Memory Organization and Buffer Construction for GPGPU for 2D Physics

Scott Young

Department of Computer Science
Memorial University of Newfoundland

A dissertation submitted to the Department of Computer Science
in partial fulfilment of the requirements for the degree of
Bachelor of Science (Honours in Software Engineering)

December 2014

Memory Organization and Buffer Construction for GPGPU for 2D Physics

S. R. Young

Abstract- A 2D physics engine written in Java using OpenCL is presented. OpenCL is a software system that describes parallel computation on CPUs and GPUs. This thesis explores the data-structures required for 2D physics engines to speed up collision detection and position updating. Three strategies used to index a collision data structure are analyzed for GPUs and CPUs.

Memory Organization and Buffer Construction for GPGPU for 2D Physics

Table of contents:

Acknowledgements

I would like to acknowledge my supervisor Dr. Rod Byrne, who's guidance and support was paramount to the writing of this thesis. I would also like to thank my parents for the continued support in my academic career and for working around the my schedule. I'd like to thank both Karen Noseworthy and Dr. Todd Wareham their emotional support and lending me an ear when I needed to talk.

Without all of you, this thesis could not have been possible.

Chapter 1: Introduction

The advantages and disadvantages of organizing the data-structures on the GPU for 2D physics engine is explored. A brief background on the 2D physics required for this thesis is discussed, and a simplified overview of heterogeneous computation with OpenCL is described. This thesis uses a minimal sized data structure to store information about a collision. This thesis then explores three different strategies for retrieving relevant information about a collision from this structure.

In chapter 2, the background research involved in this thesis will be discussed. The topics discussed are: physics engines and some examples of their use, collision detection algorithms, collision resolution strategies, the Graphics Processing Unit (GPU) architecture, and the OpenCL standard.

In chapter 3, a basic implementation of a 2D physics engine in OpenCL using a 2D arena with rigid circles will be designed. OpenCL performs its calculations through kernels acting on buffers. Speed ups are achieved by careful design of buffers and kernels, where maximum speed up is achieved by having all parallelizable operations run in parallel.

In chapter 4, the kernels and how they act on buffers, which store the information required, to implement a 2D physics engine working on circles will be presented.

In chapter 5, the program that was used to test the 2D physics engine and the testing results will be discussed.

In chapter 6, the conclusion of the results from the test program will be discussed, and areas of potential future work will be mentioned.

Chapter 2: Background Research

Introduction

This chapter presents the background research involved in this thesis. The topics discussed are: physics engines and some examples of their use, collision detection algorithms, collision resolution strategies, the Graphics Processing Unit (GPU) architecture, and the OpenCL standard.

The physics engine

Physics engines are useful in many places. In general, physics engines are used to model the interactions of objects and forces in a simulated environment. Physics interactions usually happen in two or three dimensional space, often abbreviated to 2D or 3D. 3D physics engines are used in everything from simulating real world problems, creating simulators for instructional purposes (like aircraft pilot training or simulating surgery), and for simulating object behaviour in 3D games.[13][14] In contrast, 2D physics engines are used primarily for determining object behaviour in 2D games, like the recently popular "Angry Birds" which uses the "Box2D" physics engine as seen on the "Chrome Angry Birds" about page.[23]

Collision Detection algorithms

Collision detection (the process of determining if two objects have or will have collided) is often broken up into 2 to 3 phases. These phases are Broad-Phase, Mid-Phase (optional), and Narrow-Phase. Broad-Phase takes in all objects and outputs a set of pairs of possibly colliding objects, Mid-Phase takes in a set of pairs of objects, and outputs a set of pairs of possibly colliding objects, and Narrow-Phase takes in a set of pairs of objects, and outputs a set of pairs of colliding objects.[9] While there are many algorithms used to do each of these phases of collision detection, it is outside the scope of this thesis to address all of them or discuss them in detail. Instead, focus will be on very parallel algorithms that are ideal for the GPU.

For broad-phase collision detection in a robust 2D physics engine, the Combined Spatial Subdivision with Sweep and Prune[9] algorithm is currently one of the best known parallel algorithms. This is due to the fact that it is a easy to do in parallel, and was designed for use on the GPU. However for the simple example of a physics engine used in this thesis, the embarrassingly parallel Bounding Circles algorithm will be used. This algorithm compares the sum of the radii between two minimum bounding circles, with the distance between their centres. Where ever the radii sum is more than the separating distance, a collision has occurred[19]. Since the engine will only deal with circles, this algorithm will be sufficient for narrow-phase collision detection as well.

No mid-phase algorithm will be used, as objects will not be complex enough to require further pruning before narrow-phase detection in this 2D physics engine. This engine will also conveniently only return colliding objects after broad-phase and therefore require no more detection.

For narrow-phase collisions detection in a robust 2D physic engine, the Separating Axis Theorem(SAT) Algorithm is commonly used for polygon on polygon detection. SAT tests if two objects overlap by trying to find a place where it can separate the two. It is sufficient to check for lines parallel to the potentially colliding polygons sides.[10] As previously stated, due to the nature of the rudimentary of the engine made for this thesis, Bounding Circles was sufficient.

Collision Resolution

Collision Resolution (or Collision Response) is the act of moving objects and changing their speeds to avoid overlapping in a realistic manner.[16] Collision Resolution cannot be done using applied forces (or velocities alone) for a system of rigid bodies, because the non-instantaneous speed change will give overlapping objects for several frames. First bodies will need to be pushed apart; this is called Projection collision response.[17]

Impulse is a change in momentum, and therefore is very similar to a force, but the concept is that of a straight conversion instead of a rate. Impulse "J" is the change between the initial and final momentum over a period of time. Calculating this value for a given rigid body collision allows for the momentum after a collision to be calculated.

Methods which accelerate objects are called "Penalty methods". Penalty methods rely on spring forces to keep objects made of many points together, and apply forces on them when they attempt to move closer or further apart. They are great for soft body mechanics but are unusable for rigid bodies as the bodies will be overlapping for prolonged periods of time, which disobeys the principles of rigid body mechanics.[18]

Graphics Processing Unit

The GPU is a device which handles floating point vector arithmetic operations in parallel across many cores with great speed and accuracy.[6][7][8] for example, GPGPU has been used to great affect for many computationally intensive tasks with speed ups[20] of approximately 160 times (flow cytometry data clustering) and even up to 300 times (Monte Carlo simulation of photon migration in 3D turbid media).[3][4] These tasks are of a very specific nature, they require that the task be a single task operating on multiple data points, these tasks are ideal for an architecture called Single Instruction Multiple Data, hereafter SIMD. A great example of such a task would be collision detection, and in fact an algorithm for doing this efficiently using both a CPU and two GPUs has already been made, called a "Hybrid Parallel Continuous Collision Detection" algorithm by Duksu Kim et al.[5]

OpenCL

OpenCL is an open standard for parallel programming maintained by the Khronos group.[11] It defines an N-dimensional computation domain, and assigns each component of that domain a worker to do work on it.[12] OpenCL programs are called "Kernels" and the code on them can be executed on a diverse number of processing units such as Graphics Processing Units

and Central Processing Units. It targets a variety of platforms, from supercomputers to mobile devices. The emerging popularity of this has lead to the porting of the OpenCL C API to other languages, including java. There are three popular java APIs for OpenCL: JavaCL, JOCL, and Jogamp.[19] For this Project, JavaCL was used.

OpenCL Example Using JavaCL

The kernels execute on the compute devices, and are usually scheduled from the host code on the CPU. Host code tells the devices what kernels to run, supplies the data to work on, and tells the device when to run them. The data is stored in buffers. A copy of each buffer lives in the compute device's memory and another copy may live in the host codes memory. The kernels access the buffers through pointer arguments of the type that is stored in the buffer.[12]

Executing an OpenCL program is an 8 step process.

1. The host code queries the OpenCL devices.
2. A context and queue are created and associated with one or more OpenCL devices.
3. The host must create Buffers for each device, since buffers are unique to devices.
4. Programs with one or more kernels are created on one or more associated devices.
5. The kernels to be executed must be compiled.
6. Data must then be copied to the devices as needed.
7. The kernels are queued into the command queue for execution, they may be required to wait for previous kernels to finish, which is controlled by the host code.
8. The results must be copied from the device to the host.

As an example an OpenCL application that computes vector addition with three buffer objects is presented below. The first two buffers are initialized on the CPU, then the three buffers are given to the GPU. The kernel code stores the sum in the third buffer. The host code waits for the GPU to finish its kernel before copying the data from the third buffer and printing it to the screen.

The `add_floats` kernel takes in the three buffers, called `a`, `b`, and `out`, and the parameter `n` which is used for error checking in case more work units are queued than are required.

```
__kernel void add_floats(__global float* a, __global float* b,
                         __global float* out, int n)
{
    int i = get_global_id(0); // get the index of the vector this kernel will work on
    if( i >= n ){
        return;
    }
    out[i] = a[i] + b[i]; // sum the vector components and store in the out buffer
}
```

The global id for a work unit is stored in the variable `i`, this index is used to determine which elements of the buffers this work item will work on. The host code will determine the number of work items to be generated. Each instance of the kernel will get instantiated, this is called on a work item, and gets a unique index. Error checking is done using the parameter `n` to ensure that the program will only work items with indexes in the range will perform calculations. the `out` buffer will store the vector sum of `a` and `b`.

The host code for the OpenCL application needs to get the OpenCL context using the JavaCL `createBestContext()` function (step 1 and 2), create the queue(step 2), allocate pointers of the appropriate amount of memory, before initializing the `aPtr` and `bPtr` values. These Pointer objects will be initialized to <0, 1, 2, 3, ... n-2, n-1>. Buffers on the GPU are initialized from these Pointers(step 3). The CLProgram object is created from source code(step 4), and the the kernel from the program is created in the CLKernel object(step 5). The arguments to the kernel are set(step 6), and then the kernel is queued (step 7). Finally the output is copied back to host memory(step 8).

```
import com.nativelibs4java.opencl.*;
import com.nativelibs4java.util.*;
import org.bridj.Pointer;

import java.nio.ByteOrder;
```

```
import static org.bridj.Pointer.*;
class JavaCLTutorial {
    public static void main(String[] args) throws Exception{
        CLContext context = JavaCL.createBestContext();//steps 1 & 2
        CLQueue queue = context.createDefaultQueue();//step 2
        ByteOrder byteOrder = context.getByteOrder();

        int n = 1024;
        Pointer<Float>
                aPtr = allocateFloats(n).order(byteOrder),
                bPtr = allocateFloats(n).order(byteOrder),
                oPtr = allocateFloats(n).order(byteOrder);
        //initialize the Pointer objects to [1-1024]
        for(int i=0; i<n; i++){
            aPtr.set(i, (float)i);
            bPtr.set(i, (float)i);
        }
        CLBuffer<Float>
                a = context.createBuffer(CLMem.Usage.InputOutput, aPtr),//step 3
                b = context.createBuffer(CLMem.Usage.InputOutput, bPtr),//step 3
                out = context.createBuffer(CLMem.Usage.Output, oPtr);//step 3

        String src =
IOUtils.readText(JavaCLTutorial.class.getResource("AddArrayzKernel.cl"));//step 4
        CLProgram program = context.createProgram(src);//step 4

        CLKernel addFloatsKernel = program.createKernel("add_floats");//step 5

        addFloatsKernel.setArgs(a, b, out, n);//step 6
        CLEvent addEvt = addFloatsKernel.enqueueNDRange(queue, new int[]{n});//step 7

        Pointer<Float> output = out.read(queue, addEvt); //step 8
        for(Float value : output){
            System.out.println(value);//prints all even values between 1 and 2049
        }
    }
}
```

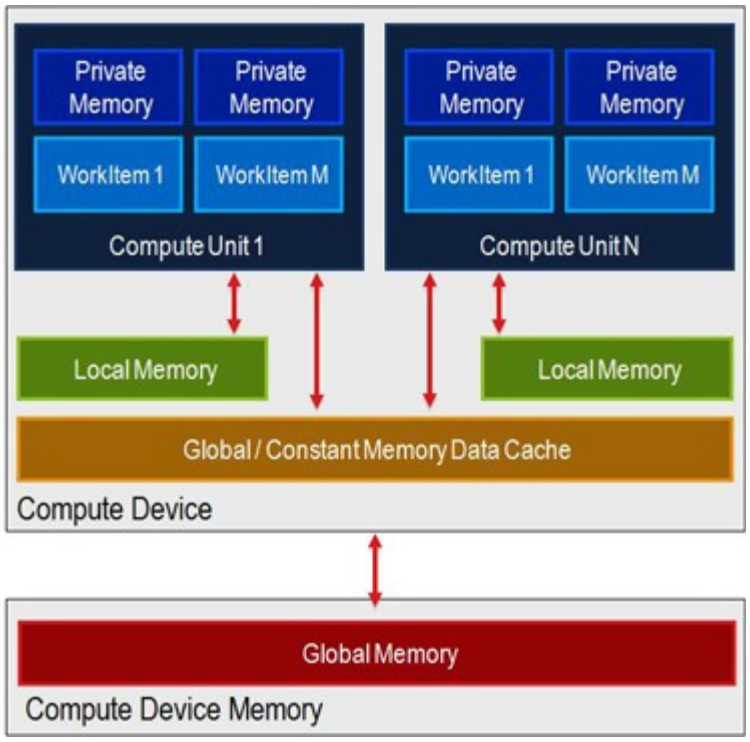
OpenCL Memory Organization

OpenCL uses three types of memory. Global which is accessible from any work item; local which is accessible from any work item inside of that local memory's compute unit (or Group), and private which is unique to each work item. This is shown below in Figure 1 [24].

Host code has to copy data to global memory on the device. Physically, when using the graphics card this means sending information over the bus. Memory buffers are one dimensional

arrays of memory that contain bytes of data which are transferred to and from a compute device. The work items then get data from global memory and either use the data as is or store the data in either private or local memory. Whether local or private memory is used will depend on the application. Then when work is done, data from private and local must be moved into global, and the kernel will report that it is done.

When the kernel is finished, the host code can copy memory from global to host memory via the memory buffers. Physically, when using the graphics card this means sending information back over the bus.



The OpenCL Memory Model

Figure 1[24]

Chapter 3: 2D Physics on a GPU

Introduction

To study the implementation of a 2D physics engine in OpenCL, a 2D arena with rigid circles will be used. OpenCL performs its calculations through kernels acting on buffers. Speed ups are achieved by careful design of buffers and kernels, where maximum speed up is achieved by having all parallelizable operations run in parallel.

2D physics

The circles in this engine will be rigid and non-deformable, meaning that each body will retain its shape at all times. The circles will exist in a space within a gravity field, so all objects will experience the same gravity. They must obey basic laws of kinematics, and therefore their positions will be updated with the following formulae for displacement *d*[21]

$$d = v_1 t + \frac{1}{2} a t^2$$

The formulae only works in one direction at a time, so it must be repeated in each dimension, in this case the *x* and *y* directions. The change in distance *d* is equal to the sum of the change in distance from velocity over time *v*t* and the change in distance over time due to acceleration $0.5*a*t^2$.

The bodies also must be able to collide in a realistic manner. To do so they must use the equation for conservation of momentum[22]

$$m_1 v_1^i + m_2 v_2^i = m_1 v_1^f + m_2 v_2^f$$

Which states that the sum of the momentum's before a collision, must equal to the sum of the momentum's after a collision. To allow objects to be bouncy or not, we introduce a coefficient of restitution for the collisions, and use the following formulas to calculate the

changes in speed after a collision between two objects[17].

$$v_a^f = v_a^i + \frac{J}{m_a} n$$
$$v_b^f = v_b^i - \frac{J}{m_b} n$$

Here v_a^f and v_b^f are the velocities of the first and second objects respectively after the collision, v_a^i and v_b^i are the velocities of the first and second objects (again respectively) before the collision, m and m are the masses of the first and second objects, n is vector that points in the direction that the collision happened in, called the "collision normal", and J is the impulse induced by the collision, calculated from[17]

$$J = \frac{-(1+e) v_{ab} \cdot n}{n \cdot n \left(\frac{1}{m_a} + \frac{1}{m_b}\right)}$$

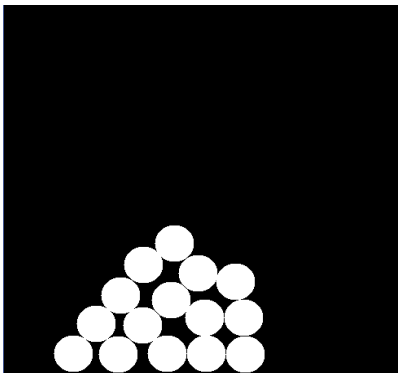
Where e is the coefficient of restitution for the collision, v_{ab} is the velocity of object a relative to object b, and the other variables are the same as described above.

Since all non-colliding circles update independently of each other in parallel, the kernel which updates these circles' positions and velocities, and the buffer which stores their states and velocities, will be indexed on the number of circles. This will achieve maximum parallelism.

The collision detection and resolution work with unique pairs of circles, and therefore each collision can be handled in parallel. The kernels which deal with these collisions must therefore be indexed on collisions, meaning that for every possible pair of objects a unique index must be assigned and the kernel should work on one collision. The buffer which stores information about a collision will also be indexed on collisions.

In summary, parallelism is obtained by indexing over circles for circle state information, and indexing over potential pair-wise collisions for collision information.

Chapter 4: Design of the Physics Engine



A frame from a simulation using the physics engine in this thesis

Introduction

This chapter will present the buffers which store information relevant to, as well as the kernels (and how they act on the buffers) required to implement a 2D physics engine working on circles. The arena will be a *1x1* square centred at *(0.5, 0.5)*.

The kernels will compute:

- Position and velocity for each circle.
- The MTV(Minimum Translation Vector) between each colliding pair of circles (3 different strategies)
- Update position and velocity of each colliding pair of circles (3 strategies)

The Buffers will represent:

- The circles' states, including its centre in Cartesian coordinates, radii, and mass
- The velocities of the circles
- The MTV of pair-wise groups of circles
- The *(i, j)* pair associated with a collision index (for strategy 1)
- The minimum and maximum collision index for a given row index (see Figure 2).

Collision Buffer Design

When considering buffer construction, a developer must consider the expense of sending information between the host code and the GPU. Sending data across the PCI x16 bus is expensive in comparison to memory access on the device, so the developer wants to minimize the number of transfers across the bus. The graphics card does cache information so that it does not need to grab it from the CPU every time it needs it, and it also has a large number of cores for doing parallel computation. Due to the large number of cores, it is beneficial to make each index independent as well. The problem is then "How should data on a buffer be designed to take advantage of how kernels access them, without taking up more memory than is required?".

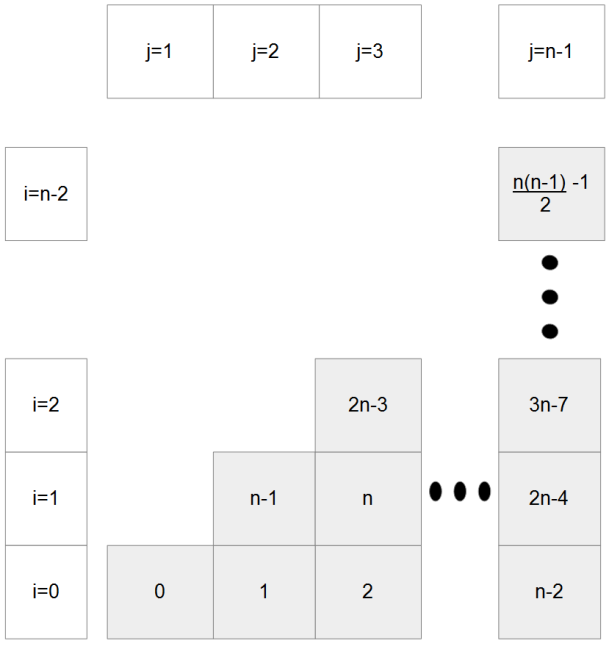
MTV buffer design

The solution is application dependant, but for 2D physics collision detection it is to create buffers using the minimum amount of space, indexed such that there is exactly one index for every collision, and use this space to store the Minimum Translation Vector(MTV), which describes how far apart the circles must move to prevent overlap. This array would have an index in the range *0 to maximum_collisions* where *maximum_collisions* can be calculated from the formulae below, given *n* circles:

$$\frac{n(n-1)}{2}$$

n is multiplied by n-1 because a circle can collide with any other circle, and this product is divided by 2 since the collisions should only be counted once (remove the case where *i* index is greater than *j* index for all *(i, j)*).

The buffer does not store the *(i, j)* pairs of objects which collide at each index. This presents a design choice, in that the there are several strategies for collecting the object indexes from the array. It is convenient to think of the buffer as a lower triangular matrix index as shown in Figure 2 below.



The lower triangular matrix representation of the buffer holding collision information.
Figure 2

The bottom row has a number of indexes equal to one less than the number of circles. The next row has one less index than the previous row, and so on until the top row which has one index. The indexes are enumerated *0* through to *n(n-1)/2* for *n* circles, increasing as i increases and as j increases. The indexing strategy for pulling an *(i, j)* pair from this array without another buffer is to first let *m=n-1,* then get *i* through the formulae:

$$ii = \frac{-2m - 1 + \sqrt{(4m(m+1) - 8c - 7)}}{-2}$$

if ii is its own greatest integer part then:

$$i = ii - 1$$

otherwise:

$$i = GIP(ii)$$

where GIP(x) returns the greatest integer part of x

To get *j* use the formulae:

$$j = c + \frac{i(i+1)}{2} - (i \times m) + 1$$

This guarantees the minimum amount of space to store the collisions (no redundancy and no duplicates) and offers a unique index for every collision. However it requires a square root, and a formulae with many operations including a greatest integer part function. This will become unstable as the number of circles in the system increases. This is the concept used for index strategy 0.

The size of this array does not change, and therefore the *(i, j)* pair for each collision could be precomputed and stored in a buffer. This buffer must be the same size as the buffer storing the MTVs, so using this indexing strategy would double the amount of memory used to store the collisions. This is the concept used for index strategy 1.

Since *j* can be calculated from *i* and *m* without a square root, a buffer that stores the first and last indexes of each row, indexed by rows, could be searched through with a binary search to find the correct *i* index. This uses an *m* sized buffer instead. This is the concept used for index strategy 2.

This engine is used to demonstrate how effective use of buffers can improve run-times and more efficiently use the limited space on the graphics hardware.

Buffers and Kernels

The physics engine represents the left-most (x) or lowest (y) point on the screen with a value of 0, and the right-most (x) or highest (y) point on the screen with a value of 1. All speeds and accelerations should be adjusted to work on this scale.

The physics engine is implemented with 6 memory buffers and 4 parameters:

- The buffers:
    1. circle – a buffer containing 4 dimensional vectors. Each vector represents the information about a single circle. The first element of each vector gives the x co-ordinate of the circle's centre, the second gives the y co-ordinate, the third gives the radius of the circle, and the fourth gives the mass of the circle. Giving each circle a radius independent of mass allows for arbitrarily density.
    2. velocities – a buffer containing 2 dimensional vectors, these vectors represent the linear velocities of the circles at their corresponding index's (i.e., velocities[i] is the speed of the circle who's information is in circle[i]). The first co-ordinate is the x component of the linear velocity, the second is the y component.
    3. gravity – a buffer storing a single vector of 2 dimensions. This vector is the acceleration that all circles in the system experience, the first component is the acceleration in the x direction, the second is the acceleration in the y direction.
    4. collision – a buffer containing 2 dimensional vectors. Each index refers to a collision. The vector stored at each index is the minimum vector that the circles must move apart in to prevent their over-lapping (minimum translation vector or MTV).
    5. indexes – a buffer containing 2 dimensional vectors. Each index refers to a collision. The vector stored at each index is the *(i, j)* pair of circles associated with this collision index.
    6. rows – a buffer containing 2 dimensional vectors. Each index refers to a row in the index table from Figure 2. The vector at each index stores the minimum and maximum collision index for that row.

- The Parameters:
    1. circles – this parameter is the number of circles in the system. It is used for error checking in case more kernels are queued than are needed, and in

equations for determining the indexes of the two circles colliding in each collision.

2. potential_collisions – this parameter is the number of collisions which might occur in the system. It is used for error checking in case more kernels are queued than are needed.
3. time_step – this parameter represents the time period over which the update circle positions occur. A smaller time step makes the system more accurate, but will cause less movement per iteration. Depending on hardware requirements for the system this may be a lower or higher number.
4. restitution – this parameter represents the restitution of collisions in the system. This engine implicitly expects all objects (walls, ceiling, floor, and circles) to be made of substances with similar "bounciness".

The OpenCL code that performs the engines position updating, collision detection and collision response is broken up into 7 kernels. The kernels indexed on collisions use functions to get the *(i, j)* collision pairs depending on the method of index retrieval they are using.

For retrieval by calculation (Strategy 0):

```
int2 collision_map_s1( int c, int circles )
{
    float circlesf = convert_float_rtz(circles-1);
    float cf = convert_float_rtz(c);
    float ii = (-2.0f*circlesf - 1.0f + native_sqrt( ( 4.0f*circlesf*(circlesf+1.0f)
- 8.0f*cf - 7.0f ) )) / -2.0f;
    if(convert_int_sat_rtz(ii) == ii ){
        ii = ii - 1.0f;
    }
    int i = convert_int_sat_rtz(ii);
    int2 pair;
    pair.s0 = i;
    pair.s1 = c + i*(i+1)/2 - i*(circles-1) + 1;
    return pair;
}
```

This function uses the formulas to calculate the *(i, j)* indexes.

For retrieval by look-up (Strategy 1):

```
int2 collision_map_s2( __global int2* indexes, int c )
{
    return indexes[c];
}
```

This function must take in an additional buffer of `indexes` which contains the *(i, j)* pair for a given collisions index and does not require information on the number of `circles` in the system.

For retrieval by row search (Strategy 2):

```
int2 collision_map_s3( int c, __global int2* rows, int circles)
{
    int i = 0;
    int low = 0;
    int high = circles-2;
    int mid = (high-low)/2;
    while(true){
        if(c < rows[mid].s0){
            high = mid;
            mid = (high-low)/2 + low;
        }
        else if( c > rows[mid].s1){
            if(mid-low == 0){
                i = high;
                break;
            }
            low = mid;
            mid = (high-low)/2 + low;
        }
        else{
            i = mid;
            break;
        }
    }
    int j = c + i*(i+1)/2 - i*(circles-1) + 1;
    int2 pair = (int2)(i,j);
    return pair;
}
```

This uses a buffer indexed on the *i* values from the *(i, j)* pairs called `rows` which contains the minimum and maximum collision index for each row. Using this, the kernel can find the *i* value using a binary search instead of the floating point calculation

1. Position updating:

The `update_position` kernel takes in the `circle` buffer, `velocities` buffer, `gravity` buffer, the `circles` argument, the `time_step` argument, and the `restitution` argument; all in that order. It uses the arguments and information from those buffers to calculate the new position of the circles and then updates the `x` and `y` component of the respective `circle` buffer index.

```
__kernel void update_position(__global float4* circle, __global float2* velocities,
                              __global float2* gravity ,int circles, float time_step,
                              float restitution)
{
    int i = get_global_id(0);
    if(i >= circles){
        return;
    }
    float2 newPosition = circle[i].xy + velocities[i]*time_step +
                         0.5f*gravity[0]*time_step*time_step;
    // check for wall collisions and resolve if they happen
    float2 changeFromWallBounce = (float2)(0,0);
    if(newPosition.x - circle[i].z < 0){
        newPosition.x = -(newPosition.x - circle[i].z)+circle[i].z;
        changeFromWallBounce.x = -2*(restitution)*velocities[i].x;
    }
    else if(newPosition.x + circle[i].z > 1){
        newPosition.x = 2-(newPosition.x + circle[i].z)-circle[i].z;
        changeFromWallBounce.x = -2*(restitution)*velocities[i].x;
    }
    if(newPosition.y - circle[i].z < 0){
        newPosition.y = -(newPosition.y - circle[i].z)+circle[i].z;
        changeFromWallBounce.y = -2*(restitution)*velocities[i].y;
    }
    else if(newPosition.y + circle[i].z > 1){
        newPosition.y = 2-(newPosition.y + circle[i].z)-circle[i].z;
        changeFromWallBounce.y = -2*(restitution)*velocities[i].y;
    }
    float2 newSpeed = velocities[i] + gravity[0]*time_step;
    circle[i].xy = newPosition;
    if(changeFromWallBounce.x == 0 && changeFromWallBounce.y == 0){
        velocities[i] = newSpeed;
    }
    else{
        velocities[i] = restitution*newSpeed + changeFromWallBounce;
    }
}
```

`update_position` first acquires its global index (one for each circle), then uses the `circles` argument to ensure that, even if more work items were assigned than the data set could use, it will never operate outside the data set of circles or velocities, which both have indexes from 0 to one less than the number of circles in the system. It then uses the standard kinematics equation for getting displacement from velocity and acceleration, and the current position (as a displacement from (0,0)) and stores this as the new position for the circle. The circle is checked against the minimum and maximum values it can have that would bound it to the screen.

2. Collision detection using no assistance buffers

The `detect_collisions` kernel takes in the `circle` and `collision` buffers, the number of `potential_collisions`, and the number of `circles`. It uses this information from all these arguments, except the `collision` buffer, to determine if a collision has occurred between two circles, and if so, stores their MTV in the `collision` buffer.

```
__kernel void detect_collisions(__global float4* circle, __global float2* collision,
                                int potential_collisions, int circles)
{
    int c = get_global_id(0);
    int i = 0;
    int j = 0;
    if(c >= potential_collisions){
        return;
    }
    collision[c].xy = (float2)( 0.0f, 0.0f );
    int2 pair = collision_map_s1(c, circles);
    i = pair.s0;
    j = pair.s1;
    float min_distance = circle[i].z + circle[j].z;
    float2 distance_vector = circle[j].xy - circle[i].xy;
    float distance = native_sqrt( dot( distance_vector, distance_vector ) );
    if(distance < min_distance){
        float2 normal = distance_vector/distance;
        collision[c] = (min_distance-distance)*normal;
    }
}
```

The kernel first retrieves its global index. It then initializes its *i* and *j* as 0, ensures its

global index is within the range of valid collision indexes, then sets the collision buffer value at its index to (0,0). Once the *(i, j)* pair is calculated by `collision_map_s1`, the distance separating the circles centres is found and is compared to the sum of their radii. If the separating distance is less than the sum of the radii, the MTV gets calculated and then stored in the collision buffer.

3. Collision detection using a of row indexed buffer

This kernel is a variation of the previous kernel. It calls the `collision_map_s3` function instead of `collision_map_s1`. In terms of the `circle` buffer `collision` buffer, the kernel operates the same.

```
__kernel void detect_collisions_precomputed_rows(__global float4* circle,
                                                 __global float2* collision,
                                                 __global int2* rows,
                                                 int potential_collisions, int
circles)
{
    int c = get_global_id(0);
    if(c >= potential_collisions){
        return;
    }
    collision[c].xy = (float2)( 0.0f, 0.0f );
    int2 pair = collision_map_s3( c, rows, circles);
    int i = pair.s0;
    int j = pair.s1;

    float min_distance = circle[i].z + circle[j].z;
    float2 distance_vector = circle[j].xy - circle[i].xy;
    float distance = native_sqrt( dot( distance_vector, distance_vector ) );
    if(distance < min_distance){
        float2 normal = distance_vector/distance;
        collision[c] = (min_distance-distance)*normal;
    }
}
```

4. Collision detection using a buffer of stored collision pairs

This variation of the collision detection performs the same operations as the other two collision detection kernels with respect to the `collision` buffer and the `circle` buffer with the same effect on the state and values of these buffers passed into the kernel.

```
__kernel void detect_collisions_precomputed_pairs(__global float4* circle,
                                                  __global float2* collision,
                                                  __global int2* indexes,
                                                  int potential_collisions)
{
    int c = get_global_id(0);
    if(c >= potential_collisions){
        return;
    }
    int2 pair = collision_map_s2(indexes, c );
    int i = pair.s0;
    int j = pair.s1;
    collision[c].xy = (float2)( 0.0f, 0.0f );
    float min_distance = circle[i].z + circle[j].z;
    float2 distance_vector = circle[j].xy - circle[i].xy;
    float distance = native_sqrt( dot( distance_vector, distance_vector ) );
    if(distance < min_distance){
        float2 normal = distance_vector/distance;
        collision[c] = (min_distance-distance)*normal;
    }
}
```

The `indexes` buffer is checked at the kernels global index to get the *(i, j)* pair for the collision index. Other than this change it is identical to the previous two kernels in how it performs its intended function.

5. Collision resolution

The `resolve_collisions` kernel takes in the `circle`, `collision`, and `velocities` buffers, the number of `potential_collisions`, the number of `circles` in the system, and the `restitution` of collisions. The purpose of this kernel is to adjust the velocities and positions of the circles in

collisions in a realistic manner.

```
__kernel void resolve_collisions(__global float4* circle, __global float2* collision,
                                 __global float2* velocities,
                                 int potential_collisions,
                                 int circles, float restitution)
{
    int c = get_global_id(0);

    if(c >= potential_collisions){
        return;
    }
    float2 mtv = collision[c];
    float mtv_length = native_sqrt( dot( mtv, mtv ) );
    if( mtv_length == 0 ){
        return;
    }
    int2 pair = collision_map_s1(c, circles);
    int i = pair.s0;
    int j = pair.s1;

    float2 posI = circle[i].xy - 0.5f*mtv;
    float2 posJ = circle[j].xy + 0.5f*mtv;
    circle[i].xy = posI;
    circle[j].xy = posJ;
    float invMassI = 1/circle[i].s3;
    float invMassJ = 1/circle[j].s3;

    float inverseMassSum = invMassI + invMassJ;
    float2 normal = mtv/mtv_length;

    float2 Vij = velocities[i] - velocities[j];
    float J = (-(1+restitution)*dot(Vij, normal))/(inverseMassSum);
    velocities[i] = velocities[i] + J*invMassI*normal;
    velocities[j] = velocities[j] - J*invMassJ*normal;
}
```

After retrieving the global index and ensuring the index lies within index ranges, this kernel retrieves the MTV between two objects then, unless the MTVs length is 0, moves the objects apart equally and in opposite directions along the MTV, before finally adjusting their speeds according to the impulse collision response equation. This kernel uses the same strategy for getting the *(i, j)* pair as the first `detect_collisions` kernel.

6. Collision resolution using a row indexed buffer

This kernel is a variation on collision resolution that uses the same binary search mechanism and buffer from the `detect_collisions_precomputed_rows` kernel. It otherwise behaves exactly as the original `resolve_collisions` kernel.

```
__kernel void resolve_collisions_with_rows(__global float4* circle,
                                           __global float2* collision,
                                           __global float2* velocities,
                                           __global int2* rows,
                                           int potential_collisions,
                                           int circles, float restitution)
{
    int c = get_global_id(0);

    if(c >= potential_collisions){
        return;
    }
    float2 mtv = collision[c];
    float mtv_length = native_sqrt( dot( mtv, mtv ) );
    if( mtv_length == 0 ){
        return;
    }
    int2 pair = collision_map_s3( c, rows, circles);
    int i = pair.s0;
    int j = pair.s1;
    float2 posI = circle[i].xy - 0.5f*mtv;
    float2 posJ = circle[j].xy + 0.5f*mtv;
    circle[i].xy = posI;
    circle[j].xy = posJ;
    float invMassI = 1/circle[i].s3;
    float invMassJ = 1/circle[j].s3;

    float inverseMassSum = invMassI + invMassJ;
    float2 normal = mtv/mtv_length;

    float2 Vij = velocities[i] - velocities[j];
    float J = (-(1+restitution)*dot(Vij, normal))/(inverseMassSum);
    velocities[i] = velocities[i] + J*invMassI*normal;
    velocities[j] = velocities[j] - J*invMassJ*normal;
}
```

7. Collision resolution using a buffer of stored collision pairs

This kernel is also a variation on collision resolution that uses the same buffer of *(i, j)* pairs from the `detect_collisions_precomputed_pairs` kernel. It otherwise behaves exactly as the original `resolve_collisions` kernel.

```
__kernel void resolve_collisions_with_pairs(__global float4* circle,
                                            __global float2* collision,
                                            __global float2* velocities,
                                            __global int2* indexes,
                                            int potential_collisions,
                                            float restitution)
{
    int c = get_global_id(0);

    if(c >= potential_collisions){
        return;
    }
    float2 mtv = collision[c];
    float mtv_length = native_sqrt( dot( mtv, mtv ) );
    if( mtv_length == 0 ){
        return;
    }
    int2 pair = collision_map_s2(indexes, c );
    int i = pair.s0;
    int j = pair.s1;

    float2 posI = circle[i].xy - 0.5f*mtv;
    float2 posJ = circle[j].xy + 0.5f*mtv;
    circle[i].xy = posI;
    circle[j].xy = posJ;
    float invMassI = 1/circle[i].s3;
    float invMassJ = 1/circle[j].s3;

    float inverseMassSum = invMassI + invMassJ;
    float2 normal = mtv/mtv_length;

    float2 Vij = velocities[i] - velocities[j];
    float J = (-(1+restitution)*dot(Vij, normal))/(inverseMassSum);
    velocities[i] = velocities[i] + J*invMassI*normal;
    velocities[j] = velocities[j] - J*invMassJ*normal;
}
```

Host Code

In order to run these kernels, host code must be created. Using Java and a set of wrapper classes for the C library function calls for OpenCL hosting called JavaCL, host code was created with a minimal development time.[19] The host code is set up as an API for use with programs.

It requires that the user of the API create a `Circles2D` object, with a Boolean `true` to use the GPU or `false` to use CPU.

The following code instantiates a number of variables that are all declared at the top of the class. The declarations have been omitted for ease of reading.

```
public Circles2D(boolean gpu){
    //boiler plate JavaCL code. Creates the context (chooses based on system)
    //then creates the queue for that context, to allow for kernel queueing
    if(gpu){
        context = JavaCL.createBestContext();
    }
    else{
        context = JavaCL.createBestContext(CLPlatform.DeviceFeature.CPU);
    }
    queue = context.createDefaultQueue();
    byteOrder = context.getByteOrder();
    position = new ArrayList<Float>();
    radius = new ArrayList<Float>();
    velocity = new ArrayList<Float>();
    mass = new ArrayList<Float>();
    circles = 0;
}
```

The constructor initializes all the objects which contain circle data before sending them to the compute device with OpenCL, creates the OpenCL context, and constructs the queue for that context. Note that passing `true` still does not guarantee that a GPU will be used, but rather that the compute device with the most compute cores on it will. For most computers that a given user might have at home this will be the GPU, however when a device with more cores does exist that will be chosen instead.

After that, circles are added via the `addCircle` method. This method requires the *(x,y)* centre of the circle, the radius of the circle, and the circles mass. Having a radius independent mass allows for arbitrarily dense circles.

```
    public void addCircle(float x, float y, float r, float mass) throws
IllegalArgumentException{
        if(mass <= 0){
            throw new IllegalArgumentException("Mass cannot be a zero or non-positive
                                                integer");
        }
        this.position.add(x);
        this.position.add(y);
        this.radius.add(r);
        this.mass.add(mass);
        circles++;
    }
```

This method puts circle data in the appropriate array lists and increments the counter used to keep track of the number of circles in the system.

The user of the API then calls the `init` method, supplying the time step for each iteration, the strategy of buffer indexing they wish to use, the restitution of collisions in the system, and a gravity "vector" in the form of an array with two elements corresponding to the $<d^2x, d^2y>$ acceleration due to gravity.

The following code instantiates a number of variables that are all declared at the top of the class. The declarations have been omitted for ease of reading.

```
    public void init(float timeStep, int strategy, float restitution, float[]
gravity) throws IOException{
        circlePointer = Pointer.allocateFloats(circles*4).order(byteOrder);
        velocityPointer = Pointer.allocateFloats(circles*2).order(byteOrder);
        gravityPointer = Pointer.allocateFloats(2).order(byteOrder);
        gravityPointer.set(0, gravity[0]);
        gravityPointer.set(1, gravity[1]);
        for(int i=0; i<circles; i++){
            circlePointer.set(4*i, position.get(2*i));
            circlePointer.set(4*i+1, position.get(2*i+1));
            circlePointer.set(4*i+2, radius.get(i));
            circlePointer.set(4*i+3, mass.get(i));
```

```
    velocityPointer.set(2*i, 0f/*(float)(Math.random()*0.4 - 0.2)*/);
    velocityPointer.set(2*i+1, 0f/*(float)(Math.random()*0.4 - 0.2)*/);
}
this.potential_collisions = circles*(circles-1)/2;
collisionPointer = Pointer.allocateFloats(this.potential_collisions *
                                          2).order(byteOrder);
for (int i = 0; i < this.potential_collisions; i++) {
    collisionPointer.set(2 * i, 0f);
    collisionPointer.set(2 * i + 1, 0f);
}
circleBuffer = context.createBuffer(CLMem.Usage.InputOutput, circlePointer);
velcityBuffer = context.createBuffer(CLMem.Usage.InputOutput,
                                     velocityPointer);
gravityBuffer = context.createBuffer(CLMem.Usage.Input, gravityPointer);
collisionBuffer = context.createBuffer(CLMem.Usage.InputOutput,
                                       collisionPointer);

src = IOUtils.readText(new java.io.File("src/opencl/CirclePhysV2.cl"));
program = context.createProgram(src);
program.addBuildOption("-cl-finite-math-only");

updatePositions = program.createKernel("update_position", circleBuffer,
                                       velcityBuffer, gravityBuffer,
                                       circles, timeStep, restitution);
if(strategy == 0){
    detectCollisions = program.createKernel("detect_collisions",
                                            circleBuffer, collisionBuffer,
                                            potential_collisions, circles);
    resolveCollisions = program.createKernel("resolve_collisions",
                                             circleBuffer,
                                             collisionBuffer, velcityBuffer,
                                             potential_collisions, circles,
                                             restitution);
}
else if(strategy == 1){
     //create collision pair buffer for strategy 1
     collisionPairPointer = Pointer.allocateInts(this.potential_collisions *
                                                 2).order(byteOrder);
    int first = 0;
    for(int i = 0; i < circles-1; i++){
        int m = first;
        first += circles-1-i;
        for(int j=i+1; j<circles; j++){
            collisionPairPointer.set((m+j-1-i)*2, i);
            collisionPairPointer.set((m+j-1-i)*2 + 1, j);
        }
    }
    collisionPairBuffer = context.createBuffer(CLMem.Usage.Input,
                                               collisionPairPointer);

    detectCollisions =
        program.createKernel("detect_collisions_precomputed_pairs",
```

```
                                circleBuffer, collisionBuffer,
                                collisionPairBuffer,
                                potential_collisions);
            resolveCollisions = program.createKernel("resolve_collisions_with_pairs",
                                                     circleBuffer, collisionBuffer,
                                                     velcityBuffer,
                                                     collisionPairBuffer,
                                                     potential_collisions,
                                                     restitution);
        }
        else{
            //create min and max collision per row buffer for strategy 2
             minAndMaxIndexPointer = Pointer.allocateInts((circles) *
                                                          2).order(byteOrder);
            int min = 0;
            int max = circles-2;
            for(int i=0; i<=circles-1; i++){
                minAndMaxIndexPointer.set(i*2, min);
                minAndMaxIndexPointer.set((i*2)+1, max);
                min = max+1;
                max += circles-2-i;
            }

            minAndMaxIndexBuffer = context.createBuffer(CLMem.Usage.Input,
                                                        minAndMaxIndexPointer);
            detectCollisions =
                program.createKernel("detect_collisions_precomputed_rows",
                                circleBuffer, collisionBuffer,
                                minAndMaxIndexBuffer,
                                potential_collisions, circles);
            resolveCollisions = program.createKernel("resolve_collisions_with_rows",
                                                     circleBuffer, collisionBuffer,
                                                     velcityBuffer,
                                                     minAndMaxIndexBuffer,
                                                     potential_collisions,
                                                     circles, restitution);
        }
    }
```

This method instantiates the pointer objects, sets their values using the data collected and stored earlier in array lists. The pointers are then used to create OpenCL memory buffers which will be used as arguments to the kernel call. The OpenCL source file is read in and the program object is created from it. Finally, the kernel objects are created from their names in the program, and their arguments.

Running the engine after the setup is done simply by calling `loop` every time the user

wishes to run another iteration of the engine. This method takes no arguments and returns a 2 dimensional array of floats that contains the circle data for each circle.

```
public float[][] loop(){
    float[][] circs = new float[4][circles];
    CLEvent update = updatePositions.enqueueNDRange(queue, new int[]{circles});
    CLEvent detect = detectCollisions.enqueueNDRange(queue,
                                                     new int[]
                                                     {potential_collisions});
    CLEvent resolve = resolveCollisions.enqueueNDRange(queue,
                                                       new int[]
                                                       {potential_collisions});

    Pointer<Float> out = circleBuffer.read(queue, resolve);
    for(int i=0; i<circles; i++){
        circs[0][i] = out.get(4*i);
        circs[1][i] = out.get(4*i+1);
        circs[2][i] = out.get(4*i+2);
        circs[3][i] = out.get(4*i+3);
    }
    return circs;
}
```

Loop creates `CLEvent` objects from queueing the kernels. These objects are used for ensuring that kernels have completed before certain functions happen. The resolve event gets used to make sure that circle data is not read from the circle buffer until the collision resolution has finished. The values from the circle buffer are read to a 2D array and returned for the API user to do with as they please.

Chapter 5: Testing

Introduction

In this chapter the test program created to run the physics engine through its tests is presented. The machine specifications for the computer which ran the program are presented, and the results of the data collected from those tests are discussed.

Test Program

The test programs used for this thesis is called `CirclesTest`. It creates a number of circles in the world at random positions with random radii and weights, all within threshold values defined below. The program uses the `Circles2D` class to simulate these circles in 2D space, and displays them to the screen using the Swing and AWT libraries of Java.

`CirclesTest` has a number of static variables which are used for easy manipulation throughout the program.

```
private static final int width = 480;
private static final int height = 480;
private static final float[] gravity = {0.0f,-0.01f};
private static final float timeStep = 1f/60f;
private static final int numberOfCircles = 1250;
private static final float minimum_radius = 0.01f;
private static final float radius_range = 0.005f;
private static final float minimum_x = 0.1f;
private static final float x_range = 0.8f;
private static final float minimum_y = 0.1f;
private static final float y_range = 0.8f;
private static final int strategy = 2;
private static final float restitution = 0.5f;
private static final long sleepTime = 17L;
private static final float minimum_mass = 1f;
private static final float mass_range = 9f;
```

The `width` and `height` variables store the dimensions for the display window. The `gravity` array holds the *x* and *y* gravity values; the units are in *%window/second*^{2}. The `timeStep`

holds the amount of time over which updating positions happens; without this value the velocities and gravity have no meaning. The `numberOfCircles` is the number of circles simulated by the system. The `minimum_<variable>` and `<variable>_range` arguments are used to scale the values into the range of desired values for the simulation. Minimum is the lowest value a variable can have, and the range is the scalar. The `restitution` is the percentage of elasticity in a collision, a restitution of 0 would cause the objects to behave completely inelastic, and a restitution of 1 would cause the circles to behave perfectly elastic. The `sleepTime` variable is used to time the window refresh rate.

The program uses an array of arrays of floats to store the circle data, a `Circles2D` object stored in the variable `engine`, and a local variable for the number of circles stored in the system.

```
private float[][] circles;
private Circles2D engine = new Circles2D(true);
private  int circleCount;
```

The constructor is simplistic, taking in only a parameter for the number of circles to be used in the system. It sets the preferred dimensions of the window, the back ground colour, calls the `init()` function, and then makes the first call to `simulate()`.

```
public CirclesTest(int circleCount){
    setPreferredSize( new Dimension( width + 20, height + 40) );
    setBackground( Color.BLACK );
    this.init(circleCount);
    this.simulate();
}
```

The `init` function takes in the number of circles in the system, initializes the circles array, adds the random circles to the system, then calls the `init` function of the engine.

```
private void init(int circleCount){
    this.circleCount = circleCount;
    circles = new float[4][circleCount];

    for(int i=0; i<circleCount; i++){
        engine.addCircle((float)(x_range*Math.random())+minimum_x,
```

```
                            (float)(y_range*Math.random())+minimum_y,
                            (float)(radius_range*Math.random())+minimum_radius,
                            (float)(mass_range*Math.random())+minimum_mass);
        }

        try {
            engine.init(timeStep, strategy, restitution, gravity);
        }
        catch(IOException e){
            System.err.println("Failed to find opencl file");
            e.printStackTrace();
        }
    }
```

`CirclesTest` has a simple function for performing a single iteration of the engine called `simulate`

```
    public void simulate(){
        circles = engine.loop();
        repaint();
    }
```

The function uses the engine to perform a `loop` of the simulation, and then `repaints` the graphics for the scene.

This class extends the `JPanel` class, and overrides the following functions

```
    protected void paintComponent( Graphics g ){
        super.paintComponent( g );
        Graphics2D g2d = (Graphics2D)g;
        for(int i=0; i<circleCount; i++){
            Color color = new Color( 0, (int)((205f/19f)*(circles[3][i]-1f)+(50f)),
(int)((205f/19f)*(circles[3][i]-1f)+(50f)));
            g2d.setColor(color);
            g2d.fillOval((int) ((float) width * (circles[0][i] - circles[2][i])),
                    (int) ((float)height-(float)height*(circles[1][i] + circles[2]
[i])),
                    (int) (((float) width) * circles[2][i] * 2f),
                    (int)(((float) height) * circles[2][i] * 2f));
        }

    }

    public static void runApplication(final JPanel app){
```

```
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                JFrame frame = new JFrame();

                frame.setSize( app.getPreferredSize() );
                frame.setTitle( app.getClass().getName() );
                frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
                frame.add( app );
                frame.setVisible( true );
            }
        });
    }
```

These functions are used to paint the circles to the screen and perform standard boilerplate for setting up a Swing application.

The `main` method of `CirclesTest` creates and initializes an instance of itself and then infinitely loops on `simulate`, with a `sleep` command to keep the screen repainting from happening more often than the frame limit intended.

```
    public static void main(String[] args){
        CirclesTest test = new CirclesTest(numberOfCircles);
        CirclesTest.runApplication(test);
        Scanner sc = new Scanner( System.in );
        while(true){
            long start = System.currentTimeMillis();
            test.simulate();
            long end = System.currentTimeMillis();
            long runtime = end-start;

            System.out.println("took " + runtime + " milliseconds to loop");
            try{
                Thread.sleep(sleepTime - runtime);
            }
            catch(Exception e){
                Thread.currentThread().interrupt();
            }
        }
    }
```

The program needed to be fast and efficient, therefore a software tool for evaluating the kernels was used during development.

Test Program Results

Testing the application on a system with the following specifications

- CPU

  Device name - AMD FX-8350 Eight-Core Processor

  Max Clock Frequency – 4096hz

  Cores – 8

- GPU

  Device name – MSI AMD R9 270x 4G

  Max Clock Frequency – 1030hz

  Stream Processing Units – 1280

The application above was used to run a test in which circles with random starting velocity and positions would interact. The tests were timed from when the simulate function was called until it returned. The application was run for 1000 iterations for each test. The tests were performed with 1250, 2500, 5000, 10000, and 15000 circles. Each of the three strategies for indexing was tested on the GPU and the CPU. The results of those tests are recorded in Figure 3.

The values are in milliseconds and represent the average time for a single iteration to complete.

Circle Count	CPU Strategy(0,1,2)	GPU Strategy(0,1,2)	Speed Up
1250	(11,2,9)	(<1,<1,<1)	(>11,>2,>9)
2500	(37,15,34)	(<1,1,1)	(>37,15,34)
5000	(128,58,121)	(2,2,2)	(64,28,60.5)
10000	(504,216,473)	(6,9,15)	(84,24,31.5)
15000	(N/A,451,1100)	(16,26,36)	(N/A,17,30.6)

Table representation of the results from the tests. Values for the CPU and GPU are in milliseconds and represent the average time to perform an iteration of the engine.

Figure 3

The GPU ran each strategy relatively equally for circle counts less than ten thousand, but after that it became apparent that look-up tables were slower to use than floating point arithmetic operations. The complete look-up table was better than binary searching through a row table. It was also noticed that at high circle counts (>2116), the floating point method would occasionally miss collisions on the GPU. This is likely due to some numerical instability in the calculation for finding the *ii* value, and the GPU performing slightly less accurate floating point arithmetic.

The CPU was much the opposite and from the start it was apparent that a complete look-up table was the best option. The complete look-up table was consistently at least twice as fast as binary searching through a row table. The row table was consistently faster than using floating point arithmetic, but until 10000 circles showed minimal improvement. At 15000 circles the program failed to run using strategy 0, and therefore a Not Applicable entry lies in its field.

Chapter 6: Conclusion

The following conclusion can be determined for the testing of the three strategies. Strategy 0 used a floating point formula for determining the row, and then an integer formula for calculating the column. Strategy 1 used a one-to-one look-up table that returned the *(i, j)* pair for a given collision. Strategy 2 used a row indexed table of bounds and a binary search to find the row, and then used the integer formula from strategy 0 to find the column.

Strategy 0 was the fastest on the GPU, due to the GPUs ability to quickly perform floating point arithmetic. But was slower than all other strategies tested on the CPU. However the GPU used in the tests failed to accurately perform the indexing due to numeric errors after reaching 2116 circles with this strategy.

Strategy 1 was the next fastest on the GPU and was reliable for all circle counts tested. On the CPU it was the fastest strategy, outperforming strategies 0 and 2 by close to a factor of 2.

Strategy 2 was the slowest strategy for the GPU. This is likely due to the GPU performing branch and jump instructions slower than CPUs. The CPU performed better with strategy 2 than strategy 0, likely due to the CPU performing jump and branch much more efficiently than floating point arithmetic.

In all cases, the GPU provided significant speed up. For less than 2116 circles, using strategy 0, the GPU outperforms the CPU by a factor of 11, and for greater than 2116 circles using strategy 1 the GPU outperforms the CPU by as much as 28 times.

OpenCL was a an efficient language for creating parallel code in and made General Programming on the Graphics Processing Unit (GPGPU) very easy to learn.

JavaCL was an easy to learn wrapper package that made memory management for OpenCL a breeze. Developing the test program and the API took very little time.

In the future, work needs to be done to devise a software design principle, a set of design patterns, and a set of design principles for heterogeneous computing for 2D physics. As well as a set of strategies for picking the right data structures for the right devices at run time. Further Testing also needs to be done on more numerically stable algorithms for calculating the *(i, j)* pair with floating point arithmetic.

Appendix: References

1: https://code.google.com/p/box2d/source/browse/branches/Predict/Box2D/Collision/b2CollideCircle.cpp (see functions “b2CollideCircles” and “b2CollidePolygonAndCircle”)

2: https://code.google.com/p/box2d/source/browse/branches/Predict/Box2D/Collision/b2CollidePolygon.cpp (see function “b2CollidePolygons)

3: http://cyberaide.googlecode.com/svn/trunk/papers/08-cuda-biostat/vonLaszewski-08-cuda-biostat.pdf

4: http://www.opticsinfobase.org/view_article.cfm?gotourl=http%3A%2F%2Fwww.opticsinfobase.org%2FDirectPDFAccess%2F7961D2BA-CF6E-A054-14A22E42DD9F09F1_187243%2Foe-17-22-20178.pdf%3Fda%3D1%26id%3D187243%26seq%3D0%26mobile%3Dno&org=

5: http://onlinelibrary.wiley.com/doi/10.1111/j.1467-8659.2009.01556.x/abstract Abstract.

6: http://www.nvidia.com/gtx-700-graphics-cards/gtx-770/

7: http://www.amd.com/en-us/products/graphics/desktop/oem/r9 under specifications

8: http://www.nvidia.ca/object/what-is-gpu-computing.html

9: “Fast GPU-based Collision Detection” by Lin Loi, Master's Thesis, University of Gothenburg, December 2010

10: http://rocketmandevelopment.com/blog/separation-of-axis-theorem-for-collision-detection/

14: http://www.khronos.org/opencl

12: http://www.khronos.org/assets/uploads/developers/library/overview/opencl_overview.pdf

13: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2810833/

14: https://code.google.com/p/bullet/

15: http://www.oodesign.com/visitor-pattern.html

16: http://chrishecker.com/images/e/e7/Gdmphys3.pdf

17: http://research.ncl.ac.uk/game/mastersdegree/gametechnologies/physicscollisionrespons

e-impulsemethods/Physics%20Tutorial%206%20-%20Collision%20Response.pdf

18: http://research.ncl.ac.uk/game/mastersdegree/gametechnologies/physicscollisionresponse-penaltymethods/Physics%20Tutorial%207%20-%20Collision%20Response%20-%20Penalty%20Methods.pdf

19: http://devmag.org.za/2009/04/13/basic-collision-detection-in-2d-part-1/

19: http://e-archivo.uc3m.es/bitstream/handle/10016/17183/finalversionPFC_Raquel_Medina.pdf;jsessionid=FD0E8F7D813816ECC4D41B8B5254843E?sequence=5

20: http://home.wlu.edu/~whaleyt/classes/parallel/topics/amdahl.html

21: http://plaza.obu.edu/corneliusk/ps/phys/kelm.pdf

22: http://mfes.com/colm_2005.pdf

23: http://chrome.angrybirds.com/

24: http://www.codeproject.com/KB/showcase/Memory-Spaces/image001.jpg