CO600 Final Year Group Project

RESEARCH INTO THE SIMULATION OF SHOCK WAVES

Technical Report

Georgina Perera
School of Computing
University of Kent
Canterbury, Kent
UK, CT2 7NY
gp221@kent.ac.uk

Robert McDonnell
School of Computing
University of Kent
Canterbury, Kent
UK, CT2 7NY
rm506@kent.ac.uk

Michael Jones
School of Computing
University of Kent
Canterbury, Kent
UK, CT2 7NY
msj4@kent.ac.uk

Liam Ireland
School of Computing
University of Kent
Canterbury, Kent
UK, CT2 7NY
lsi6@kent.ac.uk

Tomasz Mackow
School of Computing
University of Kent
Canterbury, Kent
UK, CT2 7NY
tm378@kent.ac.uk

Abstract

This report details the processes, findings and outcomes of our research and investigation into creating a realistic shock wave simulator. It covers different models of representing shock waves and the success of each mode. The most successful model was the particle model as it produced the most realistic looking output. This report concludes that accurately simulating shock waves is computationally expensive and not viable in real-time gaming applications.

Introduction

In video games, the simulation of shock waves is often fudged. They tend to use a computationally cheap technique in which a large explosion occurs instantaneously rather than over time. The reason for this is that video games need to maintain a high update rate, and including complex physical simulations is often not an option due to it being computationally too expensive. This report details our investigation into whether representing physically realistic shock waves for real-time applications is an achievable feat.

Aims

The main aims of this project were to:

1. Discover how shock waves could be realistically simulated in real time.
2. Answer the question as to why shock wave calculations are often fudged in video games.
3. Implement our own realistic shock wave simulation system.

Further details about the aims of this project can be found in the Requirements document.[6]

What is a shock wave?

A shock wave is a sudden change in pressure travelling through a medium (e.g. air) caused by an object moving faster than the speed of sound (i.e. greater than Mach 1).

Shock waves occur because "When the speed of a source equals the speed of sound [...] the wave fronts cannot escape the source. The resulting pile of waves forms a large amplitude 'sound barrier'".[1] This can be seen in Figure 2 where the build-up of waves forms a shock wave in front of the object moving faster than the speed of sound.

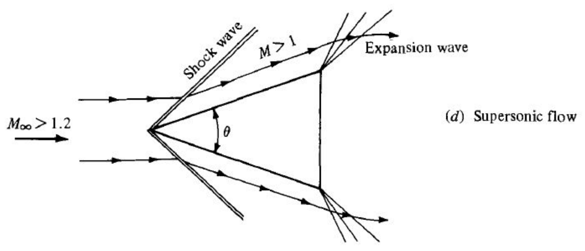

Figure 1: Shock wave formed in front of an object[8]

An example of a shock wave occurring in a real life situation is when a jet breaks the sound barrier. A shock wave is caused because the jet keeps up with the sound waves which it is emitting, causing a build up of the sound waves.

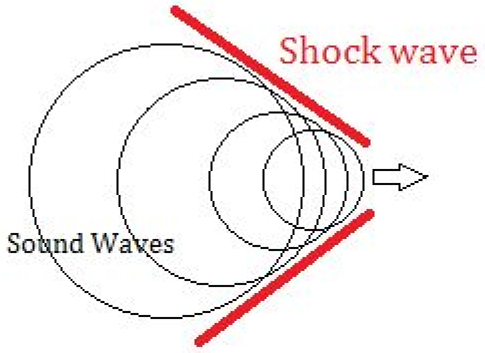

Figure 2: A shock wave caused by an object traveling faster than the speed of sound.

As the jet accelerates further it begins to overtake previously emitted sound waves causing a trail of sound waves which overlap. As these waves were emitted in a sphere outwards from the jet, there are points which build-up significantly more causing a shock wave. A visualisation of this can be seen in Figure 3 below.

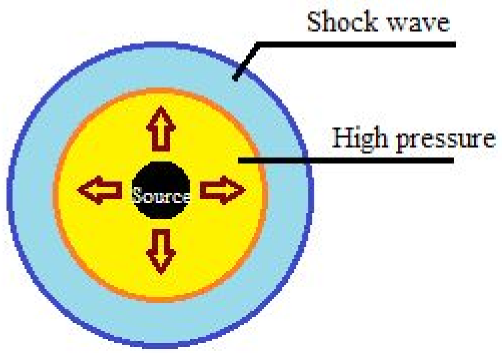

Figure 3: Diagram of a detonation

Shock waves can be caused by large explosions called detonations. In this case they are caused by a violent and sudden change of potential energy to work. A large amount of gas, sound and heat is rapidly produced by rapid combustion which causes a large increase in pressure. This pressure causes an outward acceleration of air faster than the speed of sound, causing a shock wave to form in front of it. A visualisation of this can be seen in Figure 3. Our project focuses on simulating shock waves caused by detonations.

Properties of a shock wave

Before attempting to implement anything, it was decided that time would be spent on researching how shock waves are currently implemented in games as well as researching the affects and behaviours of shock waves. This investigation covered what the physical effects of a shock wave are and how they interact with their environment.

For a wave to be a shock wave it needs to be traveling at a speed faster than the speed of sound (Mach 1) in any medium. If it drops below this speed, it is classified as a sound wave. As this project is focused on modelling shock waves, the decision was made to not put any time into modelling the transition from a shock wave to a sound wave.

For the more intricate details and physics equations of the shock wave, various resources such as the Fundamentals of Aerodynamics[2] and the Feynman Lectures[3] were consulted. These resources were used when building the implementation to try to make the shock wave seem as realistic as possible. For example, when calculating the pressure ratio between the shock wave and normal atmospheric pressure, the following equation is used, where p_1 is normal atmospheric pressure, p_2 is shock pressure, M is the Mach number (speed of the shock wave) and γ is the heat capacity ratio of the medium:

$$\frac{p_2}{p_1} = 1 + \frac{2\gamma}{\gamma + 1}(M_1^2 - 1)$$

Figure 4: Ratio of normal atmospheric pressure and shock pressure

Another property of shock waves is the energy loss over time. The velocity of a shock wave should reduce as time goes on because of energy loss due to factors such as heat, sound etc. When a wave front hits a surface it should also reflect with a certain percentage of the energy being absorbed depending on the surface it came into contact with. A further consideration was wave diffraction. Like other waves, when shock waves meet at obstacle or travel between two objects, it should diffract similar to light traveling through a slit.

The Unity Engine

In the interest of keeping to the given time frame, a pre-existing game engine as a platform for us to develop on. Following this, the decision on existing game engines was narrowed to the Unity Engine and Unreal Engine. After some further investigation, it was decided the Unity Engine would be more suitable. Unity is .NET based which means our source code would be written in C#, which is very similar to Java, a language the group were all familiar with. In contrast with developing in the Unreal Engine, which would be C++, which the group have limited experience with. Additionally, after completing tutorial videos on the editor software for both engines, Unity was found to be easier to use. Some features which also attracted us are the fact the Unity is open-source and has a well-documented API.

The general structure of Unity is that each scene is populated by entities called Game Objects. Everything that can be added to a scene is classed as a Game Object, and scripts can be applied to Game Objects. A Game Object with a rigid body attached can have forces applied to it, and Unity handles object collisions with its own physics engine.

There are two types of physical objects, those with rigidbody scripts attached that can be moved by applying forces, which we'll call dynamic objects, and those that can't be moved which do not have rigidbody scripts attached, which we'll call static objects. Unity also has many useful functions which are taken full advantage of such as Raycast and Linecast. The API is well-documented and it is available on the Unity website.[5]

Developing a physics engine

At the beginning of our project, some effort was put into creating our own basic physics interactions instead of using Unity's inbuilt physics engine. To be in complete control of the physical interactions in the world, our own physics would need to be used instead of the black box that is Unity's physics engine. However, after two weeks of development the decision was made that this route would take far too much time to implement correctly and creating a physics engine should not be one of our goals. So instead it was decided to focus our research and development purely on the shock wave and let the Unity Engine handle basic interactions between objects.

Modelling a shock wave

The first half of our project consisted of spending our time researching into different shock wave models which were as follows:

Sphere

The first attempt at implementing a shock wave involved creating an expanding sphere that when it came into contact with moveable objects, it would exert a force on the object. However, this implementation would not suffice as a sphere would not let the shock wave act in a realistic enough manner as it was not malleable enough. For example, it would have made diffraction and reflection difficult.

Mesh

Another solution that was considered was using meshes. This would have involved mapping out the shape of the wave front in a way that would have actively morphed as it hits objects and walls. While this solution seemed plausible, it proved to be very difficult to implement as the group had limited understanding of how to use and interact with the meshes offered to us in Unity. This was complicated further as the size of the mesh in reflections and diffraction would be distorted and needed to be tracked so it was decided visually representing this mesh would be extremely difficult. The reasons for not implementing meshes are similar to that of the issues faced when implementing the spheres model. This being that diffraction and reflection would have been very difficult to track properly and the visualisation in general would have not produced a good result because the shape of the wave would have been lost after just a few reflections.

Smooth Particle Hydrodynamics

Smooth particle hydrodynamics was another candidate solution for our wave simulator, however, the decision was made not to implement this. Smooth particle hydrodynamics works by dividing a fluid, or in our case a wave front, and splitting it up into a set of individual components called particles. The particles are related in such a way that their properties, such as velocity or temperature, are obtained by summing the values of all the particles which are within a specified range. The influence of each particle in this range is weighted based on their distance from the selected particle as well as their density. A function known as a kernel function is used to make this more computationally efficient by excluding particles which are very far away thus offering little contribution to the value being calculated. Due to the complexity of implementing such a system with our time constraints, the decision was made not to pursue an implementation of smooth particle hydrodynamics.

Particles

The most successful solution, and the one which was chosen for the project, is the particle model. Particles offer us a way to expand the wave in a sphere-like manner and also give us the malleability needed to allow the shock wave to diffract around walls.

Each particle represents an area of the wave front proportional to how many particles there are. So if the wave front is a sphere, each particle will represent the area of the sphere divided by how many particles there are. When a particle hits an object, it should emit a force onto the object in the same direction and proportional to the velocity of the particle.

In order to represent a realistic shock wave, each particle needed to act in a realistic manner. For example, particles needed to reflect off of walls to simulate the wave front being reflected and losing some of its velocity.

Particle shock wave system

Requirements

The chosen particles simulation model required that the system represent a shock wave as a number of point particles, simulating in discrete real-time their motion and some interactions with objects (e.g. forces).

The system must store information about the initial shock wave conditions and the properties of the wave medium given as parameter inputs.

The position, velocity and acceleration must be stored for each particle. The position of each particle must be updated at a frequency such that they appear to have smooth motion. Each particle should be (optionally) rendered as an indication of the particle's location every frame.

The particles should move outwards from a single point at the start of the simulation, in a sphere shape initially. Each particle should start equidistant from one another initially, should have linear motion, bounce off of static surfaces and exert forces on dynamic objects.

Implementation

There were three implementations of the particle shock wave model developed, each improving significantly on the previous in terms of meeting the specified requirements.

Version 1 (V1)

The first implementation (Figure 5) represented each particle as a Unity Game Object with a rigid body and spherical collider so that particle collisions and forces were handled implicitly by the Unity Engine. An impulse was applied to each particle in different directions to form an initial sphere shape. These impulse vectors were calculated such that their angles were spherically equidistant using a simple algorithm detailed by Markus Deserno.[4]

Simulating a high number of spherical rigid bodies proved to be absurdly performance intensive, and by delegating control of the physical simulation to the Unity physics engine, the specific effects of the simulation could not be controlled by our code, and so this method could never meet the exact requirements of the simulation model.

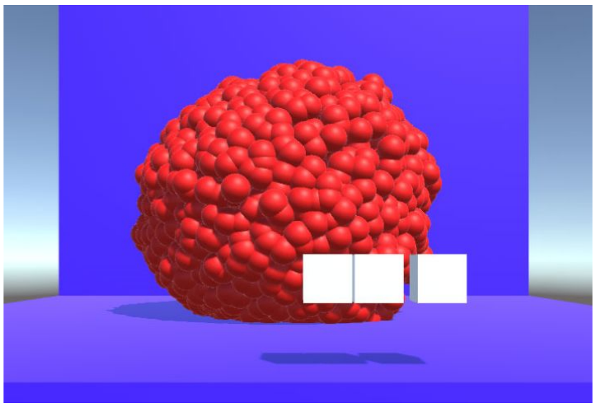
Figure 5: Particle Shock Wave V1

Version 2 (V2)

The second implementation (Figure 6) used a particle system that is part of the Unity Engine package. This implementation also added parameters for the properties of the wave medium used for calculating the force applied to an object upon collision. The following wave medium properties are given as inputs:

- Normal atmospheric pressure
- Heat capacity
- Speed of sound

The Unity particle system was configured to emit a number of particles from an origin coordinate. The system emits particles in random directions (as opposed to equidistant).

The Unity particle system handles collision detection and bounces and reports whenever a particle collides with an object, providing some data about the collision, allowing us to add some additional collision behaviour.

A behaviour was added so that upon colliding with an object, a particle exerts a force on the object in the same direction as its velocity. The magnitude of the force is calculated based on the velocity of the particle and properties of the wave medium.

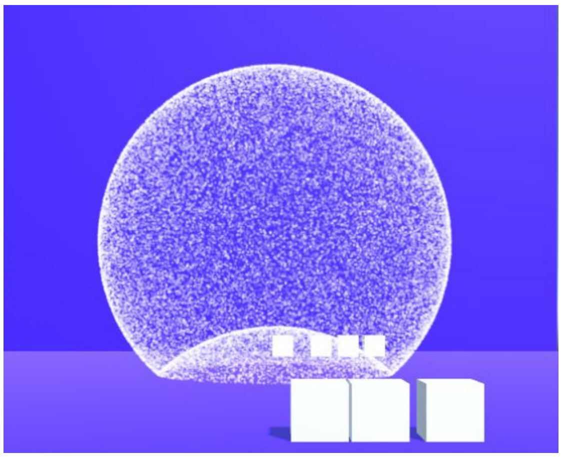
Figure 6: Particle Shock Wave V2

The Unity particle system is intended to be used for visual effects (e.g. smoke) and was deemed not suitable for use in a physics simulation as it and does not provide writable access to particle data at run-time. For example, the velocity of a particle could not be directly modified as required in order to simulate dissipation of energy during collisions. The system also did not have an option to emit particles equidistant to one another, meaning the simulation did not act as a uniform wave front, but instead had random patches of high and low pressure.

Version 3 (V3)

The third implementation (Figure 7) uses a custom particle system designed to meet the specific requirements of the shock wave model, and has input parameters similar to V2.

In this system a particle is represented as a simple object stored in the .NET environment, with the following fields:

- Position coordinates
- Velocity vector
- Deceleration constant

Particles are initially spawned equidistant on a sphere using the same algorithm used in V1.[4]

During each time-step, an update process is executed for each particle. This update process can be summarised as the following:

1. The next position of the particle is calculated based on its velocity and the time delta (since the last update).
2. A line is cast between the current position and next position of the particle. A line cast function that is part of the Unity physics

engine is used which reports any objects intersecting the line.

3. For each object intersecting the line:
    a. If the object is static (e.g. a wall), a new particle position is calculated by reflecting (bouncing) the particle off of the collision surface.
    b. If the object is dynamic (e.g. a table), a force is exerted on the object.
4. The speed of the particle (magnitude of its velocity) is changed based on its deceleration constant.

Forces (exerted on objects during collisions) are calculated in this version in the same way as in the previous version (V2).

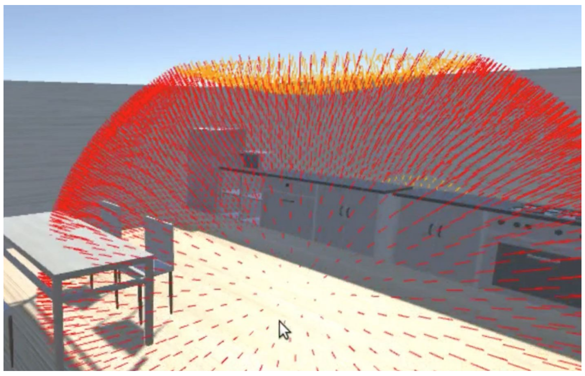
Figure 7: Particle Shock Wave V3

Particle bounces are calculated using data reported by Unity's line cast method. This calculation is visualised in 2 dimensions in the diagram below (Figure 8). This diagram shows the particle's initial position A, calculated next position B, and new position C after reflecting off of the surface.

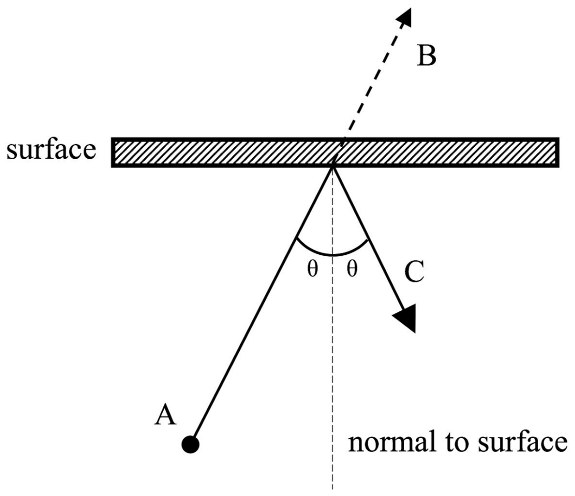

Figure 8: Particle reflection

Position C is calculated such that the distance to C from the point of intersection is equal to the distance to B, and the angle between A and the normal to the surface is equal to the angle between C and the normal to the surface (θ).

The direction of the particle's velocity at C is calculated using a generic 3D vector reflection function provided by the Unity Engine.

Note that by using a line cast, the collision with the surface is detected before the position of the particle is updated. A common alternative is to check whether a particle is intersecting the surface bounds after updating its position, which would not be sufficient for particles travelling at high speed (as is required to model a shock wave) as particles might jump through surfaces during a single time step. For example, in the above diagram, as the particle at position B is not intersecting the surface, the collision would not be detected and the particle would continue in the same direction.

The particles are visualised by a single render call to the Unity graphics library which draws a line between a particle's current position and its position in the previous time step. The line has a colour which represents the pressure of the shock front at the position of the particle. The colour is interpolated between two colours (for high and low pressure) specified as input parameters.

Evaluation

As the output of our system is a visualisation of a shock wave. The best way of testing our system was by comparing our output directly with videos of real shock waves. A variety of detonation sizes were used to ensure that our output was consistent when altering the initial speed.

Our output was compared with videos which involved objects interacting with shock waves. This helped us to see how objects behave when hit with a shock wave. This method of testing was appropriate as our output was synced up with the videos that were found and easily identify anything which stood out significantly.

During the poster fair the opportunity was taken to ask the Kent University staff and students what their opinions were on the realism of our output. This was done by asking them to fill out a questionnaire after interacting with our system. Although only 10 responses were received, the results were positive where 80% of the feedback gave ratings of 4 or higher (out of 5). for the question asking about how realistic our shock wave appeared.

Our testing has shown us that the general motion of the particles as a wave front does appear to mirror how a shock wave would travel, however not all interactions with objects appear completely realistic.

Optimisation

The system has very high computational cost due to the required number of particles (tens of thousands) and frequency of updates (50 updates per second at optimum). Our code needed to be highly optimised in order to achieve anything close to the target update frequency. Run-time analysis using a profiler showed which processes accounted for the most processor time. 3D vector operations (addition, multiplication, etc.) and checks for vector equality were shown to be the most performance intensive processes.

Any vector operations for which the two vectors were at the same angle could be optimised by storing the normalised version (of either of the vectors) separately and converting the vector operation to a numerical operation. For example, the velocity vector stored for each particle was changed so that it is stored as a direction vector and magnitude (speed). In doing so, changes in speed can be calculated as numerical operations instead of vector operations, and the velocity vector can be calculated where required as the direction vector multiplied by the speed.

Due to the system running in a third-party environment (Unity), any optimisations to our code relied on our understanding of the environment's internal processes. For example, our system made heavy use of Unity's 3D vector struct, which appeared to store vector data as a struct with the following members: x/y/z components, magnitude, and a normalized version of itself (with a magnitude of 1). The profiler revealed that accesses in our code to struct members such as magnitude had been converted into method calls by Unity's compiler, and that these method calls had significant cumulative computational cost. Repeated accesses to the same struct member could therefore be optimised by storing the member as a local variable.

An understanding of Unity's garbage collector was also required to ensure that it was not running collections unnecessarily and causing stalls in execution.

Unresolved Issues

At the time of project completion, there remained some unresolved issues with our implementation of the particle shock wave. One of these issues was that the forces applied by the particles appears to be different depending on the time-factor. The faster the time factor, the stronger the forces appear to be. This was a particularly puzzling problem for us as the total amount of force applied by all of the particles in the scene over the duration of one detonation and this force was equal for all time-scales. Another problem was over the course of one detonation, as time went on the forces appeared to be greater the further away they were from the origin of the shock wave. The duration of the project was unfortunately over before these problems were solved.

Front End

Approach

The front end build began slightly after development of the back end had started. This allowed us to perform analysis of the requirements of our entire system; from this the specifics of what would be included in the final product's front end were acquired.

Once these specifics were clearly laid out, sketches of the design were made. During group meetings, designs were altered based on the group feedback.

After initial design features had been agreed upon, the wireframes were created using Axure[7]. The same feedback process as above was used until a final GUI layout was achieved.

Using tools Unity offered, the GUI was built based on the final wireframes. Throughout the project the design strayed slightly from wireframes, due to weekly feedback from our supervisor, the rest of the group and general adaptations for ease of use.

Scene Creation

We began creating the scene, by first deciding on the scenario that would best showcase the different aspects of the wave - reflection, diffraction (functionality later removed) and particle lifetime. A small household interior was modelled as this allowed us to see how the wave moved through doorways and interacted with objects of different masses at different distances.

As creating the objects would have been possible, but very time consuming, the Unity Assets Store was utilised. The most appropriate collection of items were downloaded and dropped them into the scene (Figure 9) in a way which was most aesthetically pleasing, whilst also giving good wave interaction. To create the materials, copyright free images were imported into the project. These could then be placed onto individual objects. Once placed on an object the material was edited by changing the colour shading, brightness, etc until the desired look was achieved.

In order to make the wave have an effect on the objects, two different types of colliders were used; box and mesh. Mesh colliders wrapped to the object mesh as closely as possible and was used for more complex objects (e.g. sink, sofa, desk), whilst simpler objects (e.g. book, fridge) were given box colliders, to give the most accurate wave interaction.

Graphical User Interface

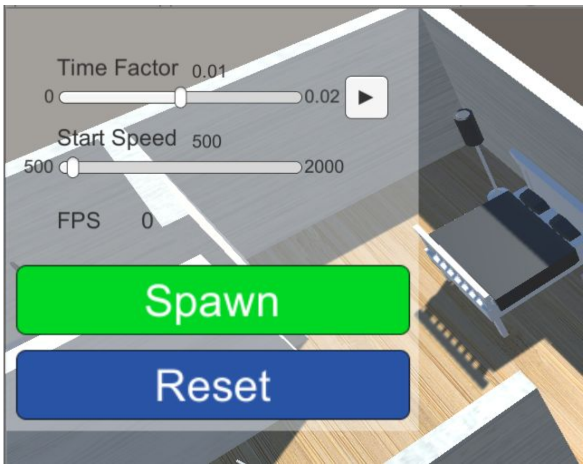

Figure 9 - Capture of the control menu

Some of the features created can be seen in the above screenshot. These features included:

Play/Pause button - The script written for this button was to toggle whether the simulation is running or paused.

Time slider - Due to shock waves travelling at more than the speed of sound, at a normal time frame, events would have happened within the scene so quickly, no discernible information would be able to be perceived. As such, the speed of time was restricted to 2% of normal time. To then allow additional user interactivity and more useful testing speeds, this fixed value was converted into a slider ranging from 0 (paused) to 2%, with a default speed of 1%.

Start Speed - This was another slider that was added to allow a further additional interactive feature. It was implemented to allow us to test that forces were working correctly, as a larger start speed of the wave, would have more force, so in turn have a bigger impact on the room. Simply, this slider altered the initial speed that the particles in the wave would be moving at when a shock wave is spawned. The slider was limited to a minimum of 500 (meters per second) and a maximum of 2,000. These limits were given, as 340m/s is the minimum speed particles need to travel at to be considered a shock wave and any faster than 2,000m/s was still too fast even at slower times.

Frame per second counter (FPS) - This was implemented in order to measure the efficiency in which our simulation ran. It was deemed as a vital feature, as some of our functional tests were measured against FPS and this was the only way to determine a pass/failure. Although Unity has its own FPS counter built in, our own version needed to be written as Unity's did not perform in the way that was expected, due to us altering the time scale.

Spawn button - It was decided that for user friendly purposes a button be used to input the origin of the shockwave. Once pressed a message appears telling them how to place the shockwave origin.

Reset button - This button is self-explanatory, it refreshes the scene when pressed, whilst preserving the position of the camera and all sliders.

The control features included on the fronted aren't just those visible in Figure 9, other features created/displayed in the final product are:

Help button - Written as an overlay which displays the instructions for controlling the scene.

Camera controls - The scripts written were done so that the user could control the scene using the keyboard, making the program more accessible for the user. As an aim for our project was to create an accurate representation of a shock wave, it was decided that the user should be able to change the view in whilst the scene is running, to see the shock wave from various different angles.

Testing and Performance

To ensure our GUI met the previously documented requirements and was also user friendly, testing was carried out numerous times throughout the development phase. This consisted of ensuring all code relevant to the front end worked as expected and as efficiently as possible. User testing was also constantly partaken through other members of the group using the system and giving feedback for any necessary changes.

In order to get as wide a range of feedback as possible, during the Poster Fair further user testing was performed by asking students and members of staff to evaluate the success our system. Analysis of this feedback was then used to determine our outcomes in our functional tests.

The functional tests were carried out by evaluating if each of the original requirements were met successfully in our final product, or not.

Conclusions

In conclusion we have successfully met two of our three main objectives. We feel we haven't met one of our objectives which was realistically simulating a shock wave. This is because we severely underestimated the complexity of the physics involved in shock wave interactions. We have however, produced an easy to use system which is visually appealing (as found in our questionnaire results) and provides a solid stepping stone in the right direction for further development.

It is clear now why many computer games fudge these interactions, not only from a computational perspective but also from a scientific standpoint. Realistic shock waves are computationally very expensive and these features would not be fully appreciated by an average gamer. However, we feel our systems target audience may be scientists wishing to run simulations where a good visualisation is important.

Overall we feel that we have successfully met our main objective which was to research how shock waves can be simulated realistically in real time as well as answering the question as to why video games often fudge these calculations.

Future Work

Given more time, there are a number of areas in which the system and demonstration software could have been improved.

Serialisation - The particles shock wave system currently does not support serialisation, means the state cannot be saved to disk. Saving and loading is a common requirement of video games.

Finding input parameter bounds - The system can act in undesired ways when given extreme high or low input parameters. Additional testing could find the optimum and bounds of input parameters.

User scene creation - a feature could be added so that the user can customise the scene, changing the layout of the room and adding/removing objects.

Destructible environment - destructible scenery and objects could be added.

Acknowledgements

We would like to thank Dr. Peter Kenny for supervising us during this project and for all the help and advice he had to offer. We would also like to thank the University of Kent staff and students who provided invaluable feedback during the poster fair and throughout the duration of the project. Finally we would like to thank the developers of the Unity Engine for providing us with a platform to work on.

References/Bibliography

[1] Erlet, Glenn, *Shock Waves. The Physics Hypertextbook*. [Online] The Physics Hypertextbook. Available from: http://physics.info/shock/ [Accessed 14 November 2016]

[2] Anderson, John D. (2001). *Fundamentals of Aerodynamics.* 3rd edn. pg 487. Boston: McGraw-Hill

[3] Feynman, Richard P. (2003). *The Feynman Lectures on Physics.* [Online] Vol. I Ch. 51: Waves. Available from: http://www.feynmanlectures.caltech.edu/I_51.html [Accessed 14 November 2016]

[4] Deserno, Markus. (2004). *How to generate equidistributed points on the surface of a sphere.* [Online] Available from: https://www.cmu.edu/biolphys/deserno/pdf/sphere_equi.pdf [Accessed 14 November 2016]

[5] *Technologies, Unity*.[Online].Unity Scripting API Reference. Available from: https://docs.unity3d.com/ScriptReference/index.html [Accessed September 2016]

[6] *'Requirements'* document found in the project corpus index.

[7] *Axure.com* [Online] Available from: https://www.axure.com/ [Accessed 3rd October 2016]

[8] Anderson, John D. (2001). *Fundamentals of Aerodynamics.* 3rd edn. pg 58. Boston: McGraw-Hill