

Simulating Gravity with WebGL

Phil Ellwood

10th May 2012

G450 Computer Science (Games and Virtual Environments)

Project Supervisor Dr Graham Morgan

Word Count - 9108

Abstract

An increasingly significant percentage of people's time is spent using the browser, and this will only increase with time. In 20 years, the browser will be the platform, everything will run solely from it. WebGL is the first step to having rich, graphically complex applications on the browser. It allows the browser to have hardware accelerated graphics, previously only available to desktop applications. The dissertation aims to explore what is possible with this new technology.

The n-body problem is the problem of predicting the movements of a number of celestial bodies, affected by each other's gravity. This is a very computationally complex task, as each body has to work out the force exerted on it by every other body in the simulation. Most methods for approximating a solution are $O(n^2)$, drastically limiting the number of bodies shown on screen.

The project aims to create an interactive n-body simulation game using WebGL and JavaScript, paving the way to more complex WebGL applications in the future, and harnessing the power of the graphics card in the browser.

Declaration

“I declare that this document represents my own work except where otherwise stated”

Phil Ellwood

Acknowledgements

I would like to thank my project supervisor Graham Morgan, who started me on the road to WebGL. He allowed me to complete the project at my own pace, and his advice has been invaluable.

Table of Contents

Simulating Gravity with WebGL	1
Abstract	2
Declaration	3
Acknowledgements	4
Table of Contents	5
1 Introduction	7
1.1 Definition of problem	7
1.2 Aims	7
1.3 Plan	8
1.3.1 Explanation of the work plan	8
1.3.2 Diagrammatic Work Plan	8
1.4 Dissertation structure	9
1.5 Summary	10
2 Research	11
2.1 Introduction	11
2.2 WebGL	11
2.2.1 OpenGL	11
2.2.2 The beginning of WebGL	11
2.2.3 Security Concerns	11
2.2.4 Examples	11
2.3 Physics Simulation	12
2.3.1 Newtonian Physics and Classical Mechanics	13
2.3.2 Euler Integration	13
2.3.3 Runge-Kutta	13
2.4 N-body problem	14
2.4.1 History of the n-body problem	14
2.4.2 Numerical Integration	14
2.5 Summary	16
3 Implementation	17
3.1 Introduction	17
3.2 Design	17
3.2.1 Structure	17
3.2.2 Modelling rotation	18
3.2.3 Prototyping	19
3.2.4 Faster matrix operations with gl-matrix	19
3.2.5 Efficient frame request	20
3.3 Implementation	20
3.3.1 Implementing physics	20
3.3.2 Handling collisions	22

3.3.3 Making the simulation more modifiable	24
3.4 Summary	25
4 Evaluation	27
4.1 Introduction	27
4.2 Findings	27
4.2.1 CPU Usage	27
4.2.2 JavaScript CPU Profile	39
4.3 Fulfillment of objectives	31
4.4 Summary	32
5 Conclusions	33
5.1 Introduction	33
5.2 What has been learnt	33
5.3 If the project could be redone	33
5.4 Project completion	33
5.5 Future work	34
References	35

1 Introduction

1.1 Definition of problem

An increasingly significant percentage of people's time is spent using the browser, and this will only increase with time. In 20 years, the browser will be the platform, everything will run solely from it. WebGL is the first step to having rich, graphically complex applications on the browser. It allows the browser to have hardware accelerated graphics, previously only available to desktop applications. WebGL is relatively new, the specification was only released on 3rd March 2011. The dissertation aims to explore what is possible with this new technology.

The n-body problem is the difficulty of anticipating where a star or other large body will be at a given time, while it is affected by the gravitational force of other large masses. Most methods use direct integration to model the positions numerically. The problem with this is that the computation is $O(n^2)$, lowering the total number of bodies that can be shown in real time. For those that overcome this problem (the model is invariably static) the user cannot change elements in the simulation in real time.

The ability to alter simulation properties in real time could have beneficial applications in gaming, leading to more realistic space games. Additionally, it could be used in astrophysics, having a model which can be updated in real-time, giving users the ability to quickly see what effect a variable change might have.

1.2 Aims

The main aim of the project was:

"To model and simulate real orbital mechanics using WebGL in a 3D game"

This was broken down into objectives, and further split into sub-objectives.

- Become proficient with WebGL
 - Make a prototype without any physics
- Research existing methods for simulating orbital mechanics
 - Research how to do basic physics simulations
 - Find information on planetary motion
 - Find the equations relating to the n-body problem
 - Port the equation into code
 - Learn methods for increasing efficiency of the equation
- Create basic gravity simulation
 - Create celestial body object with needed properties
 - Create methods for updating those properties
 - Populate game with many celestial objects

- Develop simulation into a full, player controlled game
 - Create player object, with methods for movement
 - Make player effect other celestial objects

These objectives were made to be as measurable as possible. With the exception of “Research...” sub-objectives, they can be described as successful, or unsuccessful.

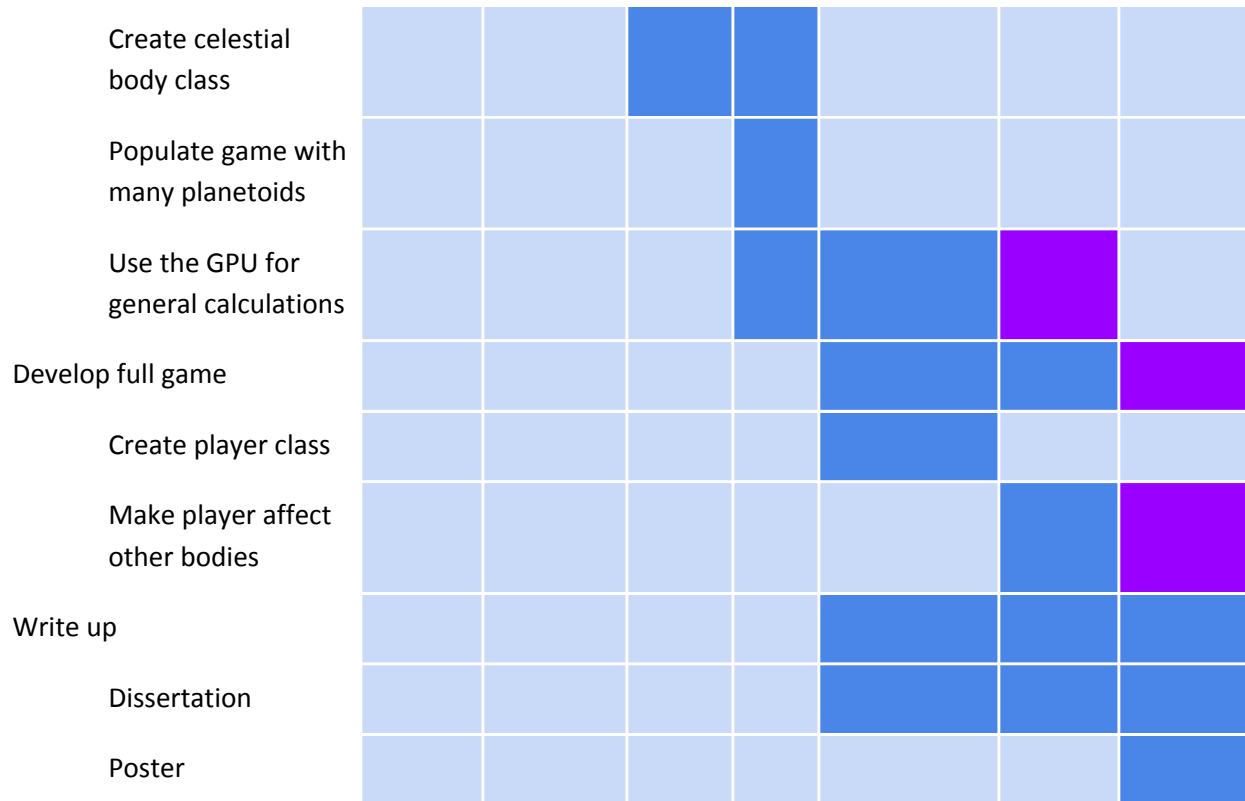
1.3 Plan

1.3.1 Explanation of the work plan

All the tasks stated in the objectives are listed. Purple marks the contingency time for tasks that may be risks. It would have been preferable to show more granularity with this chart, but it wasn't possible with the tools for this editor.

1.3.2 Diagrammatic Work Plan

	O	N	D	J	F	M	A
Make aims and objectives							
Become proficient in WebGL							
Prototype							
Research existing for coding orbital mechanics							
Basic physics simulation							
Orbital Mechanics							
N-body problem equations							
Port equation into code							
Increase efficiency							
Create basic gravity simulation							



1.4 Dissertation structure

Introduction

The introduction presents the problem the dissertation aims to correct, and its main aims and objectives. It is split into:

- Definition of Problem
- Aims
- Plan
- Dissertation Structure

Research

This chapter documents the literature review that was undertaken prior to the design and implementation. It is split into:

- WebGL
- Physics Simulation
- N-Body Problem

Implementation

The implementation chapter shows how the project was designed and carried out, and why certain decisions were made. It is split into:

- Design
- Implementation

Evaluation

In the evaluation, the implementation is analysed, along with how it could have been improved. How well the project has fulfilled the objectives is discussed. The subsections are:

- Findings
- Fulfillment of objectives

Conclusion

The conclusion discusses the success of the project, and areas the project could expand into in the future. It is split into:

- What has been learnt
- If the project could be redone
- Project completion
- Further work

1.5 Summary

This chapter introduces the dissertation, hopefully familiarizing the reader with the concepts, and letting them know more about the project, specifically:

- What WebGL is, and its place in the browser
- The difficulty of approximating a solution to the n-body problem
- The stated aims and objectives of the project
- The development plan of the project
- The structure of the dissertation

2 Research

2.1 Introduction

Research was undertaken to investigate essential areas and identify potential problems. The research was split into different categories, according to the initial objectives. These were WebGL, Physics simulation, and the n-body problem and general orbital mechanics. Each of these areas is discussed in detail in this chapter.

2.2 WebGL

This subsection goes into the research undertaken for WebGL, briefly visiting its roots in OpenGL and its birth, to examples of its use, and the frameworks and libraries that have sprung up to supplement it.

2.2.1 OpenGL

The foundations of WebGL are based in OpenGL, “a widely used specification of an API for rendering graphics” [1]. OpenGL started as a way to standardize access to graphics cards, allowing developers to write for any piece of hardware that had an OpenGL implementation and not have to concern themselves with the details of the underlying system. As a result, OpenGL became one of the most common ways to display complex graphics, and most graphics cards today have an OpenGL implementation.

2.2.2 The beginning of WebGL

WebGL’s inception came from Vladimir Vukićević’s demonstration of an OpenGL 3D context at xtech 06. The Khronos Group, “A not for profit industry consortium creating open standards for the authoring and acceleration of parallel computing, graphics, dynamic media, computer vision and sensor processing on a wide variety of platforms and devices”[2], which creates the standards for OpenGL as well, formed a group to start defining the WebGL API. The WebGL 1.0 specification was released in March 2011, it provides a JavaScript binding to OpenGL ES 2.0, a version of OpenGL for embedded systems.

2.2.3 Security Concerns

The current builds of several of the most widely used browsers have WebGL functionality built in [3] showing how widespread the adoption is. One notable exception is Internet Explorer. Microsoft has stated “WebGL is not a technology Microsoft can endorse from a security perspective” [4]. This is due to an advisory from the United States Computer Emergency Readiness Team warning about Denial of service, and cross domain attacks. This is an obvious disadvantage to WebGL, but there is a Khronos Group press release about the current state of security, and how the problems are being addressed [5]. Namely, the GL_ARB_robustness extension that prevents denial of service and out of range memory access, and other safeguards in the main specification to prevent access of uninitialized memory.

2.2.4 Examples

There are many sites that show interesting WebGL examples, from complex physics simulations [6],



to interactive, and fun, games [7].



2.3 Physics Simulation

Integral to the n-body simulation are the physics that control how celestial bodies move in reaction to a force, the gravitational force in the case of this simulation. As well as doing the calculations to find out how much gravitational force is exerted on an object, that force needs to be input into a physics engine, to see where the body will be at the next timestep. Creating a physics engine is not trivial, and

significant research was undertaken to find out how best to implement one. The following sub-chapter discusses the various implementations and their benefits.

2.3.1 Newtonian Physics and Classical Mechanics

The first step in a physics engine is how position is handled, WebGL only needs to know the position. A star's momentum, acceleration and the forces acting upon it have no bearing on what is shown in a given frame. This does not mean, however, that these other elements are unnecessary, the physics engine exists to calculate where a certain body will be next, based on the forces exerted on it. Force, in this context, is equal to mass times acceleration. Acceleration is the rate of change of velocity, rearranging these, it is seen that the rate of change of velocity is equal to force over mass.

$$F = ma$$
$$dv/dt = a$$
$$F/m = dv/dt$$

Given that velocity is the rate of change of position, if an object's position, velocity and mass are known, as well as the forces that are acting upon it, integration can find the position and velocity at some point in the future.

2.3.2 Euler Integration

Euler integration is the most basic form of numerical integration, every timestep, the position and velocity of the object is calculated using their previous values, then on the next timestep, the new position and velocity are re-calculated using these values [8].

```
//dt is the change in time
position := position + velocity * dt
velocity := velocity + ( force / mass ) * dt
```

The benefit of Euler integration is in its simplicity, not many calculations are needed. Unfortunately, Euler integration is not particularly accurate. This comes from the fact that when velocity is integrated to find the position, velocity is constantly changing over time due to acceleration, meaning the value for the position at a given time from the integrator is not the true value. To be more accurate, a way to detect how the derivatives change while integrating a value is needed.

2.3.3 Runge-Kutta

The Runge-Kutta order 4 integrator addresses the inaccuracy of the Euler integrator by calculating the derivative multiple times in a timestep, then generating an average of the samples to get a better approximation. There are two main parts to the algorithm, the first part is a function called evaluate which takes a state, the delta time and a derivative, performs an Euler integration using velocity and

acceleration, and returns a Derivative object. The second part is a function called integrate that takes a state and dt, calls evaluate multiple times to get sample derivatives, then calculates a weighted sum of the derivatives, and uses that new better derivative. [9]

```
void function integrate
    //samples some derivatives
    Derivative a = evaluate ( state, dt * 0.0f, Derivative());
    Derivative b = evaluate ( state, dt * 0.5f, a);
    Derivative c = evaluate ( state, dt * 0.5f, b);
    Derivative d = evaluate ( state, dt * 1.0f, c);

    //calculated weighted sum of the derivatives
    float dxdt = 1.0f / 6.0f * (a.dx + 2.0f * (b.dx + c.dx) + d.dx);
    float dvdt = 1.0f / 6.0f * (a.dv + 2.0f * (b.dv + c.dv) + d.dv);

    //use the new derivative to advance position and velocity
    state.x = state.x + dxdt * dt;
    state.v = state.v + dvdt * dt;
```

Using the derivative that is calculated from the weighted sum of the derivatives means that change over time, that standard Euler integration neglects, is accounted for. There are a few things to note about the above example; one is that it is only calculating the change in the x position, it is one dimensional. A model that would utilise the RK4 integrator in three dimensions, like the one proposed for this project, would need to take into account all three axes, potentially as a vector. Secondly, it uses acceleration to move the object, the proposed project would use forces, so a better way to do this would be to integrate force directly, to get momentum, though this would introduce other problems to tackle in the implementation. Lastly, the example only tackles movement, not rotation. The proposed project would need to handle the orientation of an object, and its angular momentum.

2.4 N-body problem

2.4.1 History of the n-body problem

The roots of the n-body problem are in the three body problem, first studied and discussed in Principia [10], it is the problem of ascertaining the position of three celestial bodies at a given time, while they are affected by each other's gravitational force. The three body problem is extended to the n-body problem as more bodies are added.

One of the objectives of the project was to research Kepler's laws of planetary motion [11], however brief research showed that they were only tangentially related to the n-body problem. They describe how orbits look, not how they work, or how to implement them.

2.4.2 Numerical Integration

There are a number of different methods used to do n-body simulations. Each have their own

advantages and disadvantages. The simplest method is Particle-Particle simulation. This involves finding the gravitational force exerted on every particle, by every other particle, and then integrating the equations of motion using the accumulated forces.

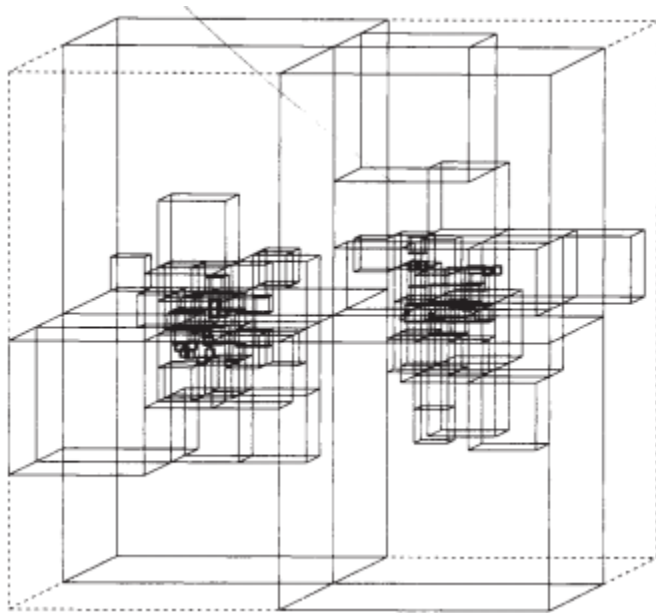
```
for every star i
    for every other star j
        force := force + calculateForce(i,j);
    end
end
integrate();
```

While straightforward, the Particle-Particle method has a high computational cost, for N particles, you are calculating the force $N-1$ times. $O(N^2)$ operations are taking place. Another problem sometimes faced by the Particle-Particle method is that as two particles get nearer to each other, the forces between them become larger which can lead to particles having nonsensically high velocities. Using a very short, or a variable timestep can overcome this though. [12]

The benefit of Particle-Particle is in its simplicity, and the fact that, with the right timestep, it can model close range dynamics very accurately.

Particle-Mesh is a more complex method, but is faster than Particle-Particle, as only $O(N + N_g \log N_g)$ computations take place. N_g is the number of grid points. Particle-Mesh works by placing a mesh over the simulation, working out an average force field over the mesh, then interpolating the force on the mesh to find the force acting on a particle. As previously stated, its main benefit over Particle-Particle is its speed, but it also has disadvantages. Because the force is interpolated over the mesh, close contact between particles is handled very poorly. In addition, Particle-Mesh does not cope very well when the particles are not in a uniform layout.

A very different method that is often used for n -body simulation is the Barnes-Hut algorithm. “The technique uses a tree-structured hierarchical subdivision of space into cubic cells, each of which is recursively divided into eight subcells whenever more than one particle is found to occupy the same cell” [13]. A tree is constructed, based on how the space is subdivided. Each cell is tagged with the total mass of its particles and their centre of mass. When calculating force on a particle, nearby particles are calculated as normal but further away, the cube’s mass is used.



The benefit of Barnes-Hut is its speed, taking only a time of $O(N \log N)$. Unlike Particle-Mesh however, it is still very accurate as neighbours are treated the same as Particle-Particle. Its disadvantage comes from the fact that it needs a large amount of memory. [13]

2.5 Summary

This chapter documents the background research undertaken. It covers:

- History of WebGL and its current usage
- How to implement a physics engine
- The n-body problem, and different algorithms for approximating a solution

The research in this chapter was used extensively in the design and implementation of the project.

3 Implementation

3.1 Introduction

This chapter will discuss the design and implementation of the project, including why certain design decisions were made, problems encountered during coding and how they were overcome.

3.2 Design

3.2.1 Structure

The project was designed to be object oriented, as the game could easily be split into classes. The player star would inherit from the more general star object and all stars would have update and animate functions, and physical state properties. In other projects, using JavaScript's prototypical inheritance made it easy to carry out functions on different object types in an array - a feature that would make it useful for iterating through the different game objects, updating their physical state and drawing them to the screen, so it was decided to utilise this feature.

While initially the design had the state of each planet contained inside that class, early research into different methods of simulating physical state and the design decision to use the Runge Kutta order 4 integrator, showed that another class would be needed for physical state properties, as a star would have two states, previous and current. In the research chapter the modifications to the example of the Runge Kutta integrator were briefly discussed, namely a modification to allow object rotation to be modelled, and a modification to use force and momentum to move the object instead of acceleration.

The state class holds the stars size and mass, which in the design were meant to be constant, though this was changed later. Attributes are split into two sets, primary physical values, and secondary physical values, this was one of the modifications necessitated by using force and momentum to move objects instead of acceleration. All of the primary values directly affect a secondary value, for example, momentum (a primary value) is equal to mass times velocity (a secondary value), this means that every time momentum is updated, velocity needs to be recalculated, to make this less error prone, a function called recalculate was added to state, that would recalculate all the secondary values from the primary values, this could then be called whenever a primary value was updated.

The primary values stored are:

- Position
- Momentum
- Orientation
- Angular Momentum

Secondary values are:

- Velocity - affected by momentum
- Angular Velocity - affected by angular momentum
- Spin - affected by angular velocity, and orientation

The star class holds a star's current and previous state; the functions needed for the Runge Kutta integrator, evaluate and integrate; the update and draw functions; and the most important function of all, forces. This is the function that calculates the gravitational forces, using the particle-particle method mentioned in the research chapter, the decision to use this method, even though it is the least efficient, is based on two main reasons, firstly, it is the simplest to implement, so after completing a rudimentary version of the game it could be extended to use a more complex method using trees or meshes. Secondly, the particle-particle method deals with collisions and near-misses more accurately than the more complex methods, though there are still optimisation that can be made

3.2.2 Modelling rotation

Some of the values stored in the state class show the other modification to the Runge Kutta integrator example that was needed. To model the rotation of an object requires more information than the amount required to model the movement of the object, to simplify this, and lower the amount of information that needs to be stored and recalculated, the design decision was made to model the objects as rigid bodies. This means that they cannot deform, cutting down on the number of calculations required.

The attributes needed to model the rotation of an object are similar to the ones needed to model movement; instead of position, there is orientation instead of velocity, there is angular velocity. Angular velocity is a vector that is both speed of rotation, determined by the length vector, and the axis around which the object is rotating, determined by the direction of the vector. Angular velocity does not stay constant in the absence of force, so angular momentum is used instead. Angular momentum is calculated similarly to the way linear momentum is calculated, as linear momentum is integrated from force, angular momentum is integrated from torque: the rotational equivalent of linear forces.

Orientation is much more complicated than position in three dimensions. There are two common ways of storing orientation in interactive graphical programming; 3x3 rotation matrices, or quaternions. Quaternions are 4 dimensional vectors that represent an axis of rotation and the rotation around that axis. It was decided to use quaternions for the project, as there would be fewer operation when calculating the change in rotation than there would be if rotation matrices were used, and it was felt that this would increase efficiency. Another reason that contributed to the decision to use quaternions instead of rotation matrices is that quaternions do not suffer from gimbal lock the same way that 3x3 rotation matrices can. Gimbal lock occurs when rotation in one axis becomes aligned with another, for example, if an aeroplane pitches up by 90 degrees, yaw and roll become the same motion. While gimbal lock should not occur with a body only rotating around one axis, it was thought using quaternions would make the project more extensible in the future.

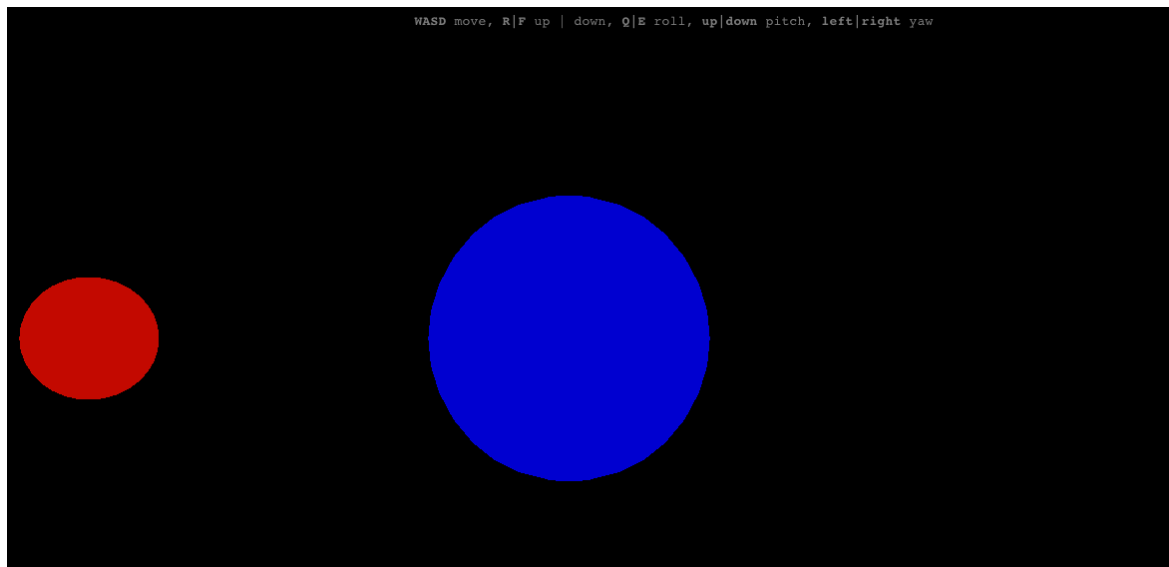
Once the method of storing orientation was decided, another problem presented itself. Similarly to how position is integrated from velocity, orientation is integrated from angular velocity, but orientation is a quaternion, and angular velocity is a vector, and it is not possible to integrate between two different

mathematical forms. The solution was to convert angular velocity into a quaternion, then use that quaternion to integrate orientation. The resulting derivative of orientation was named spin.

3.2.3 Prototyping

The initial design method used rapid prototyping, to enable more discussion at supervisor meetings, and to more easily test how different elements would work together. Originally the whole project was to be implemented in Three.js, as opposed to pure JavaScript and WebGL, as this would abstract away from the complexities of drawing complex graphics to the scene, and allow for concentration on implementing the physics engine that would drive the n-body simulation.

The first prototypes [14] did indeed use Three.js, which allowed for fast iterations. This allowed demonstrations to be shown in advance of when they would have been shown if pure WebGL had been used. However, as the author completed other projects without using frameworks, it was felt that the traditional method would allow more flexibility, and while it would lessen the speed at which new iterations could be developed, it would overall be beneficial to the project. This decision will be discussed further in later chapters.



3.2.4 Faster matrix operations with *gl-matrix*

In using the more conventional pure WebGL and JavaScript approach that was used in other projects, it was found to be beneficial to use an open source JavaScript library called *gl-matrix* [15], which was designed to handle matrix and vector calculations, of the kind that the project would make a lot of use of, very fast [16]. Implementing the sort of functions used in the library from scratch could potentially have saved memory, as not all the functions would have been used, but to implement them as efficiently as they are would have required a great deal of research into JavaScript that would have been beyond the scope of this project.

3.2.5 Efficient frame request

In the research almost all WebGL demonstrations online were found to use the same small snippet of code described by Paul Irish [17] as the `requestAnimationFrame` function developed by mozilla [18] and described in the w3c specification [19], is not implemented exactly the same in all browsers, and the snippet implements it in a cross-browser compatible way. Using `requestAnimationFrame` means that if the simulation is on a tab that is not visible, the animation will not run, meaning it will not be using the CPU, GPU, or memory, leading to a longer battery life.

3.3 Implementation

3.3.1 Implementing physics

Once the prototyping stage had been completed, the next step was building the classes, needed for physics simulation, that were decided on in the design process. Coding these in JavaScript was relatively simple, and leveraging JavaScript's prototypical inheritance to make calling the update and draw functions of all the stars very easy. After the structure of the classes had been built, the physics engine had to be implemented, this proved to be the most complex part of the entire dissertation. While the `gl-matrix` library is very efficient, the syntax is very different than a more traditional approach, it eschews assignment statements almost completely, instead the destination matrix or vector is passed as a parameter to the function, for example:

```
vec3.add(vec1, vec2, dest)
```

On its own this is fairly simple to understand, although it can still often lead to confusion over initialising the destination vector. Unfortunately, the syntax makes it incredibly hard to code more complex operations, like the kind needed for a physics engine. A good example of just how complex these operations can be is found in the Runge-Kutta integrator, instead of:

```
state.position += 1/6 * dt * (a.velocity + 2.0f * (b.velocity + c.velocity) + d.velocity);
```

The syntax makes it necessary to have:

```
vec3.add(state.position, vec3.scale((vec3.add(a.velocity, vec3.add((vec3.scale(vec3.add(b.velocity, c.velocity), 2.0)), d.velocity))), 1.0/6.0 * dt));
```

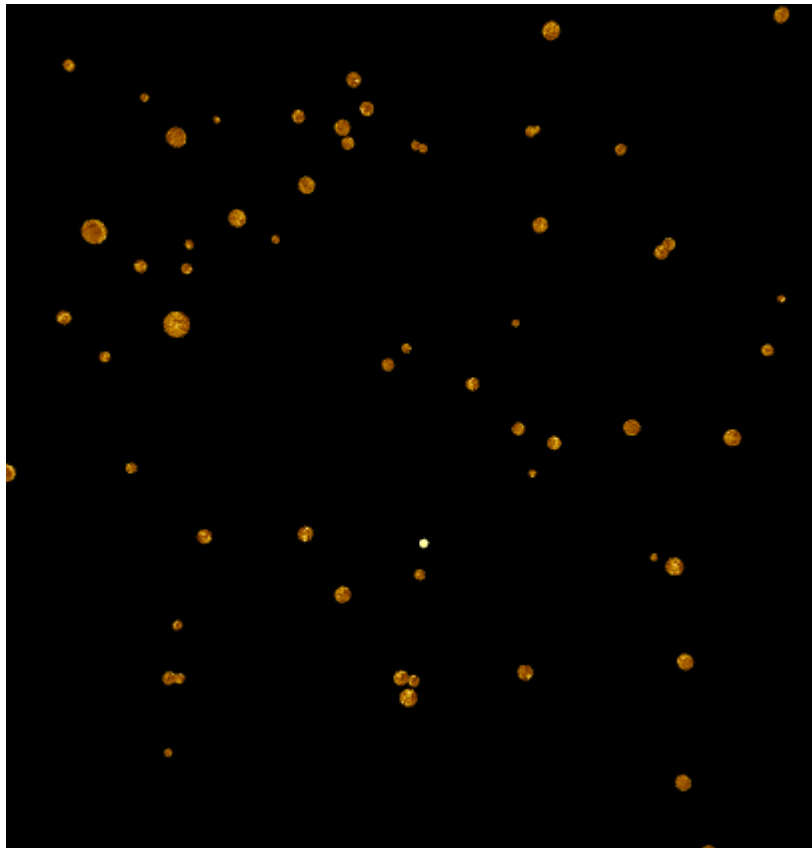
The complexities of coding in this format led to a very large number of bugs, significantly slowing development. Development was slowed even more however, by the limits of using the chrome browser JavaScript console as a debugging tool. The combination of the syntactically difficult `gl-matrix` library and the problematic JavaScript console led to approximately a month of development time being spent debugging, this significantly impacted how far the project went.

The main problem came from a few improperly coded `gl-matrix` function calls returning NaN, normally

this would not be a problem, but due to the large number of function calls needed to update a single variable, and the way most variables interact with each other, it was very hard to pinpoint which functions were at fault. When trace statements were added to the code to see where the problems were, the JavaScript console outputted them in such a way that it showed the current state of the variable, not the state at the time the trace statement was reached, this led to some strange behaviour. For example, if position was set to (1,0,0), and the very next line was a trace statement, the output would be NaN. This was not a product of incorrect initialization, as reading the value of a different, unused, but initialized vector would output the correct value.

Many methods were used to try and solve the problem, as mentioned previously, it took as long as a month to find the bugs, with the eventual solution incorporating flooding the entire code with trace statements, changing the trace statements to output a specific part of the vector instead of the whole vector object, commenting out each function at a time, and stopping the simulation from doing more than two frames a run.

Eventually, the bugs in the physics engine were found and removed, allowing development to start again, the only other notable problem came from WebGL handling textures improperly, if the background clear colour of the context was black, textures would not load after any length of time, showing only black, meaning they would not show up against the background. Once the background was changed to white, while the objects were initially black, the textures would load and be shown properly, after this, the background colour could be switched back to black.



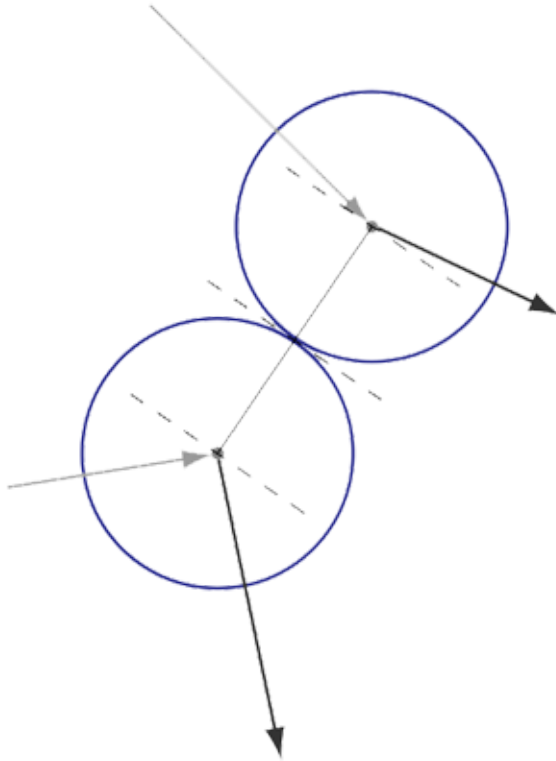
3.3.2 Handling collisions

Collision was added fairly late to the project, as in the initial plan, there were no concrete plans for the interactive part of the simulation, other than that the player star needed to affect the other stars, and vice versa. When that part of the implementation had been reached, it was noticed that the stars did not collide in a realistic manner. Most of the time, they behaved correctly, slingshotting past each other, but occasionally, they would pass through each other, it was decided that this was unrealistic, and a better approach was needed.

The first problem was to detect when a collision occurred, ordinarily, this would have involved each star iterating through all the other bodies, and for each one, calculating the distance between it and itself, then checking to see whether that distance was smaller than the sum of their radius. This would have taken a large amount of CPU time, and been a significant overhead without a more elegant algorithm, but luckily, the equation to generate the gravitational force already requires distance to be worked out, so when calculating gravity, a simple if statement can detect a collision.

Once a collision had been detected, the way to handle it had to be decided upon. The first method implemented had that when the stars collided, their velocity and momentum were negated, causing them to bounce off each other, this looked plausible when they collided head on, but when both were heading in a similar direction it looked very unrealistic.

A fix to this was considered, it involved finding the tangent to the two bodies, calculating the angle at which a body hits that tangent, then reflecting that angle. This is done by subtracting the angle from two times the tangent. Then, the speed of the two bodies is exchanged [20]. The result is much more realistic than the previous method, it looks similar to two snooker balls colliding. While this realistically models how some bodies collide, the project is simulating stars, and stars do not collide like



this.

Another method trialled involved the star with the smaller mass being absorbed by the star with the larger mass, adding its mass and size to the larger one. The direction the new, larger star is heading in being the result of a vector addition of the two original directions. The effect of this is that the two stars collide into each other, and merge. This is a very rare phenomenon, but happens more often in very high density clusters. Unfortunately, in the implementation, this ends up with the strange visual effect of the stars “popping” into each other, looking very unrealistic.

A slight tweak to that method sorted the problem by using a novel technique that simulates the mass transfer existent in close binary stellar systems [22], the pseudocode below shows how the technique works

//intersectionRatio: the percentage of the distance between the two bodies that is intersecting

```

if star.mass is greater than otherStar.mass then
    star.mass += (otherStar.mass * intersectionRatio);
    otherStar.mass -= (otherStar.mass * intersectionRatio);

    star.size += (otherStar.size * intersectionRatio);
    otherStar.size -= (otherStar.size * intersectionRatio);

    if otherStar.mass is greater than 0.1 then
        removeStar(otherStar);
  
```

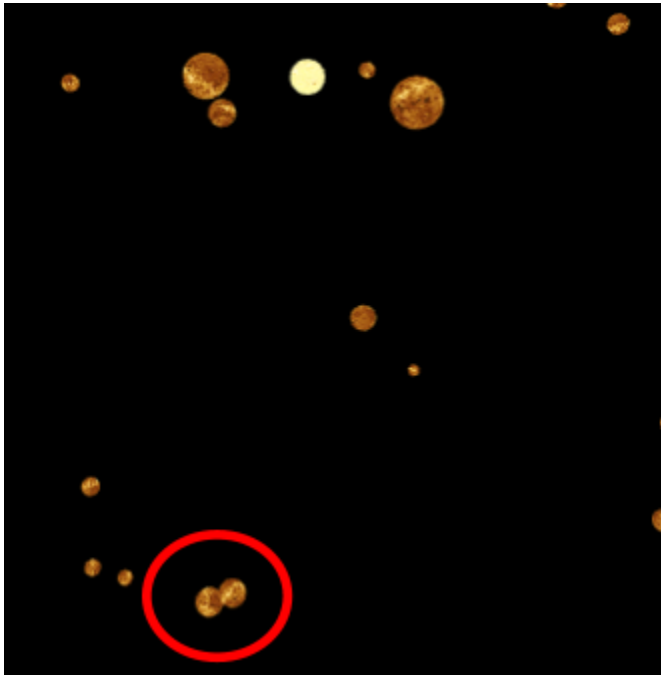
```

else
    otherStar.mass += (star.mass * intersectionRatio);
    star.mass -= (star.mass * intersectionRatio);

    otherStar.size += (star.size * intersectionRatio);
    star.size -= (star.size * intersectionRatio);

```

As the stars begin to intersect, a percentage of the mass (and size) is transferred from the star with the lower mass, to the one with the higher. If its mass is too low, the star is removed from the simulation, having been effectively “eaten” by the more massive planet, this is to stop stars taking part in the force calculations of other stars when they do not have enough mass to have a meaningful impact on the simulation, and might even be too small to be shown on screen.



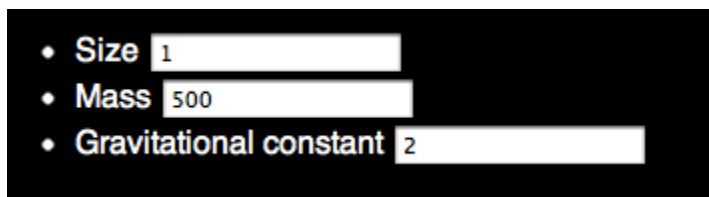
The final approach that was considered involved both stars being destroyed totally by the collision and creating multiple planetoids, with their total mass being the sum of the two parent planets. The planetoids would have been given random momentum, in random directions, with a position close to the collision, they would be given a random angular momentum. This approach was not implemented for several reasons. Firstly, this approach is more similar to planetary collisions, not the stars that the game is currently simulating. Secondly, it was thought that the bodies were likely to re-collide, creating ever smaller planetoids. Having to calculate the gravitational forces for more and more bodies would slow the simulation down considerably. Lastly, it was felt that having the planetoids move randomly was not in keeping with the projects goal of having a realistic simulation, and using smoothed particle hydrodynamics to calculate an accurate situation after the collision was beyond the scope of the project.

3.3.3 Making the simulation more modifiable

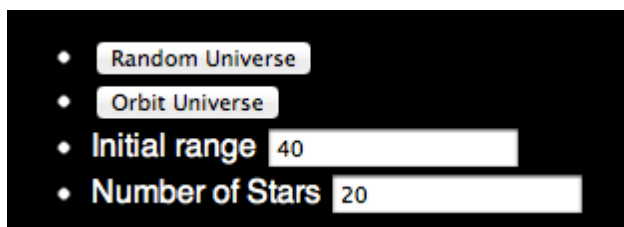
Once collision had been implemented it was decided that the simulation would be more useful

to astrophysicists, and more interesting as a game, if the user could modify variables, such as the gravitational constant, mass of the player controlled star or the initial number of stars in the system. As the game stood, the only thing the player could modify was the force acting on the controllable star, and this would indirectly affect the other stars, while technically this was all that was required in the objectives, it was not deemed controllable enough, and as such, more ways for the user to modify the simulation were added.

This was approached by leveraging the in-browser nature of WebGL, text-boxes were added to the web page that would allow user input. Unfortunately, it was not practical, and in some cases not possible, to have the value in the text box directly linked to the variable in the code. Not practical, because browser DOM lookup is very slow, and if every star had to get a value needed for a calculation every timestep, the simulation would run very slowly. In some cases, it was not possible, for example, the size of the player star was initialized at run time. To overcome these problems, a two step process was used. First, jQuery was used to link the value in the text box to a JavaScript variable, and to handle change events. Secondly, an engine object was created in the util class. The engine object holds the set of modifiable variables as attributes, and their initial values. This two step process meant that it was possible to set variables needed in the simulation to the corresponding value in engine. This overcame the DOM lookup problem, as simulation variables just looked up the engine value, and the initialization problem, as the engine attributes had initial values coded in.



After this had been coded, other ways to modify the simulation were built in. It was felt that the user needed a way to reset the simulation other than refreshing the page. More text boxes were added, so the user could change the initial conditions of the universe. Two initial universe styles were created; Random, and Orbit. The number of stars created was determined by the user. In the Random universe, stars were created with random mass, size, momentum, angular momentum, and position. Their position was constrained by a range input by the user. In the Orbit universe, stars were given position and momentum such that they would initially orbit the player, they had uniform size and mass.



3.4 Summary

The design sub-chapter details the process followed during the design of the project, it discusses:

- Why the code structure was designed the way it was
- The technologies used, and the reasons why
- The prototyping stage, and the lessons learnt from it

The implementation sub-chapter discusses:

- Coding the physics engine, and the difficulties faced
- The different methods of handling collision that were trialed
- What had to be changed to allow the simulation to be modified by the user

4 Evaluation

4.1 Introduction

This chapter details how the project was evaluated, the testing that took place, and the findings from that testing, as well as a discussion on those findings. It also discusses how well the project has fulfilled the objectives.

4.2 Findings

Testing involved seeing what effect increasing the number of bodies simulated had; changing the 'n' of the n-body problem. All testing was done using a Macbook Pro with the following specifications.

Operating System	Lion 10.7.3
Processor	2.4GHz Intel Core i5
Memory	4GB
Hard Disk	750GB
Graphics	Intel HD Graphics 3000 384 MB Integrated

During testing, the only applications open were Google Chrome, with no other tabs open, and Activity Monitor. Activity Monitor is a system application used for task management on the OS X operating system. It allows the user to view a process's CPU and memory usage, as well as quit running processes. It was used to snapshot the CPU usage of Google Chrome during testing.

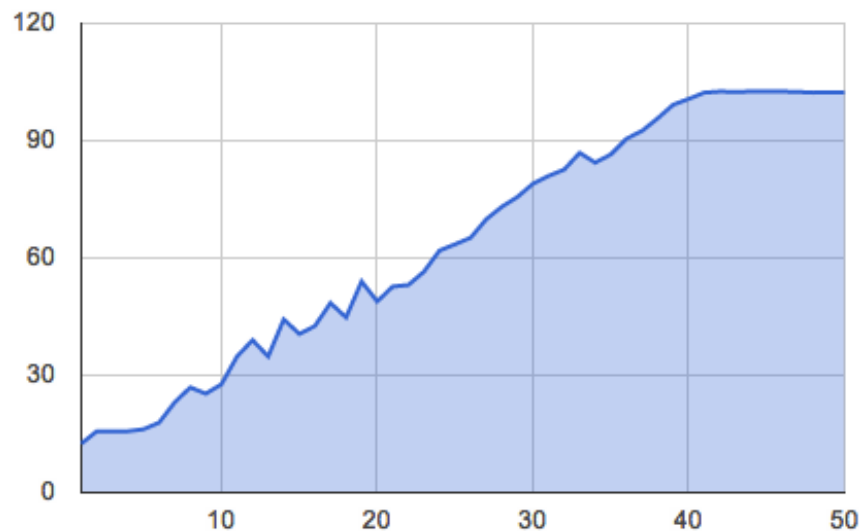
4.2.1 CPU Usage

Testing involved opening a Chrome window, opening a tab with the project webpage <http://philellwood.com/philftp/dissertation/>, setting the number of bodies, starting the simulation and then checking the CPU usage with Activity monitor. To establish a baseline, the first test was done with only the player star, this meant there were no other gravitational interactions.



PID	Process Name	User	% CPU	Threads	Real Mem	↓
489	Google Chrome Helper	phil	16.5	4	64.0 MB	I
14814	Google Chrome Renderer	phil	12.5	5	88.3 MB	I
465	Google Chrome	phil	7.7	39	196.1 MB	I
3223	Activity Monitor	phil	0.9	2	32.4 MB	I
141	SystemUIServer	phil	0.2	4	45.0 MB	I

Here, the renderer is using 12.5% of the CPU. The renderer is the thing that processes the JavaScript, so it was the process that was tested. New stars were added to the simulation one by one, and the CPU usage of the renderer was noted down. As the number of stars got towards 50, the renderer was





continually using around 100% of the CPU, so the testing was stopped there. The results were put into a graph for easier evaluation.





The CPU usage goes up fairly uniformly, and then levels out. It should be noted, that the last few tests show as using over 100% of the CPU.

PID	Process Name	User	% CPU ▾	Threads	Real Mem
14814	Google Chrome Renderer	phil	102.4	5	98.7 MB
489	Google Chrome Helper	phil	11.3	4	70.3 MB
465	 Google Chrome	phil	3.2	27	196.8 MB
3223	 Activity Monitor	phil	1.0	2	32.8 MB
15019	Screen Capture	phil	0.4	2	3.3 MB

This is due to the multi-core nature of the hardware, and the use of hyper-threading, other tests showed as high as 144.6%.

	Google Chrome Renderer	phil	144.6	5	90.6 MB
	Google Chrome Helper	phil	11.0	4	55.8 MB
	Activity Monitor	phil	1.2	2	26.9 MB
	Preview	phil	0.9	4	65.1 MB
	Finder	phil	0.1	6	29.8 MB

To check that the levelling out at 100% was not an anomaly, tests were carried out with 2000 bodies, this resulted in the same level CPU usage (but an incredibly low framerate).

PID	Process Name	User	% CPU ▾	Threads	Real Mem
14814	Google Chrome Renderer	phil	100.6	5	102.1 MB
489	Google Chrome Helper	phil	0.9	4	71.6 MB
3223	 Activity Monitor	phil	0.8	3	36.7 MB
465	 Google Chrome	phil	0.1	27	195.7 MB

Referring back to the graph, while there are small peaks, it is very linear until it levels out. It levels out fairly early compared to some other n-body simulations, this is probably due to using JavaScript, as it is an interpreted language.

With more time, it would have been interesting to compare different algorithms. Comparing the Particle-Particle method to the Barnes-Hut algorithm to see the performance gains would be enlightening. Another test that would have been completed with more time, would be seeing how not modelling the rotation of bodies affected the overall performance. In the current simulation, modelling rotation uses lots of matrix transforms, and while interesting and more realistic to look at, it does not affect the gravitational forces. Removing rotation would probably allow for more bodies to be in the simulation at once.

4.2.2 JavaScript CPU Profile

The Chrome JavaScript console has a JavaScript CPU profiler built in. This is a very useful feature, and allows the developer to see where bottlenecks in the code are. This is important, as the project makes major use of the CPU, and see which functions use it most, or are the most inefficient. Once the bottlenecks have been found, it would be possible to find ways of making them more efficient.

Self ▼	Total	Function
56.84%	56.84%	(program)
1.84%	1.84%	(garbage collector)
0.22%	41.31%	▼ tick
0.28%	35.62%	▼ integrate
0.44%	33.77%	▼ evaluate
3.23%	24.39%	▼ forces
5.25%	19.51%	▼ vec3.create
14.26%	14.26%	Float32Array
0.81%	0.81%	Float32Array
0.35%	0.35%	vec3.scale
0.29%	0.29%	vec3.add
0.20%	0.20%	vec3.negate
1.14%	3.98%	▼ Star.Derivative
2.84%	2.84%	Float32Array
0.64%	4.73%	▼ State.recalculate
1.27%	1.27%	mat4.translate
0.24%	0.81%	► quat4.create
0.22%	0.22%	mat4.multiply
0.20%	0.75%	► quat4.toMat4
0.11%	0.11%	mat4.inverse
0.09%	0.09%	quat4.multiply
0.07%	0.79%	► mat4.create
0.06%	0.06%	mat4.identity
0.15%	0.15%	vec3.scale
0.04%	0.04%	http://philellwood.com/philftp/dissertation/util.js
0.02%	0.02%	mat4.multiply
0.02%	0.02%	vec3.add
0.29%	1.52%	► State.recalculate
0.02%	0.02%	forces
0.02%	0.02%	vec3.scale

The profiler shows where the major bottlenecks in the program are very well. `tick` is called every frame, and is used to step the simulation forward. `integrate` is called on every star in the simulation. `forces` is the function that sums the gravitational force of every other star, and also handles collisions. From the profile, we can see that `tick` uses the most in total, 41.31%, this is because it is used to start the other functions. It only uses 0.22% to run itself. This is similar to `integrate` and `evaluate`, they both use little CPU, but call the next function in the chain. At `forces`, we see that it has a slightly larger overhead, but still, most of the CPU is used by a function it calls: `vec3.create`. `vec3.create` has 19.51% of the CPU used, this is almost half (47.2%) of the amount of CPU used by `tick`. This means that almost half the CPU usage of the simulation comes from creating vectors. This should be streamlined to be more efficient.

The problem comes from the fact that `vec3.create` creates a `Float32Array`. JavaScript does not normally use `TypedArrays`, but they are necessary in WebGL. WebGL expects arrays passed to it to be in the form of a `TypedArray`. `TypedArrays` have other benefits too, array indexing is already very fast in JavaScript, but it is even faster with `TypedArrays`. The profiler shows that the overhead in creating them is significant, so the throwaway object creation in `forces` is very inefficient. On the other hand, removing that object creation would take away from the readability of the code, and might slow the application down in a different way.

4.3 Fulfillment of objectives

The project aim was split into objectives, which were then further divided into sub-objectives:

Become Proficient with WebGL

Make a prototype without any physics

This sub-objective was fulfilled with the prototype made as part of the design process, it was discussed in 3.2.3 Prototyping. Developing this prototype led to a greater understanding of WebGL and many important design reconsiderations were made because of it.

Research existing methods for simulating orbital mechanics

Research how to do basic physics simulation

This sub-objective was fulfilled through the research carried out in chapter 2. Research was carried out on how basics physics works, progressing to how it is commonly implemented in games and simulations.

Find information on planetary motion

While information was found on planetary motion, and discussed in chapter 2, it was not explicitly coded in. Once the physics engine and the gravitational forces had been coded in, realistic planetary (or stellar) motion came naturally. This objective was perhaps naively created from a lack of understanding about how gravitational forces work. The research carried out on this objective showed that it was redundant. If the project was repeated, this sub-objective would have been removed.

Find equations relating to the n-body problem

Much like the last sub-objective, though this sub-objective was completed, it was unnecessary. The research that was carried out relating to the n-body problem, once it was realised that the objective was poorly decided, moved more to finding how other n-body simulations were implemented.

Port the equation into code

Again, this sub-objective was decided upon with a poor understanding as to what needed to be done in the project. Instead of "porting the equations into code", methods used in other n-body simulations were implemented in JavaScript. It could be said that this objective was not fulfilled, but there were no equations to port. "Implement a method for solving the n-body problem" would have been a better objective.

Learn methods for increasing efficiency of equation

This sub-objective was fulfilled through the research carried out in section 2.4, methods that increased the efficiency of the particle-particle method, such as particle-mesh and the Barnes-Hutt were researched in depth, though there was not enough time to implement one.

Create basic gravity simulation

Create celestial body object with needed properties

This sub-objective was fulfilled with the basic structure discussed in the design chapter under section 3.2.1 Structure. The properties were the ones discovered in the research chapter, then implemented following the design discussed previously. The final implementation used two classes to represent the

object, instead of the one class that was thought sufficient when the objectives were written.

Create methods for updating those properties

This sub-objective was fulfilled at the same time as the previous one, as both properties and methods were implemented in one go. This possibly led to the problems discussed in 3.3.1 Implementing physics. Instead of adding properties and methods simultaneously, it may have been wiser to test each method individually, though this would have been hard with the way the various methods in the physics engine interact with each other.

Populate game with many celestial objects

Once the problems discussed previously had been addressed, it was a trivial matter to add more stars. This fulfilled the sub-objective and let it be possible to view how changes to variables affected the system.

Develop simulation into a full, player controlled game

Create player object, with methods for movement

Once a general celestial body object had been created, it was simple to create a player object that inherited those properties and methods. Then, keyboard input was linked to a method which added a force in the direction the user had pushed, fulfilling this sub-objective.

Make player effect other celestial objects

Because the player object was also a celestial body object, it had the methods which made it a part of the n-body simulation. As a result of this, no additional work was needed to fulfil this sub-objective.

4.4 Summary

This chapter is the evaluation of the project, both through testing, and a discussion on how well the objectives were fulfilled.

- Testing how changing the number of bodies affects the CPU usage
- Using a profiler on the code to find the bottlenecks
- Discussing how successfully the project achieved its objectives, and whether they were appropriate

5 Conclusions

5.1 Introduction

This chapter reflects on the project overall, discussing what has been learnt and how the project might differ if it was done again. Also discussed is what is left to be completed in the project, and ideas on where it could be extended in the future.

5.2 What has been learnt

This project has been a great learning experience. Learning how to setup a basic WebGL application has been invaluable, I have already used it in other assignments and I look forward to furthering my skills and developing even more projects. As well as WebGL, I've also learnt how to code an effective, efficient and accurate physics engine. I plan to re-use the code again in more complex games, and other interesting physics simulations.

In addition, I have learnt more about debugging using the Google Chrome JavaScript console, which should be useful for any web development. Indeed, due to the number of problems in the project, I have become a lot better at debugging in general.

5.3 If the project could be redone

If the project was to be redone with the knowledge learnt from this one, significant changes would be made. As discussed in the evaluation, the objectives and sub-objectives were set somewhat naively, this was also mentioned in the feedback from the project proposal. The objectives were set without any real knowledge of how physics is implemented in computer games and simulations. Instead of concentrating on how solutions to the n-body problem have been attempted mathematically, a research objective would have been looking at how previous simulations were attempted instead.

Another aspect of the project that would be changed is how the physics engine was implemented. As discussed previously, it was all coded at once, and this contributed to the major problems that occurred and slowed development considerably. If done again, the project would implement unit tests, testing each function as it is coded. While this would be slower to code initially, it would hopefully result in fewer bugs, and avoid the problems that happened in this project.

5.4 Project completion

One thing that did not happen in the project was studying how different methods used to approximate a solution to the n-body problem would effect the simulation. This was due to time constraints as a result of the problems faced during the project. With more time, methods such as the barnes-hutt algorithm, Particle-mesh simulation, kd trees and possibly the symplectic method would have been implemented and compared. The method that allowed the most bodies to be simulated, while still dealing with phenomena such as collision, would have been implemented.

5.5 Future work

While the project did not reach as far as it could have, it still allows for significant interesting future work. One particular avenue that was suggested was instead of simulating gravity, a different force could be used. As the physics engine is force driven, it is not necessarily limited to simulating physics. A "force" could be the attraction a consumer has to different products. It would take a lot of work, but it could be possible to model what products a set of users would buy from a shop, and how different shelf placement, or web design effect users purchasing decisions.

From a more game design perspective, the project could be extended to be used as a way to have more realistic movement in a larger space game, allowing for players to slingshot round large planets to get a speed boost. Away from space, and not using gravity, as discussed in the previous paragraph, the project could be modified into a game AI system. The system could handle a large number of entities and model an AI entity's wants and emotions.

It has been mentioned throughout the dissertation that the project could be used as a tool for astrophysicists. As it stands, the simulation is only modifiable in a trivial sense. A GUI for adding stars and planets, at specific positions, with specific mass, velocity and angular velocity would allow the tool to be useful in a scientific field.

This project has strived to be as realistic as possible at all times, however, two major simplifications were made. The decision to have the objects act as rigid bodies allowed linear motion to be split from rotational motion in the simulation. This made the calculations much simpler, but in real life, celestial bodies are not rigid bodies, and angular velocity does, very slightly, effect linear motion. Almost all simulations make this simplification, but to achieve the most realism, this would have to be changed. The second simplification was the decision to discount general relativity, this would only effect the simulation at speeds that were a significant percentage of the speed of light, but again, to achieve the most realism, it would have to be implemented.

The most desired piece of future work would be running the physics engine on the graphics card. The next positions of every particle in the simulation could be calculated in parallel, and then stored in a texture. This technique would utilise the GPU to the fullest, taking away the CPU as a bottleneck and allowing more bodies to be used in the simulation. One downfall of this approach would be handling collisions, and being difficult to change the number of bodies simulated. In the end, general purpose computing on the GPU is very difficult, and was beyond the scope of this project.

References

- [1] <http://www.opengl.org/wiki/FAQ>
- [2] <http://www.khronos.org/>
- [3] <http://caniuse.com/webgl>
- [4] <http://blogs.technet.com/b/srd/archive/2011/06/16/webgl-considered-harmful.aspx>
- [5] <http://www.khronos.org/news/permalink/webgl-security>
- [6] <http://www.kamibu.com/demos/cloth-simulation/>
- [7] <http://cedricpinson.com/ggj2012/>
- [8] L. Euler, *Institutionum Calculi Integralis, Volumen Primum*, Opera Omnia vol. XI., 1768.
- [9] <http://gafferongames.com/game-physics/physics-in-3d/>
- [10] Newton, *Philosophiæ Naturalis Principia Mathematica*, 1687
- [11] Kepler, *Astronomia Nova*, 1609
- [12] http://parlab.eecs.berkeley.edu/wiki/_media/patterns/n-body_pattern_language9-09.pdf
- [13] J. Barnes and P. Hut, "A hierarchical $O(N \log N)$ force-calculation algorithm", *Nature* 324 (4): 446–449, December 1986
- [14] philellwood.com/webgl
- [15] <https://github.com/toji/gl-matrix>
- [16] http://glmatrix.googlecode.com/hg/benchmark/matrix_benchmark.html
- [17] <http://paulirish.com/2011/requestanimationframe-for-smart-animating/>
- [18] <https://developer.mozilla.org/en/DOM/window.requestAnimationFrame>
- [19] <http://www.w3.org/TR/animation-timing/#requestAnimationFrame>
- [20] <http://www.petercollingridge.co.uk/pygame-physics-simulation/collisions>
- [21] Leonard, P. J. T., "Stellar collisions in globular clusters and the blue straggler problem", *Astronomical Journal*, vol. 98, 217-226, July 1989
- [22] McCrea, W. H. "Extended main-sequence of some stellar clusters", *Mon. Not. R. Astron. Soc.* 128, 147–155, 1964

