays hold the same values and IDs as when they were created and populated on the PPU, before ever being sorted by the SPUs. The initial ID array holds all IDs in a sequential manner e.g., from 1 to the number of objects in the scene (IDs start at 1, not zero). As there are two end-points for every ID, the ID array looks like *1,1,2,2,3,3,4,4,etc.*. On this premise we can devise a look up of the relevant values, based on the current IDs that need to be tested for overlap. To find value corresponding to an ID to be tested, we multiply this target ID by two, then subtract two from it *targetID x 2*, then *targetID - 2*, this is necessary as the IDs start at 1 i.e. the first object created in the scene has *ID = 1*, but *ID = 1* will be positioned at *index = 0* (the first element the array). An example can make this clearer: if we want to find the minimum end-point value of ID 6, we multiply by two:*6 x 2 = 12*, then we subtract two *12 - 2 = 10.* To find the maximum value (the corresponding end-point) we simply add one to the index that we found with the previous operation. A table is presented in figure 4.8 showing how values, IDs and array indices correspond. Performing the previous value look-up example with *ID = 6*, on this table shows how it can work.

4.3 Issues

The complex nature of the Cell's unique architecture presented some issues during implementation. Inexperience with powerful low-level processing functionality such as

SIMD presented some initial teething problems, but these were soon overcome. The nuances of Direct Memory Access (DMA) operations took a little more time to get used to. In particular, the limit of data that can be sent in one DMA transfer is 65536 bytes or $2^1 6$ bytes. To send more than this, DMA list transfers must be used, which essentially queue up multiple transfers to be performed one after another. This was implemented at one stage during the Cell implementation, but was later removed in favour of a different approach. The final technique used for sending data larger than the limit, was required for sending back the collision hash keys generated on the SPUs to the PPU, as explained in section 4.2.6. In order to send more than the allowed amount of data, a chunk based approach was utilised, where data would be accumulated until it was larger than a certain threshold, this chunk of data would then be sent and the process repeated.

The biggest issue encountered apart from the identical values issue that didn't register swaps in ID, was the size constraints of the SPUs local store (LS) which is only 256KB. The LS must contain all of the instructions, the data and stack. In terms of implementation it meant that large arrays of data structures could not be allocated without having to sacrifice some other variables. Inexperience when dealing with the SPUs local store size, lead to the pair manager being implemented in full on the PPU, which in turn lead to the costly update procedure that had to be performed outside of the SPU algorithm code.

Chapter 5

Results and Evaluation

The purpose of this chapter is to present the results on the performance of the two different sweep and prune algorithm implementations. The Cell algorithm was tested on Cell Blade server, consisting of 8 SPUs. The Cell PPU code and SPU code were both compiled with the highest level optimisation flag available, *-O3*, using a *g++* compiler. The compilation flag *-funroll-loops* was used also for the SPU code, which unroll loops in the code, attempting to boost execution speed. The x86 version was run on an AMD Phenom(tm) 9600 QuadCore 2.3GHz with 4GB RAM, to represent a moderately powerful, commercially available, standard desktop PC. The x86 version was compiled in a release build using Microsoft Visual Studio 2008. A set of tests scenarios were discussed in section 3.2.3. The Cell tests use the first overlap testing method explained in section 4.2.6. The second overlap method also described was merely explored, but resulted in large amounts of collision hash keys being transferred from SPU to PPU, it does not factor into the tests. During the initial design phase it was believed that there would simply be one Cell implementation and one x86 implementation to compare. During the implementation phase, some different factors were uncovered that can have important effects on the performance of the Cell implementation. The most important of these factors, is that the Cell implementation requires that the results of the collision tests performed on the SPUs be transferred back to the PPU and applied to the pair manager structure, which manages the collision status of all pairs of objects. This was caused by a structural oversight in the design, due to inexperience with the Cell processor and time constraints as the implementation phase neared the end of its life

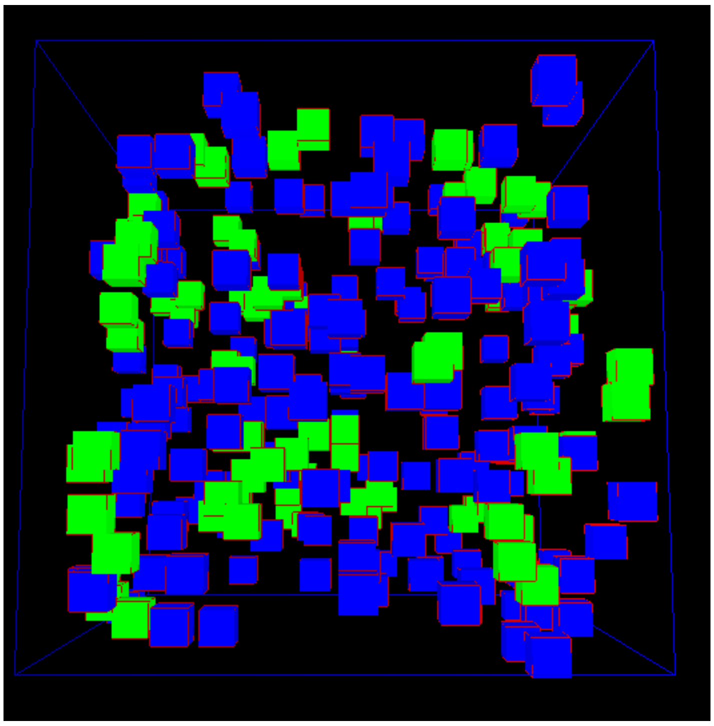


Figure 5.1: x86 implementation. Green highlight represents boxes that are colliding.

cycle. The oversight which requires these updates be applied outside of the overlap tests, could be rectified in the future, but for the purposes of this project, the impact that it has must be detailed. For that reason, when comparing the performance of the algorithm under the strain of some dynamic objects, two different sets of results will be presented: the results of the algorithm as it operates on the SPU only and the results of the SPU algorithm and the associated updates that need to be applied to pair manager on the PPU side. Three sets of results will be presented: the Cell SPU algorithm with updates applied versus the x86 algorithm: performing the initial sort of objects, the Cell SPU only algorithm with no updates versus the x86: on a dynamic scene where 10% of the objects are moving and finally a third set of results which shows the effect on performance of applying the collision test results to the pair manager. Figure 5.1 demonstrates what the x86 algorithm looks like when it is running.

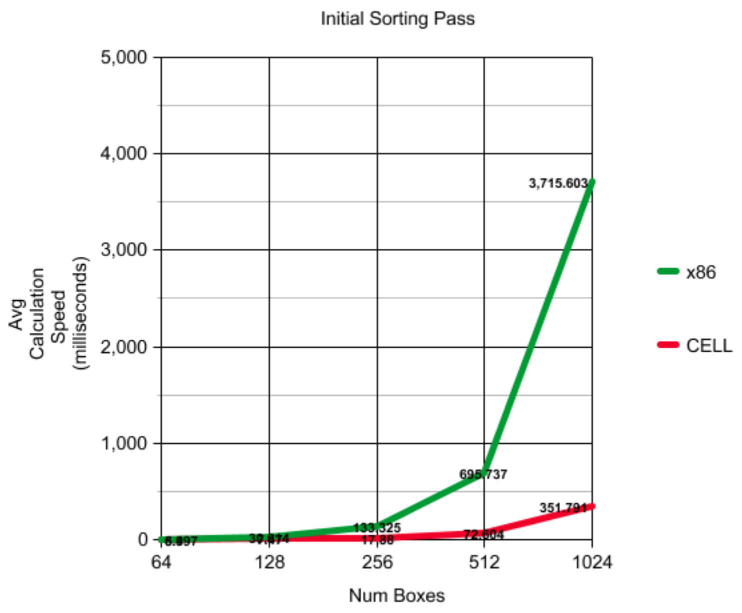


Figure 5.2: Results: Initial Sort

5.1 Initial Sort

The initial sorting pass in figure 5.2 demonstrates the strength of the algorithm the clearest, as it is at this stage that there is the most work to do. With regards to the notion of updating the pair manager, as mentioned in the introduction to this chapter 5, the set of results for the Cell in this test include updating the pair manager. It is evident that in spite of the extra processing time needed to iterate through a list of collision hash keys, returned from the SPUs, the Cell implementation is still far faster at sorting and detecting overlapping pairs than the x86, as the load increases.

5.2 Static Scene

The costs for a static scene shown in figure 5.3, after the initial sort has taken place, highlights the fact that to beat the x86 on a static scene, is very difficult. This is because the x86 insertion sort algorithm has nothing to do, as the lists are already sorted. The lists are sorted for the Cell version also, however, it does have a little more work to do and there is an inherent cost in just setting up and running code on

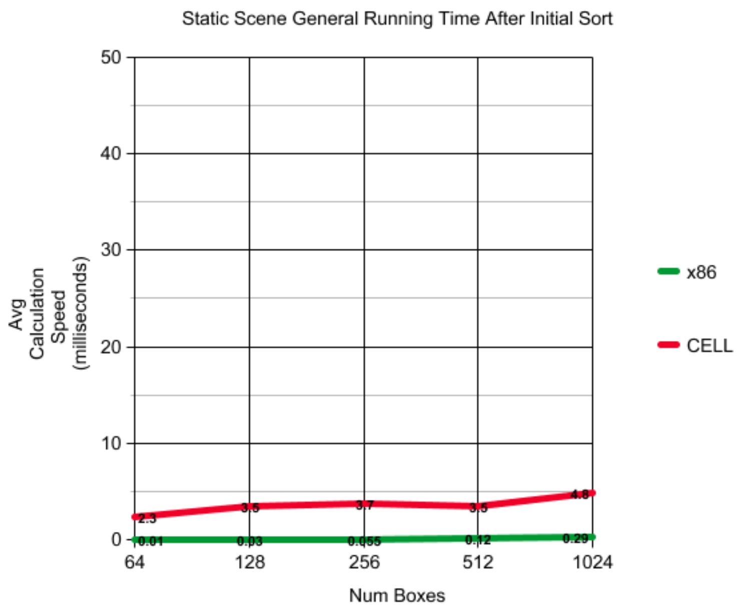


Figure 5.3: Results: Static scene

the SPUs. The Cell algorithm could also be further optimised to ignore steps in its algorithm, based on whether any objects have moved significantly or not.

5.3 Dynamic Scene

Some additional thought was required for the dynamic set of tests, as it was difficult to decide on what exactly should be tested. Sometimes frame rate is considered, but as the Cell server that ran the algorithm had no graphical capabilities, this would not be accurate, as the full frame cost would not be accurate without rendering. The best way to test the algorithm under these conditions was to record the longest execution time spent on the sweep and prune algorithm. To achieve this, all tests were run for 1000 iterations and the longest execution time recorded. The timing code used was cross platform between Windows for the x86 and Linux for the CELL. This ensured a consistent set of tests. Figure 5.4 shows the performance of the Cell algorithm when it is not updating the pair manager on PPU, versus the x86 implementation. It is important to note here, that although the Cell endures the same minimum approximate *3ms* as the static scene results, it again out performs the x86 for sorting performance

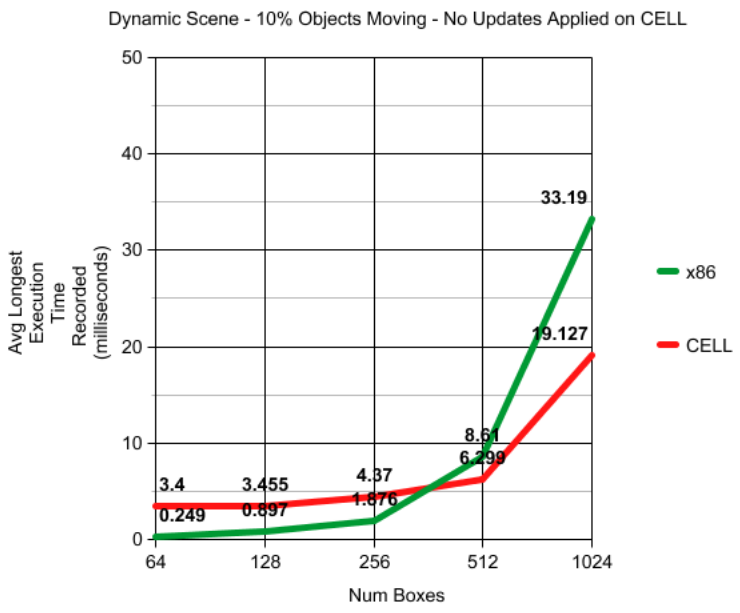


Figure 5.4: Results: Dynamic Scene. Cell results based solely on SPU algorithm

as the number of objects and their AABBs increase. Further tests would follow the same trend, increasing the gap in performance even more so. Also notice that the Cell implementation's longest running time for 64 boxes is quite similar to it's longest running time for 512 boxes. This exhibits the consistency of the algorithm and would result in a steadier frame rate than the x86 version, as the load increases.

Unfortunately, a hindrance of the current Cell implementation framework is that the changes in collision status cannot be applied as they are detected. This is the strength of x86's insertion sort process, where the pair manager gets updated during an overlap test. As explained in section 4.2.6 a previous set of collision IDs must be maintained in order to change the collision status of objects that are no longer colliding. This hindrance, as well as the updates that must be applied to the pair manager for objects that are colliding, increases execution time. The effect of this is shown in figure 5.5 as all three versions tested on the dynamic scene are presented together: the Cell SPU algorithm only, the Cell SPU algorithm with applying updates and maintaining previous collisions, and the x86 implementation. Notice that even with these extra costs the Cell can perform admirably up until 512 boxes, where the execution will limit the ability to produce an interactive frame rate. The cost of updating the pair manager

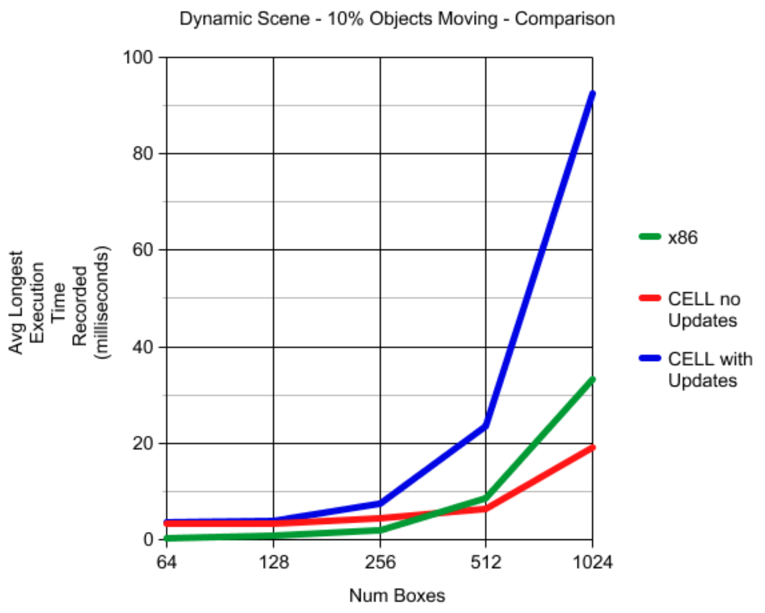


Figure 5.5: Results: Dynamic Scene. All three sets of results presented together

and maintaining previous collisions increases greatly as the number of objects in the scene gets larger. It would be possible to perform these updates as part of the SPU algorithm, but as previously mentioned, initial inexperience with Cell programming lead to this slightly flawed design.

Chapter 6

Conclusion and Future Work

The aim of this chapter is to assess the project. The achievement of the original goals set for the project, will be discussed. Some notes on future work and possible avenues of investigation will be presented also.

6.1 Conclusion

There were several high level goals set for this project, during its conception. These goal were as follows:

- To investigate and explain the background of collision detection and broadphase collision detection in particular, to discuss related work and provide an overview of the Cell architecture.
- To develop a simple rigid body based physics simulation environment that will provide the target for the broadphase algorithm. This implementation will be the same for both x86 and Cell.
- To develop a standard broadphase algorithm for an x86 processor.
- To identify opportunities for parallelism in the broadphase algorithm.
- To implement a parallelised broadphase algorithm that exhibits SIMD processing to exploit the capabilities of the Cell processor.

- To compare and contrast the two implementations.

The first step was to investigate the area of collision detection and provide an in-depth account of the various techniques employed today. A thorough analysis was conducted that focussed primarily on broadphase techniques, explaining core concepts from bounding volumes to the different types of broadphase algorithm. Other than the many works referenced with regard to these broadphase techniques, a separate section explored the most recent work in the field. Some of the more relevant papers were presented that have examined the idea of implementing broadphase collision detection on parallel architectures. It was also imperative that some background information be provided with regard to the Cell Processor. An overview of the architecture was presented and also some recent work that has been conducted with the Cell, examined.

A key stage in the project was to develop a rigid body based simulation framework, not as a contribution to the field, but simply to allow a broadphase algorithm to perform. This framework provided the raw materials for the sweep and prune algorithm. The design chapter provided a description of this system. The design chapter also discussed some of the important decisions made for the project, such as the choice of broadphase algorithm for the project. The reasons behind these decisions were detailed also. The design of the experiment to evaluate the performance of the algorithms, was also thoroughly described.

The implementation chapter dedicated much attention to the x86 version of the sweep and prune algorithm. This proved important as it lay the building blocks for the Cell version also. A thorough account of both implementations was carried out, with the use of diagrams and code snippets. The more detailed low-level implementation specifications were also explained, as they were integral to the unique algorithm that was developed for the Cell implementation. Examples of SIMD functions and their usage were provided to aid in the explanations. The final Cell implementation, to the best of its potential, exploited the Cell architecture and its processing capabilities.

Finally the experiment tests were run on their target platforms. Each test was carefully designed to highlight different aspects of the algorithms and explain the results in an open manner. The primary goal of the project was to investigate the applicability of the sweep and prune algorithm to the Cell. After running the tests, this could be addressed. The results showed that the Cell implementation could sort a large set effectively, especially during the initial sorting test. The problems of the Cell implementation were not glossed over. A combination of navety and lack of experience with the Cell processor led to some design decisions that in hindsight, could have been improved. The need for the Cell based implementation to update the collision status of objects separately to the algorithm, proved costly. The correlation between the Cell and x86 version of the algorithm was explained. The difference between the results of the Cell SPU only algorithm and Cell SPU algorithm coupled with the update costs, was clearly evident. The notion that broadphase techniques are not easily amenable to parallelisation, is not without some merit. However, the ideas presented in this dissertation provide results that show that the sweep and prune broadphase algorithm is certainly applicable to a parallel architecture, the Cell processor. With careful design consideration and a slight restructuring of the algorithm, the update process could be incorporated into the SPU algorithm and the costly issue could be eradicated. As the true penalty in execution time only became fully evident during the testing stage, it was not possible to implement the restructuring, for this project. If this were to be achieved however, the potential of the Cell algorithm shown in some tests could be realised in full and true real-time performance of the Cell sweep and prune algorithm would become a reality.

6.2 Future Work

Throughout this body of work, many sections alluded to potential for future work. As explained in the previous section, the most pertinent of these would be the restructuring of the Cell sweep and prune algorithm, to incorporate updating collision status as part of the algorithm that runs on the SPUs. The second overlap test method explored in section 4.2.6, may hold the key to performing the collision tests and updating the collision status together, as this method performs a brute force check of all objects that have moved significantly since the last frame, checking if they overlap or not, whereas the first method, the one used for the tests, can only detect new collisions.

The Cell algorithm itself could be optimised further as mentioned in the implementation chapter. While it performs some of its execution steps using SIMD, other steps do not exploit the SIMD capabilities of the Cell, namely the reverse sorting and duplicate removal steps as well as some other array searching functionality.

An interesting avenue for future work could be the parallelisation the axis aligned bounding box generation. This was identified early as part of the reason for choosing the sweep and prune algorithm, as it only works on these AABBs. The focus of the project then became the sweep and prune algorithm itself. However, there is potential for parallelism with these AABBs because, like the sweep and prune algorithm itself, the three cardinal axes can be dealt with separately, in parallel. The hope is to research this idea in more detail in the future.

Appendix

...

Bibliography

[1] S. Oh, H. Kim, and K. Wohn., "Collision handling for interactive garment simulation," *Proceedings of VSMM*, 2002.

[2] I. Rudomin and J. Castillo, "Realtime clothing: Geometry and physics," *WSCG 2002 Posters, Czech Republic*, p. 4548, 2002.

[3] C. Koh and Z. Huang, "A simple physics model to animate human hair modeled in 2d strips in real time," *Proceedings of Computer Animation and Simulation*, 2001.

[4] B. Honzik and Y. Hamam, "Obstacle avoidance for non-point mobile robots," *Proceedings of 3rd IMACS Symposium on Mathematical Modeling*, p. 887890, 2000.

[5] S. Cotin, H. Delingette, and N. Ayache, "Real-time elastic deformations of soft tissues for surgery simulation," *IEEE Transactions on Visualization and Computer Graphics, volume 5*, 1998.

[6] C. Ericson, *Real-Time Collision Detection. The Morgan Kaufmann Series in Interactive 3D Technology.* Morgan and Kaufmann, 2005.

[7] G. Bergen, "A fast and robust gjk implementation for collision detection of convex objects," *Journal of Graphics Tools, volume 4*, 1999.

[8] D. Baraff, *Dynamic Simulation of Non-Penetrating Rigid Bodies.* PhD thesis, Cornell University, Ithaca, New York, USA, 1992.

[9] D. Eberly, "Dynamic collision detection using oriented bounding boxes," *Technical Report, Magic Software*, 2002.

[10] E. Larsen, E. Gottschalk, M. Lin, and D. Monacha, "Fast proximity queries with swept sphere volumes," *Technical Report, Dept. of Computer Science, University of North Carolina*, 1999.

[11] J. Klosowski, M. Held, J. Mitchell, H. Sowizral, and K. Zikan, "Efficient collision detection using bounding volume hierarchies of k-dops," *IEEE Transactions onVisualization and Computer Graphics, vol. 4, no. 1*, p. 2136, 1998.

[12] D. Eberly, "Intersection of cylinders," *Technical Report, Magic Software*, 2000.

[13] M. Held, "Erit: A collection of efficient and reliable intersection tests," *Journal of Graphics Tools, vol. 2, no. 4*, p. 2544, 1997.

[14] D. Eberly, "Intersection of a sphere and a cone," *Technical Report, Magic Software*, 2002.

[15] S. Krishnan, A. Patteka, M. Lin, and D. Monacha, "Spherical shell: A higher order bounding volume for fast proximity queries," *Proceedings of 3rd International Workshop on Algorithmic Foundations of Robotics*, p. 177190, 1998.

[16] L. Guibas, A. Nguyen, and L. Zhang, "Zonotopes as bounding volumes," *Proceedings of 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2003.

[17] R. de Sousa Rocha and M. A. F. Rodrigues, "An evaluation of a collision handling system using spheretrees for plausible rigid body animation," *Proceedings of the ACM symposium on Applied computing*, 2008.

[18] T. Kay and J. Kajiya, "Ray tracing complex scenes," *Computer Graphics (SIGGRAPH 1986 Proceedings), vol. 20, no. 4*, p. 269278, 1986.

[19] P. Hubbard, "Approximating polyhedra with spheres for time-critical collision detection," *ACM Transactions on Graphics (TOG) archive, vol. 15 , issue 3*, pp. 179 – 210, 1996.

[20] B. Mirtich, "Efficient algorithms for two-phase collision detection," *Practical Motion Planning in Robotics: Current Approaches and Future Directions, John Wiley & Sons*, p. 203223, 1998.

[21] V. Havran and F. Sixta, "Comparison of hierarchical grids," *Ray Tracing News, vol. 12, no. 1*, 1999.

[22] J. Bentley, "Multidimensional binary searchtrees used for associative searching," *Communications of the ACM, vol. 18, no. 9*, p. 509517, 1975.

[23] J. D. Cohen, M. C. Lin, D. Manocha, and M. K. Ponamgi, "I-collide: An interactive and exact collision detection system for large-scale environments," *In Proceedings of the 1995 Symposium on Interactive 3D Graphics*, p. 189196, 1995.

[24] P. Terdiman, "Sweep and prune document," *http://www.codercorner.com/SAP.pdf*, 2007.

[25] T.Hudson, M. Lin, J. Cohen, S. Gottschalk, and D. Manocha, "V-collide: accelerated collision detection for vrml," *Proceedings of the second symposium on Virtual reality modeling language*, pp. 117–ff, 1997.

[26] P. Terdiman, "Opcode and memory-optimized bounding-volume hierarchies opcode," *http://www.codercorner.com/Opcode.htm, http://www.codercorner.com/Opcode.pdf.*

[27] Sony, "Cell programming primer," *http://www.kernel.org/pub/linux/kernel/people/geoff/cell/ linux-docs/ps3-linux-docs-08.06.09/CellProgrammingPrimer.html*, 2008.

[28] M. Scarpino, *Programming the Cell Processor: For Games, Graphics, and Computation.* Prentice Hall, 2008.

[29] N. K. Govindaraju, S. Redon, M. C. Lin, and D. Manocha, "Cullide: Interactive collision detection between complex models in large environments using graphics hardware," *In Proceedings of the 2003 ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, pp. 25–32, 2003.

[30] R. Sathe and A.Lake, "Rigid body collision detection on the gpu," *Proceedings of the second symposium on Virtual reality modeling language*, 2006.

[31] O. Lawlor, *A grid-based parallel collision detection algorithm.* PhD thesis, University of Illinois at Urbana-Champaign, 2001.

[32] O. Lawlor and L. Kale, "A voxel-based parallel collision detection algorithm," *Proceedings of the 16th international conference on Supercomputing*, 2002.

[33] M. Figueiredo and T. Fernando, "An efficient parallel collision detection algorithm for virtual prototype environments," *ICPADS'04*, 2004.

[34] I. Grinberg and Y. Wiseman, "Scalable parallel collision detection simulation," *Proc. of Signal and Image Processing*, 2007.

[35] D. Kim, J. Heo, and S. Yoon, "Pccd: Parallel continuous collision detection," *http://sglab.kaist.ac.kr/PCCD/, Korea Advanced Institute of Science and Technology, South Korea*, 2008.

[36] M. Tang, D. Manocha, and R. Tong, "Mccd: Multi-core collision detection between deformable models," *Graphical Models 72*, pp. 7–23, 2010.

[37] H. Sammet and R. Webber, "Hierarchical data structures and algorithms for computer graphics," *IEEE Computer Graphics and Applications, volume 4*, pp. 46–68, 1998.

[38] V. G. Q. Avril and B. Arnaldi, "A broad phase collision detection algorithm adapted to multi-cores architectures," *Proceedings of Virtual Reality International Conference*, 2010.

[39] M. C. Tran, "Porting bullet physics library to cell be with libspe 2," *University of Stuttgart*, 2007.

[40] N. Hjelte, *Smoothed Particle Hydrodynamics on the Cell Broadband Engine.* PhD thesis, Umea University, Sweden, 2006.

[41] P. Terdiman, "Sweep and prune benchmark," *http://www.codercorner.com/SweepAndPrune.htm.*