Collision detection on mobile devices GPU

Simon Saffer & Johan Jungbeck
Lunds Tekniska Hogskola

September 27, 2010

Abstract

Using the powerful processors of the GPU for physics calculations is already common on stationary computers. Mobile devices might also benefit from this. In this project we will try to see how well the GPU can be used for collision detection on a handheld device, such as a mobile phone, and go through some of the common problems that exist when accessing the GPU on a mobile device. To do this we will compare a CPU implementation with a GPU implementation of a collision detection algorithm. There are many different collision detection algorithms, but we have chosen to use algorithms that are suitable for using in combination with rigid body physics.

Contents

1 Introduction

1.1 Background

As software becomes more and more complex on mobile devices, the need for hardware support is increasing. Most of the new devices have support for graphic hardware acceleration with GPU. Since modern 3D-games often require physics the support of hardware accelerated physics is needed. A big part of rigid body physics is to determine if objects collide or not. This is commonly known as collision detection. Collision detection is mostly vector based calculations, so it could be beneficial to do these on the GPU. Also GPUs consist of several processors that make it possible to do parallel calculations, another advantage against the CPU. On stationary computers, GPGPU based languages that have hardware support for GPGPU already exist to do this such as CUDA, OpenCL and PhysX. Benchmark test have been done that have shown that collision detection and physics calculations, such as rigid body physics and more, can be significantly accelerated using API's such as PhysX (see section 1.9.2). Mobile devices still lack support for such API's but even before hardware GPGPU support existed on stationary computers, GPGPU has been used to successfully speed up calculations. Since mobile devices also lack specific GPGPU hardware support this work (where GPGPU is done without API's) is very relevant our thesis project. In section 1.4 we briefly account for what they have done and in section 1.5 we will explain how this relates to our work. GPU accelerated collision detection and physics calculations have not been done previously on mobile platforms and in this thesis we therefore investigate whether OpenGL ES 2.0 can be used to implement a GPGPU solution for mobile devices with support for programmable shaders. Mobile platforms have limitations that stationary computers do not, such as lower memory bandwidth, less cache, less powerful graphic cards and limitations in programming languages and API's. OpenGL ES 2.0 is one API that is a lot more limited for mobile devices than OpenGL for stationary computers. The lack of GPGPU API support forces us to resort to traditional GPGPU methods which are not as fast as GPGPU with API's but can still be significally faster than the CPU (see [14], [3]). However it cannot be concluded without investigation that just because it can be done on stationary computers it can also be done on mobile platforms. The difference in hardware might or might not be too big for mobile GPGPU to be effective. This is why we decided to investigate the matter in this thesis.

1.2 Problem

Our purpose with this thesis work has been to investigate whether collision detection can be accelerated by performing them on the GPU (Graphics Processing Unit) instead of on the CPU (Central Processing Unit). At first we wanted to find out if it was at all was possible to perform these calculations on the GPU and then, to see if it in fact would give reduced execution time. Since the collisions will be used with physics, it is important that the output data from the collisions can be used for physics. We have therefore implemented a CPU based physics engine, however the performance of the physics will not be measured.

1.3 Terminology

BV	Bounding Volume
CPU	Central Processing Unit
GPU	Graphics Processing Unit
GFLOPS	Giga FLoating point Operations Per Second
OpenGL ES	Open Graphics Library for Embedded Systems
GPGPU	General Purpose computing on Graphics Processing Units

1.4 Related Work

James Anthony Bird in his thesis at the University of Abertay Dundee [3], used GPU on a stationary computer to do collision detection. Bird implemented collision detection between triangles and rays to detect intersection between solid objects and particles but claims that his results can be applied to other types of collision detection. At the beginning of his work he implements a simple collision detection algorithm for CPU, vertex shader and fragment shader and concludes that the fragment shader executes the fastest. He then goes on to implement a couple of different acceleration structures that are available for modern graphics cards for stationary computers. He compares the execution time when using no acceleration structure, uniform grid, spatial median KD-Tree and SAH KD-Tree and shows that the SAH KD-Tree executes faster especially when large numbers of triangles are involved. He also looks into an algorithm to speed up the process of reading back the results of the collision detection. This was shown to be effective only when there are few actual collisions in the scene.

Dave Knott and Dinesh K. Pai developed an alternative method to the usual geometry algorithm in their thesis CInDeR - Collision and Interface Detection in Real-time using graphics hardware[10]. CInDeR is an image space algorithm that uses the fact that rendering to the depth buffer and accumulation buffer is done by the graphics hardware. The algorithm uses rays casted from a viewport towards a 3D objects polygon adding 1 for each front facing polygon it intersect and subtracting 1 for each back facing polygon it intersect. If the ray intersects a polygon from another 3D object the ray stops. If the result of the count is non-zero the object is intersected. This is done in a 3 -pass rendering algorithm per viewport. To use this method for physics is a bit tricky since you need to determine the collision points (and objects if there are many objects that collide). There are tricks to overcome this by giving each polygon a unique color for the collision point detection and rendering the polygons where the accumulation buffer is non-zero. Then you can look up the color to see what polygon collided. This adds some more rendering pass making the algorithm to a 7 -pass algorithm.

Markus Malmsten and Simon Klason at Lund Institute of Technology [12] tried Dave Knott's and Dinesh K. Pai's algorithm to do real-time collision detection but did not give any quantitative results. They also added Cg (a shading language, much like GLSL) code to avoid Z-fighting that can occur.

There is also a GPGPU community [5] that tries to gather information about GPGPU from many different sources. Here one can find some lectures on gen-

eral GPGPU independent of what the specific problem domain is. Dominik Göddeke at the University of Dortmund has done a lot of GPGPU work and has some very good tutorials (see [4]) on general GPGPU principles. We present some basic GPGPU theory in section 3.

1.5 Approach

To investigate if collision detection could be done effectively on mobile devices we need to base our work on previous results. For instance we know from lectures from GPGPU.org and from Dominik Göddekes how to try to structure the problem in a way conducive to calculating on the GPU. We also conclude from James Anthony Birds [3] and Dominik Göddekes work that the algorithms should be implemented in the fragment shader and not in the vertex shader. Bird was able to greatly accelerate his collision detection by using tree-structures on the GPU. However this is not possible on the current mobile GPU's and therefore this method cannot be applied to our work. The only way to implement some kind of tree structure would be to have the tree structure on the CPU and then send data to the GPU and consequently retrieve it for each pair of nodes that is tested. As explained in section 3.1 this is very inefficient when doing calculations on the GPU. Our work will consist of implementing our collision detection algorithms and then measuring execution times for CPU and GPU implementations of these algorithms. A big part of our work is figuring out how to be able to do these calculations in OpenGL ES 2.0 which imposes many limitations on what can be done.

There are several possible factors that might influence the performance when doing the collision detection on the GPU that aren't factors when computing on the CPU.

- Hardware properties - Is there any overload limit where the calculation speed is reduced for large amounts of data.
- Memory read/write per calculation - Arithmetic is faster on GPU but writing data to/from GPU memory takes time.
- Whether all data has to be sent to the GPU for each frame or if some things can remain on the GPU between frames.
- Programming limitations due to the shading language.

In order to be able to measure if our chosen algorithms for collision detection can be accelerated by calculating on the GPU we need two different implementations of the collision detection where the same techniques and algorithms are implemented, one where the collision detection is done on the CPU and the other where it's done on the GPU. To test how the hardware properties and memory read/write time affect performance we plan to run the collision detection with varying numbers of objects in the simulation. By studying performance with large amounts of objects we should be able to get an idea of if there is any overload where the hardware is no longer able to work as effectively. By studying performance with small amounts of objects we should be able to get an idea of

how big a factor the read/write time to/from the graphics memory is. We assume that there is some limit where the time loss caused by writing and reading to and from the graphics memory is compensated by the calculation speed of the GPU and passed this limit is where it is profitable to do collision detection on the GPU. One thing that would speed the GPU calculations up considerably, if it is possible, is to keep some data that won't change on the GPU between frames. So we also intend to investigate if there is any way of accomplishing this. This and the shading language limitations are not things to be measured but rather things that we will try to find solutions to while implementing.
To answer the question whether doing collision detection on the GPU is profitable or not we will compare execution time for the CPU and GPU broad phase implementations and the CPU and GPU narrow phase implementations separately to see if any of the parts works better than the other. See section 2 for a description of the broad phase and the narrow phase of collision detection.

The remainder of this report is organized as follows.
Section 2 describes the algorithms to be used for collision detection.
Section 3 contains an introduction to GPGPU theory, how collision detection can be implemented on the GPU and some technical issues when doing GPGPU with OpenGL ES 2.0.
Section 4 presents the results of the comparison of the CPU and GPU implementations.
Section 5 discusses the conclusions drawn from the measurements.

1.6 Work division

As the thesis work should be done individually but more than one person can work on the same project we will now give an account of how the work was divided between us.

1.6.1 Literature studies

We both studied previous work in the area individually but frequently shared our findings with the other.

1.6.2 Programming work

The implementations of the broad phase, both for the GPU and the CPU, was done by Simon while the implementation of the narrow phase, both for the GPU and the CPU, was done by Johan. To make it easier to verify correct behaviour for the collision detection Johan implemented a simple rendering engine and Simon implemented simple collision response on the CPU.

1.6.3 Introduction

Johan wrote the sections Background, Platforms and Languages, Graphics, CUDA and which collision detection algorithm should be used? Simon wrote Problem, Related Work, Approach, Running on the CPU vs. Running on the GPU and PhysX. Both contributed to the section Terminology.

1.6.4 Collision Handling

Johan wrote the section about collision handling.

1.6.5 GPGPU

Simon wrote the section about GPGPU except for the section Collision handling on the GPU which was written by Johan.

1.6.6 Results

This section was written by both.

1.6.7 Conclusions

This section was written by both.

1.7 Platforms and Languages

There exists a large variety of mobile platforms, all of them are not compatible with each others. The code cannot be compatible with all mobile platforms. For the thesis project we have developed the code for iPhone 3GS. iPhone 3GS supports OpenGL ES 2.0 and OpenGL ES SL 1.0 so they are used for graphics and shading programming. The code will most likely be compatible with all mobile platforms newer than the iPhone 3GS. Different implementations of OpenGL ES SL 1.0 use different maximum sizes of shading programs or numbers of allowed variables but we have only tested the code on the iPhone. A lot of development for iPhone requires C/C++ and objective C, so for the implementation these languages have been used.

1.8 Outline/Scope

The purpose of this thesis project was to investigate whether doing collision detection on the GPU could enhance the overall performance of a mobile device game engine. There is a near infinite amount of things that can be done to achieve increased realism in graphics or to optimize the performance or memory usage of a game engine, but in this work these things have not been the priority. The only reason that we have implemented graphics is to see that the collision detection works right.

1.8.1 Running on the CPU vs. Running on the GPU

In the work of implementing collision detection on the GPU there are many design choices that have to be made. Many times the question has arisen what should be calculated on the CPU and what should be calculated on the GPU. Because the purpose of this work is to compare the performance of a GPU-implemented collision detection with CPU-implemented collision detection, we have tried to do as much of the calculations as possible on the GPU. However when there was an obvious gain in calculating something on the CPU we have done so. For instance there are things that can be calculated once on the CPU and then sent to the shader program instead of having the shader program calculate it once for each object in the simulation.

1.8.2 Graphics

Our graphics engine implements per vertex shading with only one directional light source. The graphics engine is also used to check that the result of the collision detection is correct. The calculation of the light follows a simplified phong model [2] where the specular component is not taken into account, also there's no summation since there's only one light source:

$$\text{Pixel color} = M_{amb} \cdot L_{amb} + M_{dif} \cdot (N \bullet D_{light}) L_{dif}$$

Where M_{amb} is the ambient component of the material and M_{dif} is the diffuse component of the material. N is the normal of the surface being shaded. L_{amb}, L_{dif} are the ambient and diffuse components of the light. D_{light} is the direction from the light source to the surface.

1.9 GPU API's

1.9.1 CUDA

CUDA is an acronym for Compute Unified Device Architecture. It is a parallel computing architecture developed by Nvidia. It is a general GPGPU language and it has hardware support GPGPU which gives several benefits. Here is a list of some of these advantages taken from [20].

- Shared memory - CUDA exposes a fast shared memory region (16KB in size) that can be shared amongst threads. This can be used as a user-managed cache, enabling higher bandwidth than is possible using texture lookups.
- Faster downloads and read backs to and from the GPU
- Full support for integer and bitwise operations, including integer texture lookups.

The improved read/write speed means that one of the big GPGPU drawbacks (see section 3.1) is overcome and makes it possible to use CUDA even when the amount of reads and writes is relatively high compared to arithmetic operations. Bullet which is a very popular physics library uses CUDA to speed up calculations [19]. CUDA is not a physics API but can be used for any GPGPU calculations [20]. CUDA code is written in 'C for CUDA' and then compiled to be used on a Nvidia graphics cards (G8X and greater). CUDA can speed up calculations greatly [8] but to this date, no cell phones support CUDA, and only a few mobile devices [22].

1.9.2 PhysX

PhysX is a game physics API provided by Nvidia. It has support for rigid body dynamics, character control, ray-cast and articulated vehicle dynamics, volumetric fluid creation and simulation, cloth and clothing authoring and playback, soft bodies and many more aspects of game physics. PhysX has support for both running the physics on the CPU and on the GPU. When running on the GPU a clear performance boost is achieved. In a benchmark test done by NinjaLane [15] with both modern CPU and GPU hardware running different

games that use the PhysX API it was shown that the GPU achieved superior frame rates. Comparing an Intel Core i7 920 @ 2.8 GHz with a GeForce GTX 285 in SLI mode to do graphics and PhysX work, for several different games, the GPU produced at least twice as high frame rates and at most 13 times higher frame rates. PhysX supports hardware acceleration both by PhysX PPU (a chip dedicated to physics calculations) or a CUDA-enabled GeForce GPU. Mobile devices often lack both so few, if any, mobile devices supports PhysX.

1.10 Which collision detection algorithm should be used?

1.10.1 Image space algorithms

There exist algorithms that detect collisions in the image space. One of these is the CInDeR [10] as mentioned in section1.4. This method make use of virtual ray casting from a viewport for each object in a scene, and uses a accumulation buffer to count the number of times a ray has intersected polygons. If the result is more than zero, then there's a collision along the ray. This works very similar to shadow volume algorithms. Since the algorithm make uses of the image plane and accumulation buffer, the work is done on the GPU.

The good points with this algorithm are that it is linear to the number of objects and it can handle non-convex polyhedrons. The drawbacks are that it is a 3 pass algorithm (4 if used with physics) per object in the scene and it makes heavy use of reading from framebuffers, a very slow operation on graphic cards. Another drawback is that the precision is dependent on the resolution of the viewport since the calculation is done in the image space. It also suffers from self interference due z-fighting that has to be solved [12]. Since mobile devices has a more limited graphic hardware and OpenGL ES 2.0 does not support accumulation buffer we decided to not use this approach for the project.

1.10.2 Geometry algorithms

With geometry algorithm we mean algorithm that uses the data about the objects, such as position to calculate if a collision has occurred. This is a very common approach that is often implemented in today's physics engines. This approach is often a $N(O^2)$ problem, where each object has to be checked against each other object. However it is common to divide the algorithm into two phases to make it faster: one broad phase that culls out objects that are far away, and one narrow that checks the remaining objects if they collide.

The good points are that it is a robust algorithm that is very exact. The precision depends only on the precision of the data variables. It is also easy to modify and to make use of the strength of the GPU. The bad points are that it has to upload a lot of data to the GPU, but it is faster than reading data from the GPU. Also it cannot handle non convex polyhedrons (though a normal approach is to divide a non-convex polyhedron into several convex polyhedrons). Based on this and the large amount of information there is about this algorithm, we have chosen to use this approach.

2 Collision Handling

Collision handling is the part of a physics simulation that determines when any of the objects in the simulation have collided, and then based on the velocities, densities and masses of the colliding objects, calculates the impulses that are to be applied to the object as a result of the collision. To do both quick and accurate collision detection, it can be divided into two parts which are often called the broad phase and the narrow phase. The broad phase makes some approximations and quickly discards all the objects in the simulation that are not even close to colliding. The broad phase then passes all the objects that might be colliding on to the narrow phase. The narrow phase then determines exactly which objects collide and which don't and for each collision it calculates an exact point of collision and a collision normal. Collision detection and collision response are explained further in the following sections. How the collision handling fits into the physics simulation can be seen in appendix A.2

2.1 Collision Detection: Broad Phase

In the broad phase we use simplifications of the shapes in the simulation (like boxes or spheres) that allow us to quickly discard objects that are not close. These shapes are often referred to as bounding volumes or BV [18, p.437-440]. We are then left with some objects that may or may not collide. These objects are then passed on to the "narrow phase" which will determine which of these objects collide.

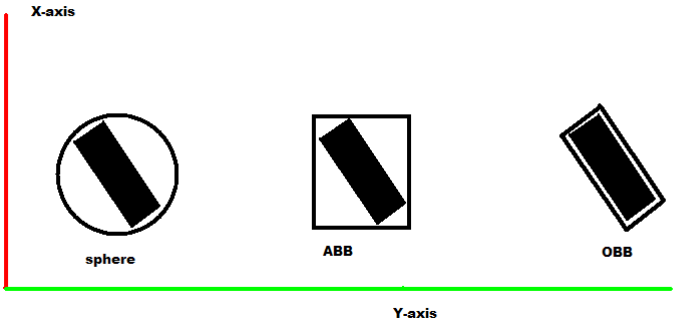

Figure 1: To the left a sphere, in the middle an AABB, to the right an OBB

There are a couple of common BV that are often used for broad phase for their simplicity, see figure 1. The simplest bounding volume is the sphere. To check weather two spheres has collided, you simple add the radius of the spheres and the do a check to see if the distance between their centers is less than the sum of the radius. If it is, these two objects might collide and are checked in the narrow phase. The test is very simple for spheres but the bounding volume tends to be much larger than the actual object it contains. This gives the narrow phase more objects to check than necessary. The axis aligned box (ABB) is a box

that, like the name suggest, is fixed with the world coordinate axis. It means that if the object rotates, the box has to be recalculated. To check weather two ABB's are colliding is slightly more complicated that for a sphere and therefore slower. First the ABB must be projected into 1D space, for each axis. If the boxes is overlapping for every projected axis, they collide and are sent to the narrow phase. ABB tends to fit the object better and gives less false collisions to narrow phase than a sphere. However, some object still get a quite bad fit, as an example a long thin object that has a 45 degrees angle in the x- and z-axis. To get an even better fit one can use the object aligned box (OBB). It has the same orientation as the object, so if the object rotates the OBB will also be rotated. An OBB volume doesn't need to be recalculated like the ABB, but must instead be calculated only once. It is not as easy to have a general formula for calculating a good OBB. There are a few algorithms like calculating the covariance matrix of the position of the vertices and use the eigenvectors as axis [18, p.447-448] , but this is not in the scope of the project. Instead the user must pass a pre-calculated OBB for the mesh. Checking whether two OBB's are colliding is quite complicated. In this thesis project we use the separating axis theorem to determine if two OBB collide or not [1]. The separating axis theorem checks, just as the name suggest if there's an axis that can separate the two OBB, see figure 2. 15 axis are checked. First the three base vectors of one of the object, then the three base vectors of the other object. After this, a combination of each axis of the objects must be checked, for another nine tests. Thus in total 15 tests. If none of these axes separate the BV's then the BV's have collided, but if any of the axes separate the BV's then the BV's do not intersect each other.

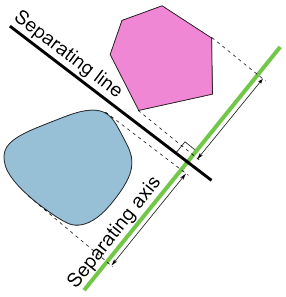

Figure 2: Illustration of the Separating Axis Theorem, taken from Wikipedia

2.2 Collision Detection: Narrow Phase

As said before, the narrow phase determines which objects collide and which don't. The ones that do are then passed on to the collision response part of the collision handling.

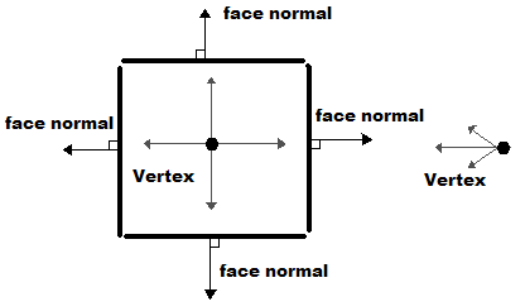

Figure 3: The vertex to the left is inside of the faces since the directions to each of the faces are same as the face normals. The vertex to the right is outside, since only 3 of the face normals are in the same direction.

To determine if a vertex is inside of a convex polygon (a vertex intersection test), the dot product of the vector from the vertex to a face of the polygon and the normal of the face is calculated. If the dot product is positive, (i.e. vectors have the same direction) then the polygon is behind the face. See figure 3. If all the dot products of the vertex and all the face normals are positive, the vertex must be inside of the polygon. This requires two things. First the polygon must be concealing (having faces connecting in all directions) and secondary, the polygon must be convex. If the vertex is in front (having negative dot product) of any of the faces, then the objects are not intersecting. In most cases when two objects do not intersect, not all of the faces need to be tested. (however, in our implementation it is not possible to quit the test until all vertices has been checked, see section 3.6.2). This test is sufficient for the project, but actually two more tests need to be made. It is possible that only an edge (a line between two vertices) is intersecting, and that two identically objects are in exactly the same space. These two tests are normally implemented after the vertex intersection test, but for most narrow phases, only the vertex intersection tests is required [18, p.453-455].

2.3 Collision Response

The narrow phase determines exactly what objects collide and also determines several important variables that are needed for collision response. The collision response is out of the scope for this report but is described in appendix A.3.

3 GPGPU

GPGPU is when the GPU is used for other types of calculations than calculations for graphics. The reason that GPGPU is interesting is that GPU's have many processors and therefore they can perform many calculations simultaneously. This means that they can do a lot more calculations per second than a CPU as illustrated in Figure 4. Thus, if a problem can be formulated in a certain way, suitable for GPGPU, the calculations can be done a lot faster on a GPU than on a CPU. But not every problem is suitable for calculation on the GPU, and although the GPU is programmable to a certain extent it is a lot less flexible in what you can do and how to do it. Also it's very hard to make any generalizations and abstractions when programming for the GPU. What kind of problems are suitable for GPGPU will be explained later. Collision detection calculations on the GPU fall into the category of GPGPU and therefore in this section we will give a brief explanation of some of the principles of GPGPU.

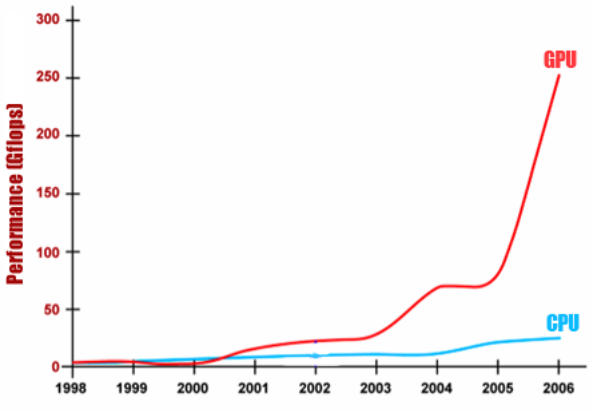

Figure 4: Increase in Computational Velocity for CPU and GPU Over Time [16]

3.1 What kinds of problems are suitable for GPGPU?

As mentioned previously the strength of the GPU is its ability to perform several parallel calculations. But for the GPU to actually be able to do the calculations parallelly there are some constraints on the type of calculations that can be done. In order for calculations on the GPU to be effective there has to be a function f that takes one or more arrays of length N as inputs and outputs an array of length N. For instance the traditional use of the GPU is to take arrays with positions and optionally normals and/or texture coordinates for each of the N pixels and output a color for each of the N pixels. When such a function as previously described is implemented on the GPU, one cannot be sure in what order the elements of the arrays will be processed. Therefore algorithms that depend on the order of calculations cannot be implemented in a good way on the GPU. For instance if you have a function $f(x) = y$ that needs the result of y_i to calculate y_{i+1} this cannot be effectively implemented on the GPU. In the ideal GPGPU problem the array elements are very inter-independent, because

it is then that there can be a lot of parallel calculations [11]. Problems that can be mapped to a 2d-grid are often well suited for GPGPU. Sending and fetching data to and from the graphics card memory is quite slow. Therefore problems with large data sets where you send the data and do a lot of calculations before fetching the result are well suited for GPGPU. But if there is a lot of sending and fetching of data in comparison to the arithmetic operations performed then the gain in calculation time is lost when sending the data back and forth. In other words, you would want a high computation per data load ratio.

3.2 Representing a problem on the GPU

We will in this section try to give a brief overview of how a problem can be taken from the CPU-domain to the GPU-domain.

As explained previously the input to a GPGPU application should be some kind of data array where each element requires similar computation, which is known as a stream. Data arrays or streams are loaded into textures on the GPU in order to be accessible in the shader program. There is support for 4-vectors (or less) and 4x4 matrices (or less), so if you have some data that fits then using those data types is preferable. Therefore, doing a texture sample on the GPU is the equivalent of indexing an array or doing a memory read. The computations that are performed as a step in the algorithm would usually be implemented inside a loop on the CPU. They can also be seen as a kernel to be applied to all the input array elements. This translates into a fragment shader program on the GPU. Storing data in an array corresponds to rendering to a texture on the GPU. This means that the texture has to have one texel for each data element that has to be calculated in the output array. A brief explanation of how to load the data onto textures and then set up to do the calculations and render the results is given in the section 3.5. Below is a table that summarizes the mapping of a problem from CPU to GPU.

CPU Domain	*GPU Domain*
Stream/Data Array	Texture
Memory Read	Texture Sample
Kernel/Loop body/Algorithm step	Fragment shader program
Save data in array	Render to texture

There are other possible approaches to GPGPU. Especially for new desktop GPU's which are constantly becoming more flexible to program and there are even GPGPU languages (CUDA, OpenCL, PhysX).

Note: Storing data arrays in textures has the consequence that if you need to send more data to the GPU than what can fit on a texture with maximal size allowed it will be very difficult to implement a solution to that the problem in a good (efficient) way.

3.3 Pros and Cons of implementing a GPGPU solution

Pros	*Cons*
Higher calculation speed	Takes longer to implement
More power efficient	Can't be done with unlimited precision
	Harder for someone else to maintain & update code

3.4 Implementing the algorithms

3.4.1 Programmable Shaders

An important feature in modern GPU's is that they have programmable vertex and fragment shaders that allows a greater flexibility in what one can do with the GPU. In our case having programmable shaders enables us to do GPGPU.

The algorithms are usually implemented in the fragment shader for several reasons which are listed below.

1. One can easily control how many fragments should be calculated and easily make sure that the output ends up in the right texels in the texture where the result of the calculations is stored.

2. There are more fragment pipelines than vertex pipelines.

3. You directly get the output as the fragment processor is at the end of the rendering pipeline (see Figure 5).

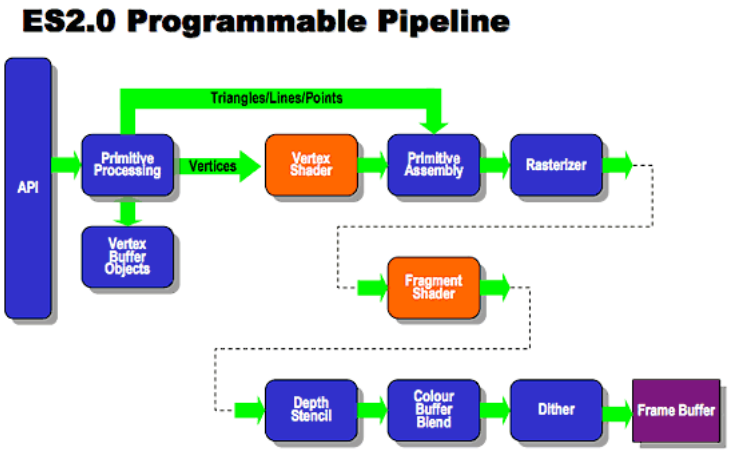

Figure 5: The OpenGL ES 2.0 Rendering Pipeline [6]

3.4.2 Vertex shader

The vertex shader usually doesn't implement any part of the algorithm directly, it only passes on texture coordinates to be interpolated and given to the fragment shader.

3.4.3 Fragment shader

This is where the actual function/kernel/algorithm is implemented. The code in the fragment shader is invoked once for each fragment or in other words once for each data element that is to be computed. To preserve parallelism one should try to avoid branches (such as conditional statements) in the code and in general try to have as few values that change dynamically as possible.

3.5 Setting up the calculations

Processing the input data and producing the output is done by rendering to the output texture. To be able to do this, first one can do the following [4]:

1. Load the input data onto a texture on the GPU.
2. Bind the output texture to a framebuffer object as the color attachment.
3. Bind this framebuffer so that any rendering done will be done to the framebuffer.
4. Create a quad (or two triangles that make up a quad) that has the same dimensions as the output texture or that covers the entire framebuffer if using normalized coordinates.
5. Set texture coordinates in each of the corners of the quad. These should be sent to the vertex shader program along with the positions of the quad. They will then be passed on to be interpolated for the fragments shader.
6. Set the viewport to the same size as the output texture.
7. Then draw the quad. All the steps above will ensure that the fragment shader will run once for each element in the input array and that the program will have the texture coordinates to access each of the input elements.

3.5.1 Loading the data onto the GPU

In OpenGL ES 2.0 data is loaded onto a texture with the command **glTexImage2D** [13]. This function takes as one of its parameters a pointer to an array. This means that even though we want the texture to be two-dimensional we need to put the data in a one-dimensional array. If we want to load data for a texture with dimensions NxM we need to put the data in an array of length NxM. The data is put in the array in order of the rows that we want the texture to have. So first in the array we put row one of the texture, then row two and so on. This is illustrated in Figure 6.

3.6 Collision handling on the GPU

Here we describe what has to be done in order to perform the broad phase and narrow phase of collision detection on the GPU. Since we already describe these algorithms in section 2 the focus here will be on all the things that differ from performing these algorithms on the CPU. In both broad phase and narrow phase we have had to go through a lot of trouble to be able to access floating

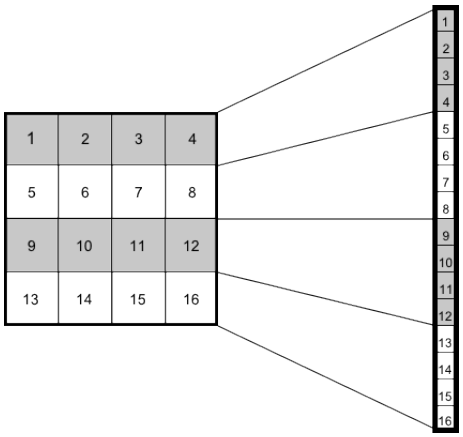

Figure 6: Illustration of how to put 2D data in a 1D array

point data within the shader programs. Since this is an issue in both phases and in any GPGPU calculation that you want to do on a platform that does not provide floating point textures this problem and it's work around are described in section 3.7.1.

3.6.1 GPU broad phase

In the broad phase each object in the simulation is compared with all of the other objects in the simulation. So if there are N objects in the simulation the result from the broad phase can be seen as an NxN matrix with one element for each comparison. As mentioned in previous sections the input data to the algorithm has to be loaded onto a texture and then the fragment shader program has to do lookups in that texture to retrieve the data necessary to do the calculations. To do SAT collision detection one needs quite a lot of data for each object being tested. The OBB's three basis axes that defines it's orientation relative to the world space, it's position (in world space) and it's halflengths with means half of the length of each side of the OBB are all needed. So for each of the objects in the simulation 15 float values have to be loaded onto the texture. Once the texture has been created the hardest part about retrieving the data from inside the shader is figuring out how to use the texture coordinates to access the right data at the right time.

Since all we want to do in the broad phase is determine whether any of the objects have collided we can return a RGBA value with (256,256,256,256) if two objects collide and (0,0,0,0) if they don't. This means that the shader will render onto a NxN texture where each texel will contain information whether a certain combination of objects collided with each other. But the input will be a 15xN texture. In the shader we will get texture coordinates $(s, t) \in [0.0, 1.0]$ that correspond to the middle of each texel in the output texture. So we need

to find a way to go from the texture coordinates that we get in the shader to the ones that will fit our input textures.

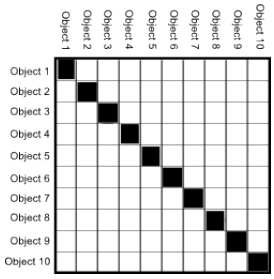

Figure 7: Output Texture for Broad Phase Collision Detection

If the output texture has the dimensions NxN the texture coordinates will have the form

$$s = i\frac{1}{N} + \frac{1}{2N}, \quad i = 0, 1, 2, \ldots, N-1$$

$$t = j\frac{1}{N} + \frac{1}{2N}, \quad j = 0, 1, 2, \ldots, N-1$$

As seen in Figure 8. In this example none of the objects collide except with themselves. This is why the diagonal is black.

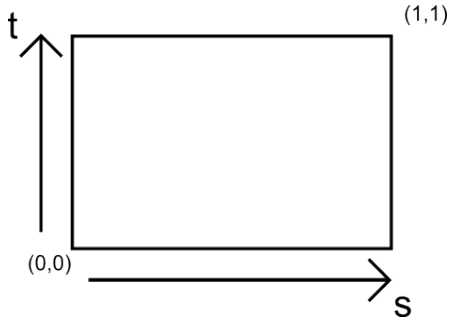

Figure 8: Texture Coordinates for an Arbitrary Texture

We have chosen to put the data that the SAT-algorithm needs for each object in the rows of the texture. So the texture coordinates for the rows correspond to what object is to be tested and the coordinates for the column point out what data value is to be fetched for a specific object. This is illustrated in Figure 9. What texture coordinates are sent to the fragment shader is determined by the dimensions of the output texture. One pair (s,t) is generated for each texel in the output texture. Because the output texture has the dimensions NxN we will get NxN pairs, one for each test between two objects that is to be done. Each time the fragment shader is run we need to retrieve the data for the two objects

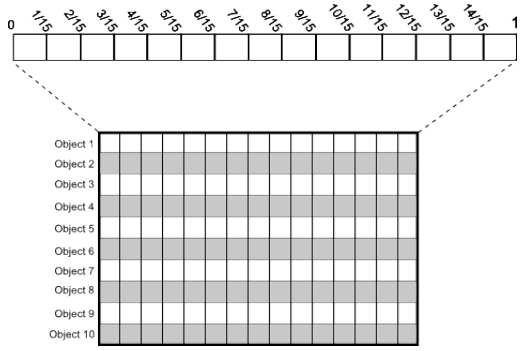

Figure 9: Input Texture for Broad Phase Collision Detection

that are to be tested against each other. Since the data for each object is put in the rows we can access each of these objects by using s as the row coordinate for one of the objects to be tested and t as the row coordinate for the other. Then we need to sample the texture at 15 column texture coordinates to get all the data we need for the object. Textures always has the interval $[0.0, 1.0]$ so the size of one texel will be $\frac{1}{15}$. To be sure that rounding errors don't put us in the wrong texel we always add half a texel to the texture coordinates. So in the fragment shader we get the following coordinates.

$$\left.\begin{array}{rcl} obj1_i & = & (s,i) \\ obj2_i & = & (t,i) \end{array}\right\} i = 0, \frac{1}{15}, \frac{2}{15}, \ldots, \frac{14}{15}$$

Where each $objX_i$ means the texture coordinates for one of the 15 data values needed for each object.

Therefore one has to fill an 1D array and with the data that has two dimensions. The way this is done is that the data that we want to become the first row in the texture is put first in the 1D array and then the second row and so on.

3.6.2 GPU narrow phase

Narrow phase is more complicated than the broad phase on the GPU. This is because the size of the data is varying. To do the narrow phase, all vertices in one object must be checked against all the faces of the other object. Faces are defined as a triangle polygon, so a face is 3 vertices that forms this polygon. Since the number of vertices and faces can be different for all the objects, it is difficult to write them all in a NxM sized array (texture). For the project, we have divided each collision into two arrays, one containing vertices for one of the objects, and the other containing faces for the other object that might collide. So if object A has 12 vertices, and object B has 6 faces, a 12x1 texture and a 18x1 texture will be created. If object A and B are colliding in broad phase, A's vertices will be checked against the faces of object B. The result that is returned is each vertex checked against every face in a NxM matrix, where N is the number of vertices, and M is the number of faces. A result of true for a

vertex against a face is generated if the vertex is behind the face, see section 2.2. This means that a collision has occurred if at least one of the vertices returns true for all faces.

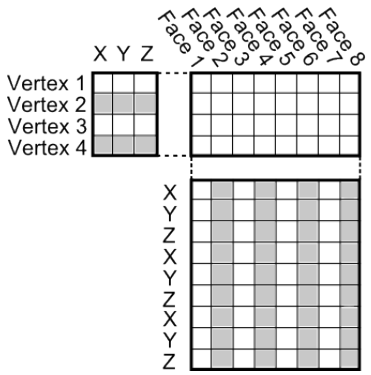

Figure 10: Both Input Textures and the Output Texture for Narrow Phase Collision Detection

3.7 Floating point textures

The support of floating point textures (textures that the values are represented as floats) is not standard for ES 2.0. There is an extension that supports it [13] and the chip that iPhone uses (PowerVR) supports this extension. Unfortunately Apple has chosen to not implement this feature in their version of OpenGL ES. Instead normal textures that only support integers must be used.

3.7.1 How to cope without floating point textures

When data is loaded onto textures in OpenGL ES 2.0 they are loaded as integers, unless floating point textures are supported. When a RGBA texture is sampled it can only be retrieved as vec4 (4 normalized floating point number). There is no possibility of doing bit-wise operations or reinterpreting the bit pattern as some other data type. So it is very hard to pass floating point values to a shader program.

But there is a way to be able to use floating point data within the shader without having floating point textures. The solution is converting the floating point numbers to fix point numbers and loading the fix point data into the texture and then converting the data from fix point numbers to floating point numbers in the shader. Therefore we will give a brief introduction to fix point numbers and then show how they can be used to pass floating point numbers to the GPU.

Fixed point numbers are integers that are scaled by a certain factor. So a fix point number f is an integer i scaled with a factor s meaning that $f = \frac{i}{s}$. The difference between fix point numbers and floating point number is that

floating point numbers have variable scaling factors that are embedded in the data type [21]. The reason why we can handle a fix point number on the shader is that because of the fixed decimal point we don't have to do bit-operations to be able to interpret the bit pattern (that we fetch from the texture) as a floating point number. All we have to do is divide the fix point number with the factor s to get the floating point number. This means that the application must have know what factor to use. For instance, in our case we use 32-bit fix point numbers where the factor s is 2^{16}. This means that the 16 first bits (starting from the most significant) represent the integer part of the number and the latter 16 represent the fraction.

Before putting them on the texture each floating point number float_nbr is converted to a fixed point number fix_nbr.

```
int fix_nbr = (int) (float_nbr *( (float) 1<<16 ));
```

In the shader we then fetch the value from the texture and convert it to a floating point number like this.

```
vec4 vec = texture2D(s_coord,t_coord);
int a = int(vec.x*255.0);
int b = int(vec.y*255.0);
int g = int(vec.z*255.0);
int r = int(vec.w*255.0);
int fix_nbr = r*256*256*256+g*256*256+b*256+a;
float float_nbr = float(fix_nbr)/(256.0*256.0);
```

And then have retrieved the original floating point number (although we might have some disparity due to loss of precision during the conversions).

If the application requires floating point output from the shader it can be done by converting the floating point number to fix point and then to a RGBA value.

```
float_nbr *= 256.0*256.0;
int fix_nbr2 = int(float_nbr);

int a2 = int(mod(float(fix_nbr2), 256.0));
int b2 = int(mod(float(fix_nbr2)/256.0, 256.0));
int g2 = int(mod(float(fix_nbr2)/(256.0*256.0), 256.0));
int r2 = int(mod(float(fix_nbr2)/(256.0*256.0*256.0), 256.0));

gl_FragColor = vec4(float(a2)/255.0,float(b2)/255.0,
float(g2)/255.0,float(r2)/255.0);
```

Then when the rendered values are retrieved, one has to convert them from fix point numbers to float numbers. If we again use the names float_nbr and fix_nbr, it can be done like this.

```
float float_nbr = ( (float) fix_nbr )/( (float) (1<<16) );
```

There can be issues about precision loss. Since 32-bit floating point are converted to 32-bit fixed point, the precision can at most be $1/2^{16}$, this is however often sufficient for most uses.

4 Results

With this thesis work we wanted to answer the question whether a collision detection on mobile devices could be accelerated by the performing it on the GPU. In order to answer this question we implemented different versions of the components in collision handling (see section 2), one version that does the calculations on the CPU and another that does them on the GPU. Below is found a UML-diagram over the collision handling classes.

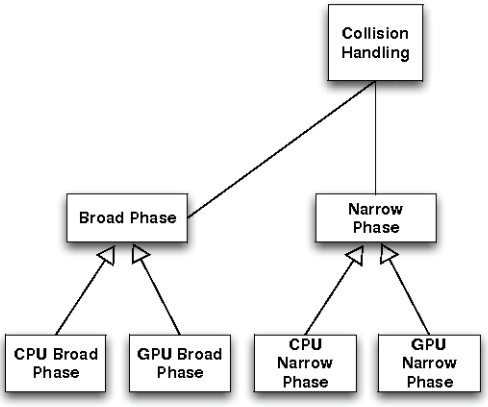

Figure 11: UML Diagram for the Collision Handling

By using inheritance we could easily switch between the different versions of the different components and measure execution time in the different cases. In the tables we present the average execution time per object. The test were run for a large number of times until the average time had converged to some value. The reason that we measure for different number of objects is to explore the impact of hardware limitations. Each object has 36 vertices and 12 faces (which is relevant to the narrow face).

Table 1 shows time measurements when using different combinations of broad phase/narrow phase versions with different numbers of objects in the simulation. We also did measurements to investigate the impact that all the conversions discussed in 3.7.1 has on execution time. We therefore measured the time when not doing these conversions. When doing this the collision handling does not get any correct results because the input that it receives is just uninitialized data. So when doing these tests we do the same texture lookups and we perform the same algorithm we just don't convert the data to floating point. Thus we get a good idea of how much time the algorithm would take if we had access to floating point textures. In both table we have used the average of atleast 1000 messurements.

We then did additional tests with higher numbers of objects in the simulation, to compare the different implementations of the broad phase of collision handling. The results can be seen below in Table 2.

	CPU/CPU	*CPU/GPU*	*GPU/CPU*	*GPU/GPU*	*GPU*/GPU**
4 Objects	0.082/0.105	0.058/1.070	0.403/0.116	0.595/2.029	0.311/1.103
16 Objects	0.175/0.102	0.176/1.052	0.381/0.109	0.447/1.935	0.232/1.058
49 Objects	0.273/0.106	0.277/1.071	0.471/0.103	0.470/1.914	0.145/1.036

Table 1: Results of broad- and narrow phase in milliseconds.*Here the conversion from floating point to fix point and back is not done.

	Broad Phase CPU	*Broad Phase GPU*	*Broad Phase GPU**
49 Objects	0.27	0.485	0.145
100 Objects	0.361	0.827	0.218
400 Objects	0.706	2.721	0.587
900 Objects	1.264	-	1.194

Table 2: Results of broad phase for large amount of objects in milliseconds. *Here the conversion from floating point to fix point and back is not done.

In figure 12 the result are displayed in a chart. Note that the axes are not linear.

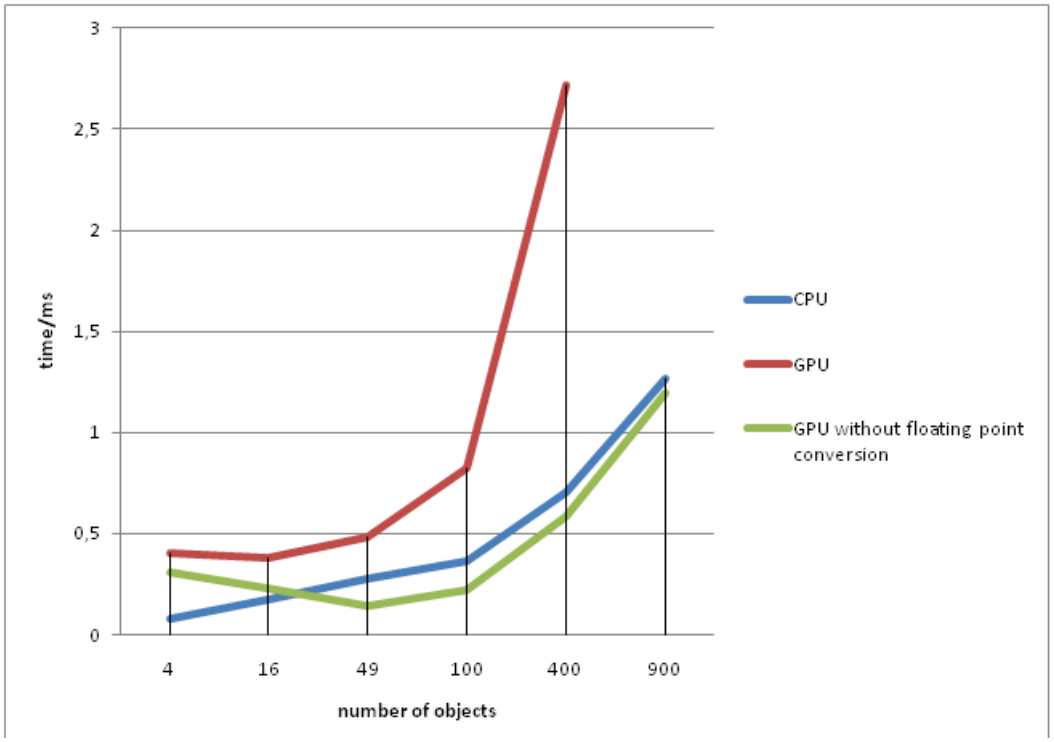

Figure 12: Result of the broad phase.

5 Conclusions

Our goal with his thesis work was to find out whether collision detection on mobile devices could be accelerated by performing the calculations on the GPU. In Table 1 and Table 2 we present the measurements of execution time on which we base the following conclusions.

Because the broad phase of collision detection has the time complexity of $O(\frac{N \times N}{2})$ and we measure the time per object (which means dividing by N) we expect that the broad phase execution time per object should increase linearly. However there are other factors such as hardware limitations that might also affect the execution time.

The broad phase is always slower than the CPU if the floating point conversion is used. However, without the conversion then for 49 objects and more, the GPU is faster. Especially around 49-100 objects the GPU is almost twice as fast. Then for more than 100 objects, the time difference starts to decrease. The conclusion is that broad phase is more effective for GPU if there's a large number of objects, and if there's support for floating point textures. For very large number of objects, the difference is not so big, probably because the texture for broad phase is getting too large for the hardware to handle it well. The standard implementation of ES 2.0 only supports texture sizes of 1024×1024, which limits the number of objects possible.

The time complexity of the narrow phase is linear with regards to the number of objects. However the narrow phase takes a lot longer for complex objects, i.e. objects with many vertices and faces, than for simple objects. So from an algorithmic point of view the execution time per object should be constant for the narrow phase.

Our measurements show that the CPU version of the algorithm performs approximately 10 times faster than the GPU implementation. One reason for this is that we have one texture per object so that for each pair of objects that might collide we need to allocate new memory on the GPU and load the data for the textures, and this is very slow. Then a big factor is also the conversion between floating point and fixed point numbers. When running the GPU narrow phase without float conversions it executed about twice as fast as with the conversions. Another factor that makes the GPU implementation slower is the following: When the CPU implementation finds that one vertex tests false against any face it stops because it can then be concluded that the vertex does not collide with the object. But the GPU implementation test all vertices against all faces and the result is read after all the test have been made. For a vertex that doesn't collide the GPU implementation therefore does many more test than the CPU.

To summarize our result, using the GPU for collision detection only the broad phase of the collision detection could be accelerated. Other physics calculations could not be made more effective. The broad phase is much faster around the interval of 49-100 objects, given that floating point textures are available (see section 3.7). We therefore conclude that it is not currently profitable to use the

GPU to do collision detection for mobile devices, since there is no hardware support for GPGPU. However at the time of finishing this work we have seen that mobile device manufacturers are working at implementing OpenCL and CUDA [17] [7]. This will most like lead to PhysX also being available with harware support for mobile devices. We therefore assume that GPU accelerated physics will come to mobile devices but not with the approach presented in this report, but with API's such as PhysX.

References

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

[2] Edward Angel. *Interactive Computer graphics 4th Edition*. Addison Wesley, 2006.

[3] James Anthony Bird. Gpu accelerated collision queries. Technical report, University of Abertay Dundee, School of Computing and Creative Technologies, 2008.

[4] D. Göddeke. Gpgpu–basic math tutorial. Technical report, FB Mathematik, Universität Dortmund, 2005. Ergebnisberichte des Instituts für Angewandte Mathematik, Nummer 300, `http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu`.

[5] GPGPU.org. `http://www.gpgpu.org`.

[6] Khronos Group. Opengl es 2.x and the opengl es shading language. `http://www.khronos.org/opengles/2_X/`.

[7] Khronos Group. Conformant products. `http://www.khronos.org/adopters/conformant-products/#topencl`, 2010.

[8] Hardware.fr. Cuda benchmark test. `http://www.hardware.fr/articles/678-7/nvidia-cuda-plus-pratique.html`, 2007.

[9] Chris Hecker. Physics - behind the screen. *Game Developer Magazine*, 1995-1997.

[10] D. Knott and D. K. Pai. "cinder - collision and interference detection in real-time using graphics hardware, 2003. www.cs.rutgers.edu/ dpai/papers/KnottPai03.pdf.

[11] David Luebke. General-purpose computation on graphics hardware. `http://gpgpu.org/static/sc2006/slides/01.luebke.Introduction.pdf`, 2006.

[12] Markus Malmsten and Simon Klasén. Practical collision detection on the gpu, a case study using cinder. Technical report, Department of Computer Science, Lund Institute of Technology, Lund, Sweden, 2005.

[13] Aaftab Munshi, Dan Ginsburg, and Dave Shreiner. *OpenGL ES 2.0 Programming Giude*. Addison Wesley, 2008.

[14] Hubert Nguyen. *GPU Gems 3*. Addison-Wesley, 2007.

[15] NinjaLane. Physx performance tests. `http://www.ninjalane.com/articles/general_information/physx_test/`, 2009.

[16] Tony Smith. Cool fusion: Amd's plan to revolutionise multi-core computing. *RegHardware*, 2006.

[17] Softpedia. Nvidia to offer cuda for mobile devices too. http://news.softpedia.com/news/NVIDIA-to-Offer-CUDA-for-Mobile-Devices-too-98360.shtml, 2008.

[18] Alan Watt and Fabio Policarpo. *3D Games: Real-Time Rendering and Software Technology*. Addison Wesley, 2000.

[19] Wikipedia. Bullet (software). http://en.wikipedia.org/wiki/Bullet_(software).

[20] Wikipedia. Cuda. http://en.wikipedia.org/wiki/CUDA.

[21] Wikipedia. Fixed point arithmetic. http://en.wikipedia.org/wiki/Fixed-point_arithmetic.

[22] Wikipedia. Nvidia tegra. http://en.wikipedia.org/wiki/Nvidia_Tegra.

A Physics calculations in 3D-games

Here we will give a brief discussion of the physics that is usually calculated in a typical 3D game engine.

A.1 Positions and orientations

In our physics engine we limit ourselves to rigid body dynamics. Most of our formulas have been taken from Chris Heckers article in Game Developer Magazine [9]. Because we're only dealing with rigid body dynamics, positions and orientations for each of the objects in the physics simulation is really all that we have to worry about. However there is a lot to do to be able to calculate the positions and orientations in a 3d physics simulation. Before we go into what we need to do to calculate the positions and orientations, we are going introduce the variables that are needed to solve this problem. For each object in the simulation we need to calculate or keep track of the following:

In all of the following definitions $\rho(x, y, z)$ is the function that gives us the density of the object at any given point in 3d.

The objects total mass M.

$$M = \int \rho(x, y, z)$$

The center of mass CM which is the 3d-position around which the object will rotate.

$$CM_x = \int\int\int x\rho(x, y, z)dxdydz$$

$$CM_y = \int\int\int y\rho(x, y, z)dxdydz$$

$$CM_z = \int\int\int z\rho(x, y, z)dxdydz$$

$$CM = \begin{pmatrix} CM_x \\ CM_y \\ CM_z \end{pmatrix}$$

Inertia tensor I.

$$I = \begin{pmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{pmatrix}$$

where

$$I_{xx} = \frac{1}{M}\int\int\int(y^2 + z^2)\rho(x, y, z)dxdydz$$

$$I_{yy} = \frac{1}{M}\int\int\int(x^2 + z^2)\rho(x, y, z)dxdydz$$

$$I_{zz} = \frac{1}{M}\int\int\int(x^2 + y^2)\rho(x, y, z)dxdydz$$

$$I_{xy} = I_{yx} = \frac{1}{M} \int\int\int xy\rho(x,y,z)dxdydz$$
$$I_{xz} = I_{zx} = \frac{1}{M} \int\int\int xz\rho(x,y,z)dxdydz$$
$$I_{yz} = I_{zy} = \frac{1}{M} \int\int\int yz\rho(x,y,z)dxdydz$$

The 3d position or the translation of the center of mass δ_{CM}.

$$\delta_{CM} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

The linear velocity of the center of mass v_{CM}.

$$v_{CM} = \dot{\delta_{CM}}$$

The linear acceleration of the center of mass a_{CM}.

$$a_{CM} = \dot{v_{CM}}$$

The objects 3d orientation which is represented by a matrix R.

$$R = \begin{pmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{pmatrix}$$

The angular velocity of the object ω, which is how fast the object rotates around some axis n_t at time t.

$$\omega = \dot{\theta} n_t$$

The axis subscript is to indicate that the rotating axis may be different at any time t in the simulation, because it depends on the forces and torques applied to the object.
The angular acceleration α.

$$\alpha = \dot{\omega}$$

The angular momentum about the centre of mass L_{CM} is the sum of the angular momentum of all the points in the object body, where m_i is the mass, v_i is the velocity, and δ_{CMi} is the position relative to center of mass for point i.

$$L_{CM} = \sum \delta_{CMi} \times m_i v_i = \sum \delta_{CMi} \times m_i \dot{\delta_{CMi}} = \sum m_i \delta_{CMi} \times (\omega \times \delta_{CMi}) =$$

$$\sum -m_i \delta_{CMi} \times (\delta_{CMi} \times \omega)$$

The cross products in the equation can be written on matrix form. For instance if a and b are 3x1 vectors we can represent the matrix form of a cross product with the tilde operator like this.

$$a \times b = \tilde{a}b = \begin{pmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

Using the tilde operator the angular momentum can be rewritten in terms of the inertia tensor and the angular velocity.

$$L_{CM} = \sum -m_i \tilde{\delta_{CMi}} \tilde{\delta_{CMi}} \omega = I\omega$$

The torque τ, is the derivative of the angular momentum.

$$\tau = \dot{L_{CM}}$$

A.2 The physics simulation

Here we show some pseudo code for the loop that does all the physics calculations i.e. calculates new positions and rotations for the objects.

```
dt = current_time - last_time;

for all objects:
    position = position + dt*velocity;
    rotation = rotation + dt*angular_velocity_as_cross_product*rotation;

    total_forces = total_forces_on_object();
    total_torque = total_torque_on_object();

    velocity = velocity + (dt*total_forces)/mass;
    angular_momentum = angular_momentum + dt*total_torque;
    inertia_tensor_inverse = rotation* inertia_tensor_inverse* rotation;
    angular_velocity = angular_velocity +
    inertia_tensor_inverse*angular_momentum;

    rotation.orthonormalize();


collision handling:
    may_collide = do_broad_phase();
    did_collide = do_narrow_phase(may_collide);
    do_collision_response(did_collide);
```

A.3 Collision Response

When we have established that two objects have collided, collision response is what we call the resulting impulses that are applied to the objects as a result of the collision. Given that a collision has occurred at the point p with the collision normal n the collision response is calculated with the following formulas. Here we write the center of mass for object i as δ_i and the velocity of object i as v_i and the mass of object i as M_i and it restitution coefficient e_i. The magnitude of the collision impulse j is

$$j = \frac{-(1+e)(v_A - v_B) \cdot n}{n \cdot n \left(\frac{1}{M_A} + \frac{1}{M_B}\right) + \left[(I_A^{-1}((p-\delta_A) \times n)) \times (p-\delta_A) + (I_B^{-1}((p-\delta_B) \times n)) \times (p-\delta_B)\right] \cdot n}$$

where $e = \frac{e_A + e_B}{2}$. The new velocities and orientations for object A and B can then be calculated.

$$v_A = v_A + \frac{j}{M_A} n$$

$$v_B = v_B - \frac{j}{M_B} n$$

$$L_A = \delta_A \times jn$$

$$L_B = \delta_B \times jn$$

$$\omega_A = I_A^{-1} * L_A$$

$$\omega_B = I_B^{-1} * L_B$$

A.4 ODE

To get the positions and orientations for an object at time t the following differential equations need to be solved.

$$v_{CM}(t) = \dot{CM} \Rightarrow CM(t) = \int v_{CM} dt$$

And the velocity itself is calculated from

$$a_{CM}(t) = \dot{v_{CM}} \Rightarrow v_{CM}(t) = \int a_{CM} dt$$

The orientation also has to be calculated by integration, however it's not necessary to use the time derivative of the 3x3 orientation matrix. Instead we use the ω that we defined above with the tilde operator.

$$\dot{R} = \tilde{\omega} R \Rightarrow R = \int \tilde{\omega} R dt$$

And ω is calculated as

$$\alpha = \dot{\omega} \Rightarrow \omega = \int \alpha dt$$

We use the Euler method to solve these ODE's, and thus at time $t + 1$ with a time step of size dt we get the following.

$$v_{t+1} = v_t + dt\alpha_{t+1}$$

$$CM_{t+1} = CM_t + dtv_{t+1}$$

$$\omega_{t+1} = \omega_t + dt\alpha_{t+1}$$

$$R_{t+1} = R_t + dt\tilde{\omega_{t+1}} R_t$$

Note: To make it more readable we have dropped the CM subscripts to give space for the time subscripts, but v and a still refer to the velocities and accelerations of the center of mass.