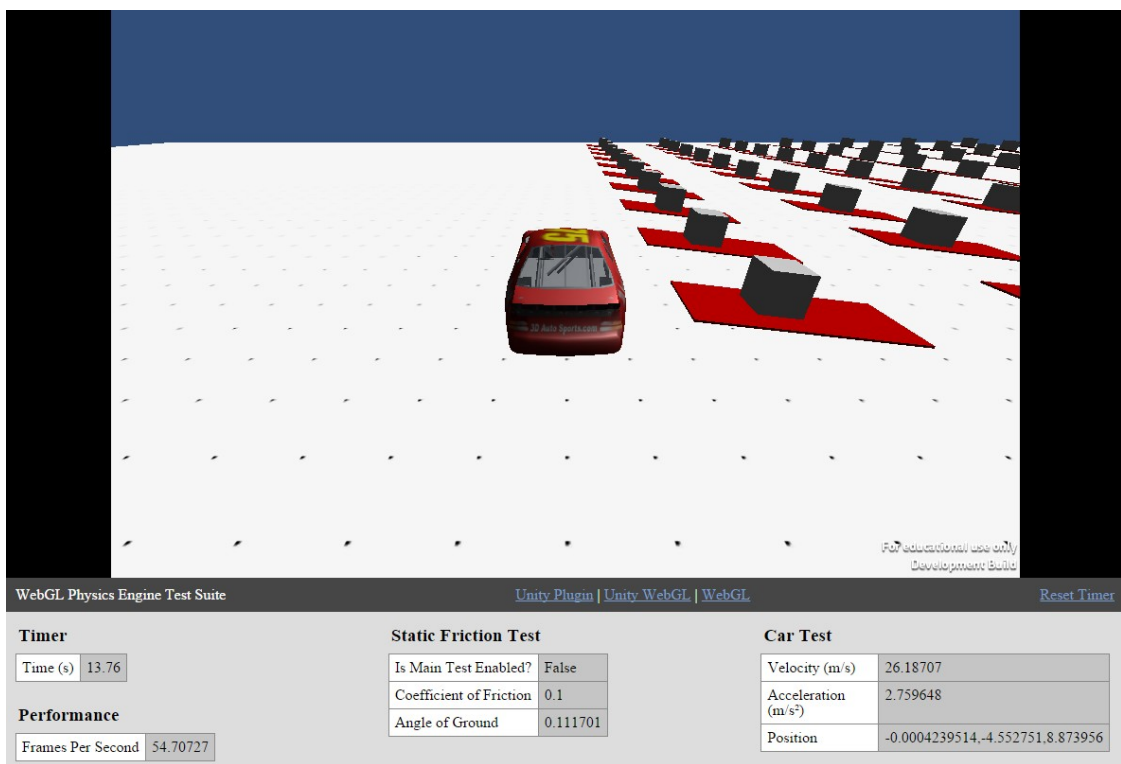


Research on the Accuracy and Performance of Physics Engines in Web Browsers

by

Paul Shields

11th of May, 2015



WebGL Physics Engine Test Suite [Unity Plugin](#) | [Unity WebGL](#) | [WebGL](#) [Reset Timer](#)

Timer
Time (s) 13.76

Performance
Frames Per Second 54.70727

Static Friction Test

Is Main Test Enabled?	False
Coefficient of Friction	0.1
Angle of Ground	0.111701

Car Test

Velocity (m/s)	26.18707
Acceleration (m/s ²)	2.759648
Position	-0.0004239514,-4.552751,8.873956

For educational use only
Development Build

A dissertation submitted in partial fulfilment of the requirements for the degree of MASTER OF ENGINEERING in Computer Science from the School of Electronics, Electrical Engineering and Computer Science, Queen's University of Belfast

Abstract

The comparison of physics engines for video game development is a well-researched area, for desktop and console video games, however, it was unknown for physics engines built for the web browser. The aim of this research is to provide a quantitative analysis of the accuracy and performance of the physics engines used in two of the fastest growing web-based game frameworks, Unity, via the Unity Web Plugin and via the WebGL build, and WebGL, using the most accurate physics engine for the framework currently available. Both the web browser and the hardware in which the game was running on were examined.

The experiments used to compare these physics engines comprised of a Static Friction Test, comparing how the physics engines calculate the angle at which a box began to slide on a plane to real world static friction data, and a Car Test, examining if there was any side-to-side deviation from a car, created using the physics engine's vehicle module, accelerating in a straight line and the stopping distance of the car at various speeds.

The results showed that all tested engines ran at an acceptable level of performance and accuracy for use in a game. The Unity builds has the best performance overall, whereas WebGL with Bullet was more accurate. Hardware had little-to-no effect on the engines.

Acknowledgments

Thanks to John Busch and Queen's University of Belfast for providing me the opportunity to explore this research area, and to John for the guidance provided through the last 8 months.

To my friend Nathan McKee for providing a wealth of writing knowledge and for proof reading this dissertation.

And last but not least, to my parents, who have supported and encouraged me without fail in all my endeavours.

Contents

1 – Introduction.....	6
1.1 – Hypothesis.....	6
1.2 – Structure of the Dissertation.....	7
2 – Background.....	8
2.1 – Real World Physics Background.....	8
2.1.1 – Gorst and Williamson.....	8
2.1.2 – Andersson, P. Hugoson et al.....	9
2.2 – Video Game Physics Background.....	9
2.2.1 – Boeing and Bräunl.....	10
2.2.2 – Seugling and Rolin (2006).....	10
2.2.3 – Yogya and Kosala.....	12
2.3 – Video Game Physics Background - Vehicles.....	13
2.3.1 – Mannerfelt and Schrab.....	14
2.4 – Software & Frameworks Background.....	14
2.4.1 – Unity.....	15
2.4.2 – WebGL.....	16
2.4.3 – Web Browsers.....	17
2.5 – Decisions Based on the Research.....	18
2.5.1 – Game Frameworks.....	18
2.5.2 – Browsers.....	19
2.5.3 – Hardware.....	20
3 – Experiment.....	22
3.1 – Static Friction Test.....	22
3.2 – Car Test.....	23
4 – Implementation.....	26
4.1 – Method.....	26
4.2 – Development Parity.....	26
4.3 – Unity.....	27
4.4 – WebGL.....	27
4.5 – Test Suite.....	28

4.6 – Automation.....	29
5 – Results.....	31
5.1 – Static Friction Test.....	31
5.1.1 – Correctness.....	31
5.1.2 – Browser Influence.....	33
5.1.3 – Performance.....	35
5.1.4 – Hardware Speed Influence - Accuracy.....	36
5.1.5 – Graphical Rendering Influence - Performance.....	37
5.1.6 – Graphical Rendering Influence - Accuracy.....	38
5.2 – Car Test.....	39
5.2.1 – Car Drift Correctness.....	39
5.2.2 – Stopping Time Correctness.....	40
5.2.3 – Browser Influence.....	42
5.2.4 – Performance.....	43
5.2.5 – Hardware Influence – Stopping Time Correctness.....	44
6 – Conclusion.....	45
6.1 – Conclusion from Results.....	45
6.1.1 – Does the web browser have an effect on the performance/accuracy of the physics engine?.....	45
6.1.2 – Does either physics engine, running on Unity or WebGL, perform accurately to real world physics models?.....	45
6.1.3 – Does the speed of the CPU and GPU of the machine have an effect on the performance and accuracy of the physics engines?.....	46
6.1.4 – Do the physics engines run at an acceptable level of performance and accuracy for use in a video game?.....	46
6.2 – Further Research.....	46
References.....	48
7 – Appendices.....	50
7.1 – Raw Data.....	50
7.1.1 – Static Friction Test.....	50
7.1.2 – Minutes from Meeting.....	55

1 – Introduction

In recent years, there has been a shift in trends in software development. Previously, applications were written in a language that was native to the Operating System it was to be executed upon. Nowadays many applications are now written specifically for the web; made to be accessed through a web browser [1].

Video games have been at the forefront of adapting to this shift in focus, with browser based games seeing tremendous growth in the last few years [2]. Their popularity can be attributed to the advent of platforms such as social media website Facebook and browser game platform Kongregate. These platforms allow the video games to be played virtually instantly and allow users to play against other players worldwide. With this connected society we are currently living in, this ease of access and wide appeal is advantageous compared to traditional methods of distribution of video games.

Physics is a huge aspect of many of the most popular video games on any platform [12]. The majority of the most popular video games today, games such as Call of Duty, Grand Theft Auto and Assassin's Creed rely on physics engines to create a rich and realistic world that a player can interact with and believe in. Physics engines aren't just for the large AAA video game titles, many smaller independent titles, such as Angry Birds and Cut the Rope are specifically built around the user's interaction with physics.

With the renewed focus on web based games in recent years, it is important to understand which methods of development will give the best results for this hugely popular category of video games.

It is currently unknown which method of browser based game development will give the best performance for physics based games. The aim of this research is to test physics engines in native WebGL and JavaScript, Unity with the Web Player plugin and Unity built for WebGL to see which system will give the best performance for a standardised physics based game environment.

1.1 – Hypothesis

As this research is able to cover a broad area of tests in regards to a physics engine, a specific set of requirements had to be set out. These requirements are described in described in Table 1.

RQ1	Does the web browser have an effect on the performance/accuracy of the physics engine?
RQ2	Does either physics engine, running on Unity or WebGL, perform accurately to real world physics models?
RQ3	Does the performance of the physics engine have an effect on the accuracy?
RQ4	Does the speed of the CPU and GPU of the machine have an effect on the performance and accuracy of the physics engines?
RQ5	Do the physics engines run at an acceptable level of performance and accuracy for use in a video game?

Table 1: Research Questions

1.2 – Structure of the Dissertation

Chapter 2 provides background research into the various aspects of this paper, including the experimentation of real world physics, experimentation of video game physics, information on the tested physics engines and game frameworks, and information on the web browsers involved.

Chapter 3 provides the details of the experiments conducted. These include a Static Friction Test, a Car Test and modifications to these tests that comprise the suite of experiments conducted in this paper.

Chapter 4 contains information on the implementation of the experiments: how they were implemented, the areas in which development parity was achieved and issues specific to each physics engine and game framework.

Chapter 5 provides the results to the detailed experiments in Chapter 3. The results are split up by the main test, its modifications, effects of the web browsers and effects of the hardware.

Chapter 6 gives a conclusion to this paper, attempting to answer the research questions from section 1.1 and providing further research that could be conducted, extending the results collected in this paper.

2 – Background

In this chapter, previous research will be provided, covering the current relevant findings in real world and video game physics, web browsers, WebGL and Unity. Currently used methodologies for testing real world and video game physics is also covered and the decisions for this research based on previous works are provided.

2.1 – Real World Physics Background

For this research to be robust and valid, we must have real world data to compare our generated game engine data to. Initially, the aim was to find a simple physics experiment that could be replicated precisely to test the basic correctness of each physics engine. This would provide a base to build the more complex tests.

2.1.1 – Gorst and Williamson

Friction in Temporary Works [7] provides a theory based around static friction. It states that when two items are placed one on top of the other and are not in motion there is a certain value of lateral force which can be resisted across the interface. This lateral force is dependent on the materials of the two items that are in contact with each other. This leads to the theory of the Coefficient of Friction: a value that is represented by the ratio of the coarseness of the two items in contact, which is used to calculate the resistant lateral force.

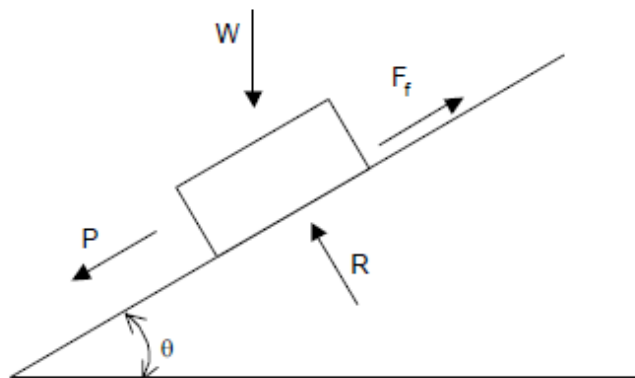


Fig 1: Diagram showing the described experiment

In their experimentation, they constructed a test rig by placing two objects on a “tilting table”, consisting of a flat surface balanced on one end by a jack. This jack was used to manually raise one end of the tilting table manually using a winding handle. The table was raised to the point at which

the upper object slipped, at which point the angle of slip was recorded. With the recorded results, they were able to compare the findings with the values of the coefficient of friction defined by the European Standard and by the UK Code Of Practice.

Gorst and Williamson [7] found that many results, such as the value of the coefficient of static friction does not appear to be affected by the position of one object on another and an increase in weight to the upper object has a small increase in frictional value, leading to a maximum and minimum value for the coefficient to be allied to each material combination.

Using the data they collected from this research, they were able to calculate a proposed friction table, showing values of the coefficient of friction for pairs of materials, in both dry and wet situations. This shows the real world uses for this test on friction; they were able to use it to inform the choices made on the surfaces of cargo platforms and packaging of cargo.

Their findings prove the correctness of the formula for the coefficient of friction. By adopting an experiment similar to that used in this paper, the results they found can be used to provide real world data for our research into the video game simulation of physics.

2.1.2 – Andersson, P. Hugoson et al.

This method of testing the static friction of an object has become the standard. In 2007 [8], a large scale version of this experiment was conducted by using a cargo platform. Two objects were placed on top of each other on a cargo platform, with rubber mats placed under the lower object to avoid slippage. This cargo platform was then tilted until the upper object began to slide. They used an Inclinator to get accurate inclination angles and an accelerometer to test if the vibrations of the engine had any effect on the slippage of the cargo.

From this paper, Gorst and Williamson, as well as the research performed by Boeing and Bräunl [6] in the video game physics section, this method of testing and experimenting is proven to be an accurate method of testing static friction. With these papers being experiments executed in real world physics, we can use them as a source of real world data to compare our generated video game based data to, allowing us to calculate accurate results in our conclusion.

2.2 – Video Game Physics Background

There is much existing work in the analysis [5, 6, 10, 11], comparison and testing of various 2D

and 3D physics engines. While most of this work was not in the Browser based physics engines space, many of the testing procedures and lessons learned can be applied to this research.

2.2.1 – Boeing and Bräunl

Boeing and Bräunl [6] aimed to provide a qualitative evaluation of a number of free, publicly available physics engines for simulation systems and game development. They tested seven different traditional physics engines using the Physics Abstraction Layer (PAL) [16]. The Physics Abstraction Layer provides a unified interface to a number of different physics engines. This means that a developer can write the physics scenario once, in C++, and execute the same code using different physics engines to generate their results. In their testing, they determined 6 essential factors that determined the overall performance of the tested physics engines: Constraint accuracy; Numerical accuracy of the simulation, through the integrator; The accuracy of the collisions in the simulation through object representation; Collision detection and contact determination; Material properties, determining the frictional properties; and Constraint implementation.

Many of the experiments carried out on the physics engines mimicked real world physics tests, something which we have taken into consideration when deciding on the tests used in this paper's research. For example, in the material properties simulation, a 5x1x5m box was placed on an inclined plane. A static friction coefficient was placed on the box and plane and the angle of the plane was increased until the box began to slide. This test is very similar to a test used to calculate the coefficient of friction of different materials on a constant metal plane in real world physics.

The researchers do not list any issues with the research carried out, or any future research that could be carried out on the work they have done. This would be useful for anyone looking to use this paper as influence for further research on other aspect of physics engines. The paper also doesn't list the hardware that the experiments were executed on. While all the experiments are built around the accuracy and correctness of the physics engines, the speed of the hardware can have an influence on the physics engines, so this information would have been useful to know.

2.2.2 – Seugling and Rolin (2006)

A. Seugling and M. Rolin (2006) [10] conducted research on physics engines with the aim of comparing freely available physics engines and commercial physics engines, with the aim of finding a suitable physics engine to use in a plugin module for a 3D-authoring tool.

To find the physics engines they were going to test, they ran an evaluation of nine different physics engines to the requirements they specified for the plugin. From this evaluation, they chose Novodex, ODE and Newton Game Dynamics for their research. With the engines chosen, they were then subjected to several performance tests, which were chosen to evaluate as many different aspects of a physics engines as possible.

For the Dry Friction Test, they used a box sliding on a plane scenario, similar to that used by Gorst and Williamson in their real world experimentation. They measured both static friction, where the objects were initially not moving and the force required to start one of the objects moving is found, and dynamic friction, where the objects are initially moving between each other and the frictional force required to cause them to slow down and stop is found. In this scenario, all three engines provided good results. While Novodex had a higher threshold for movement to be initiated, all the engines provided realistic frictional data thereafter.

In a bounce test, the scenario was constructed to find whether objects in each scenario gained, lost or preserved the correct amount of energy on collisions. A sphere was placed above a plane in world with no gravity and a constant velocity was placed upon it. In a second test, the sphere was placed in collision with the plane to examine how the physics engines were able to handle such a scenario. The engines varied in their approaches to solving the two scenarios. Newton's solution caused the sphere to lose energy when it should have stayed constant, Novodex provided the correct results consistently and ODE gave good, but not perfect results.

The researchers concluded that, while all physics engines tested were better in certain situations than others, the commercially available Novodex was the best overall physics engine available. However, they added that in choosing a physics engine for a user's own software or projects, they should research what aspects of a physics engine they require and find the best solution for their needs.

This paper was thorough in all aspects of their investigation, from the initial decisions around the physics engines to the variety of scenarios they tested the engines in. Their results are good, giving well explained and logical reasoning for their decisions. The main aspect of this paper that can be included in this research is the level of detail that the researchers went into for their experiments; the description thereof, the reasoning, the clear and concise results formed from the experiments and the ramifications of the results.

2.2.3 – Yogya and Kosala

When performing qualitative evaluation of a number of free WebGL physics engines, R. Yogya and R. Kosala [5] defined four different tests that would adequately cover the available feature sets of the physics engines tested:

Collision System

The process used to test physics engines' collision systems involved first constructing a scenario where an inverted square pyramid mesh is constructed, as shown to the right. The pyramid apex was 1m deep, and the opening of the pyramid measures $2 \times 2\text{m}^2$. An 8×8 grid of spheres with a radius of 0.04m was constructed above the pyramid and the spheres were pulled down into the mesh by gravity. This was tested at multiple refresh rates, 100Hz, an unrealistically high refresh rate, 15Hz, an average video game rate and 5Hz as an extreme case.

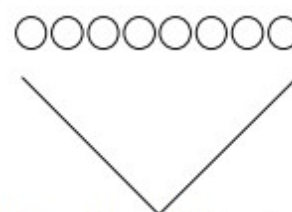


Figure 11 – Collision test configuration

First, it was measured if the physics engines were able to detect the falling spheres without them falling through the thin pyramid. Passing this test, then penetration of the pyramid is detected by comparing all of the spheres positions to the polygons that make up the pyramid. If any sphere is less than its radius away from the pyramid's polygons, then a penetration error is accumulated.

Constraints

Constraints in physics engines are a method of limiting an objects movement in such a way that it appears to be constrained by another object, such as being tied to something by a piece of string.

The method used for testing the stability of these constraints involved generating a chain of spherical links connecting spheres and attaching it to two boxes on either side of the chain as shown in the diagram to the right. Each sphere in the chain had a radius of 0.2m, and a mass of 0.1kg. The boxes on either side had a mass of 40kg and measured at $1 \times 1\text{m}^2$. Over twenty seconds, they measured the constraint error by recording the distance between two links minus relative to the initial case.

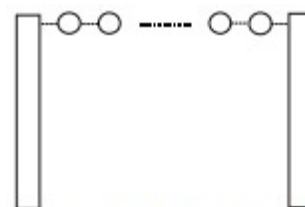


Figure 8 –Constraint test configuration

Accuracy of the Simulation

For correctness testing, the method used was to write a test scenario that includes a static object positioned on the ground and an object positioned in the air above it. The object is allowed to fall via gravity. The positions of the objects are updated and recorded until the objects collide. When the objects collide, the physics engine should handle the collision appropriately. The velocities of these objects are measured at each point of update and these velocities are analysed after program execution for its correctness by using the relevant physics formula stated below:

$$\theta = \arctan(\mu)$$

Figure 2: Algorithm used to calculate the angle of slide in the Accuracy of the Simulation experiment

Another process suggested involved the integrator, which is responsible for calculating a body's position given the forces acting on it. To test this, a sphere is generated at the origin and a gravity value of -9.81m/s is applied to it. At each timestep of 0.01, positions of the sphere were recorded. These recorded values were then compared with the correct real world values, using the formula:

$$s = \frac{1}{2}at^2$$

Figure 3: Algorithm used to calculate the displacement of an object over time

Where r is the bodies displacement, a is the bodies acceleration and t is time.

Stacking Test

As a test of stability and how realistic a physics engine can model the real world, a stacking test can be used. An implementation of this test involved a set of $1 \times 1 \times 1 \text{m}^3$ cubes at 1kg each stacked on top of each other, with a distance of 0.1m between them. Each cube is then moved horizontally, parallel to the ground, by a maximum of 0.1m in any direction. As there is no feasible method to verify if the stack of boxes is obeying the laws of physics, i.e. at which point it should collapse, it was examined through visual inspection.

2.3 – Video Game Physics Background - Vehicles

From reading the papers in the previous section, a trend had emerged: the papers all examined various basic aspects of any general physics engine; however none of them examined aspects of the

physics engines specific to video games. As this paper is based around the use of physics engines in video games, testing a video game specific aspect of the physics engines was a natural extension of previous research.

2.3.1 – Mannerfelt and Schrab

Mannerfelt and Schrab [11] detailed a test suite they developed to test physics simulations at Meqon. In their test suite, they had to come up with a number of different test cases that were able to cover every aspect of the physics engines. These tests cases included a stacking test case, a spring stability test case, a raycasting test case and a vehicle test case.

For the vehicle test case, the researchers the vehicle module within the tested physics engine to construct a car. In physics engines, the vehicle module is a module that enables the creation of vehicles with a typically higher performance than a user would get if a vehicle was created with regular rigid body elements. To test the vehicle module, they accelerate the constructed vehicle along a straight line for a certain length of time. They record the position of the vehicle before and after the movement, which can then be used to calculate if there was any deviation from the straight line by the vehicle. If the vehicle module is accurate, then there should be no deviation from the line.

They state that the vehicle module is a complex module, with many parameters that could be tweaked for many different types of vehicles. To remove any unnecessary variation from the results, the same vehicle is constructed, no matter what physics engine is being tested.

The vehicle test is a good, simple test that has the potential to provide accurate results for the vehicle module in the physics engine. As the experiment is built for automated tests, the experiment described in the previous research has to give results that can be processed automatically by an algorithm. By using this approach as a base test in the research for this paper, the experiment can be expanded to collect more result from the same test case that can be processed by a researcher themselves.

2.4 – Software & Frameworks Background

2.4.1 – Unity

Unity3D is a cross-platform game creation system [22]. Compared to WebGL which is a royalty

free web standard, Unity is built as a proprietary, self-contained Integrated Development Environment (IDE) and game engine system that allows a user to develop a game in a single language (either C#, UnityScript or Boo [23]) and build it for multiple platforms, including Windows, OS X, Linux and the web browser. Unity is available in two editions, a Personal edition that is free to use (within Unity's terms of use) and a Professional version, which adds support for features such as a customisable splash screen and use of Unity's cache server for asset sharing [29].

As of Unity 5, both the Web Player and WebGL build version of a Unity game uses PhysX 3.3 as its underlying 3D physics engine [32]. PhysX in Unity is a native C++ middleware physics engine in the Web Player, which is then compiled to JavaScript using emscripten, a LLVM-to-JavaScript (Low Level Virtual Machine) compiler [33]. As part of this research, we will examine whether this added compilation will have an effect on the accuracy or performance of the physics engine.

Currently, embedding a Unity game inside a web browser requires the end user to download and install a Web Player to play the game, which is a major disadvantage compared to WebGL based games. With the recently released Unity 5.0, however, Unity users are able to export their games directly to WebGL. This will potentially give the best of both worlds: Plugin-free browser based games with the power of the Unity engine running the games.

In 2010, M. Labschutz, K. Kroszl [13] used Unity 3.1 to develop a game with 26 computer science students. It was chosen due both the ability to prototype easily and the amount of different platforms the engine can build a game for. This, they state, is why the engine attracts many small and middle-sized development studios who are unable to invest in expensive high end rendering engines. The engine is particularly suitable for prototyping and quick development due to its WYSIWIG (“What You See Is What You Get”) editor, which allows for drag-and-drop movement for objects in the scene and live variable editing. The engine also comes with a number of prebuilt scripts that can be applied to any object in the scene, these scripts include character controllers, follow up cameras and other important features.

Labschutz and Kroszl do state the lack of integrated modelling abilities as a drawback to developing in Unity, requiring external modelling tools such as Maya [28]. Unity was found to have excellent support for the .FBX format for 3D models, exported from Maya. This allowed Unity to fit seamlessly into their concurrent team workflow. While the modelling for the level was created in Maya, they used Unity for effects such as fire smoke and lens flare, and sound. This was all easy to

implement using the provided tools in Unity, as from the 1550 recorded person-hours, only 310 person-hours were spent in Unity's tools.

The focus of this paper was on the 3D modelling in Maya. While the paper cannot be faulted for this, there are multiple aspects of the final product they created that are dependent on the technology in Unity, such as the sound and effects. These are summed up quickly in short chapters and do not detail the ease of use or issues they had with implementing these features. The information they give on the Unity feature set and relative time taken to develop on the platform, however, is useful. It can be used to compare to WebGL, comparing the feature set and experience in developing for the platform.

2.4.2 – WebGL

WebGL is a low-level 3D graphics API [5], developed by the Khronos Group, the same organisation that develops OpenGL. The API was built to allow web developers direct access to the low level hardware within a user's computer. The technology is similar to the desktop-based OpenGL and is based upon OpenGL ES 2.0. The main advantage for this new technology is that it is built as a cross-platform, royalty-free web standard [5], meaning that it is built directly into modern web browsers and requires no additional plugin download or installation to play a game built in WebGL. WebGL, however, is just a 3D graphics API. It does not include any game engine modules and requires the user to supply it themselves if they are developing a game. The WebGL API is accessed through JavaScript through the HTML5 Canvas element as Document Object Model interfaces.

For this research, a physics engine will have to be chosen to represent WebGL's ability to play physics based video games. The option of developing a physics engine specifically for this research was a viable possibility, but in this research we used a publicly available off the shelf engine. This decision was made as in previous research; it was shown that the majority of video games are developed on off the shelf game engines [4], due to the stability of the already developed engine and amount of support that a developer can get for the software, as opposed to the level of difficulty in developing their own. By using a popular, available physics engine, this research will be designed to be representative of a normal video game and its development.

The engine used in this research was chosen using the findings of previous research. In 2014, a comparison of physics engines in the WebGL environment was conducted [5]. This paper tested 3

physics frameworks: cannon.js, bullet.js and jiglib.js. The frameworks were put through a number of tests in order to compare their Performance, Correctness, Completeness and Compatibility.

For the engines tested using the performance testing process (Cannon.js and Bullet), Cannon.js performance was by average 13.88% faster compared to Bullet. Cannon.js was definitively faster in calculating sphere rigid bodies compared to Bullet, however for box rigid bodies the performance in both were similar. They did not include Jiglib.js in this testing due to incompatibility with their game engine. In the correctness testing, Bullet was found to be accurate for both box and sphere collisions, whereas cannon.js had issues with sphere collisions.

These tests were executed in isolation to each other, so there are no results on the effect, for example, of a performance drop in the game affecting the accuracy of the simulation. Physics engines are large, complex pieces of software that have many interacting systems to simulate a real world physics scenario. By merging some of the tests in various forms, the effect of one situation on another can be examined to see if the results of the latter situation stay true. These issues should be addressed in this research. From this paper, the decision was made to use the highest rated engine from this paper, Bullet, in this further research.

2.4.3 – Web Browsers

As there are many different web browsers available on the Internet, free of charge, it is important that this is taken into account when analysis on any web based research is taking place [3].

For this research, as WebGL is one of the web-based game platforms being tested, it is important to understand the performance and capabilities of the JavaScript interpreters and engines within each of the major web browsers. The current work on web browsers [3] use Apple's SunSpider JavaScript Benchmark [14], which provides a thorough testing of many features within JavaScript and is well recognized [9] as a reliable measure of a browser's JavaScript performance. As part of the SunSpider tests, the JavaScript's ability for 3D manipulation is tested, which means the results of the testing should show a correlation with the speed of execution of a WebGL game.

In the SunSpider test suite, nine different tests are executed to examine the performance of the JavaScript renderer. The tests are designed to be relevant to real world JavaScript uses by covering a variety of advanced workloads and programming techniques, and reports a single score, balancing all the recorded results. The tests include: 3D manipulation, access, bit operations,

control flow, cryptography, date/time, mathematics, regular expressions, and string operations. The 3D manipulation test is a simple 3D raytracer program [31] that tests arrays and floating-point maths in relatively short-running code.

Nelson et al. [3] shows that Firefox had the quickest 3D rendering, at 0.3 seconds. Firefox was followed by Opera, IE then Safari. Overall, Safari scored the best, with the SunSpider test averaging around 8 seconds per run, followed by Opera. This would suggest that, while Firefox may be the fastest with plain WebGL rendering, when we integrate a physics engine into the scene, it may not perform as well due to general JavaScript execution being slower. It was worth noting that the newer, unstable versions of the web browsers showed much better results than their stable counterparts, showing much advancement in JavaScript technology at the time.

The major issue with this paper, however, is its age. The paper tests much older versions of web browsers than we will be using (e.g. Firefox 3.0 as compared to Firefox 37). Its methodologies are sound though and have been used by many sources for web browser comparison since. A recent study [15], once again using SunSpider, places IE11 ahead of the other browsers in terms of JavaScript execution speed, at 131ms, with Chrome 34, Firefox 29 and Opera 20 coming in around 230ms each.

While this information is important, as the WebGL implementations of the scenarios depend on JavaScript as their execution language, it will be interesting to see if the results measured here by the benchmark test will be supported by the results of the browser influence test in this paper.

2.5 – Decisions Based on the Research

2.5.1 – Game Frameworks

In this research, careful consideration was taken to decide what web based game frameworks would be used to represent the current trends in web based games development. From the research in Previous Research, the following were decided as the frameworks to test:

Unity 5.0 – Web Plugin: Unity is currently one of the most popular game frameworks and engines today, with its ability to develop a game once and publish it to many platforms with a few clicks. The developers of Unity have always been supporters of web based games, with the engine being able to publish a game for their own proprietary web plugin. Unity uses a modified version of PhysX 3.3 for its physics engine.

Unity 5.0 – WebGL: From version 5.0, Unity has included the ability to publish a game specifically for the WebGL platform. As a recent addition to the Unity game engine, we wanted to test if the WebGL version was as stable and as accurate as it's more established Web Plugin counterpart.

WebGL: In addition to the Unity game engine being published to WebGL, it was important to test a game that was written specifically for WebGL through the base JavaScript bindings. For our game, we used the most popular WebGL game framework, Three.JS [21]. From our research, it was found that Bullet was the most accurate and widely used physics engine in WebGL, so it was chosen as our representative WebGL physics engine. However, instead of using the direct emscripten port from the original C++ Bullet implementation, we used PhysiJS, which provides us a direct interface to Three.JS without sacrificing any performance or accuracy.

2.5.2 – Browsers

Today there are many different browsers on the market that users can use to access the internet. Each of these browsers uses their own unique technology to implement the same web standards that are used across every website and web based video game. To this extent, we needed to ensure that, in this research, we covered both the most popular browsers and the most diverse set of implementations of these web technologies as possible. With this in mind, we decided upon the following browsers:

Google Chrome (Version: 42.0.2311.90): Google Chrome now has the largest market share of web browsers with 31.8% in the United States [17], so it was imperative that this was the first browser that we test. The browser is built upon Blink for its rendering engine, a form of the popular WebKit layout engine. It uses a custom developed, open source JavaScript engine named V8 which fully supports WebGL 1.

At the time of this decision, Chrome supported the Unity Web Plugin, as well as WebGL. As of April 2015, however, Chrome now disables the NPAPI API by default, the API used by many web browser plugins, including the Unity Web Player [27]. For these experiments, the NPAPI API will be re-enabled via a command line argument, but it is important to note that by September 2015 Chrome plans to completely remove support for this API, and thus the Unity Web Player.

Mozilla Firefox (Version: 37.0.2): Firefox currently only has 11.56% of the web browser market share worldwide; however it has historically been a stable platform for WebGL. Firefox uses Gecko for its layout engine, which is also custom built and open source, and SpiderMonkey for its JavaScript engine.

Google Chrome Canary (Version: 44.0.2383.0): For this research, the decision was made to test a newer, cutting edge browser to see if some of the newer, bleeding edge technology in the space would perform better than the stable implementations in the current stable releases of browsers. This decision was made based on the research from Nelson et al. [3], which showed that the bleeding edge versions of the tested web browsers showed significantly faster JavaScript results than their stable counterparts. For this, we tested Google Chrome Canary, the bleeding edge, daily released branch of Google Chrome.

Second only to Google Chrome, Microsoft’s Internet Explorer (current version: 11.0.9600.17728) currently has 30.9% of the web browser market share. Unfortunately, while Internet Explorer supports WebGL since version 11, we are not able to use it in our research as Unity’s WebGL export currently does not support it [26]. 25% of the market share is also held by Safari (current version: 8.0.5), the majority of which coming from its presence on mobile. It was not included as this research was focused on desktop operating systems; however with the current level of support of WebGL on mobile platforms, executing this research on mobile platforms is marked as an area of future development.

2.5.3 – Hardware

In all of the research papers on the comparisons of physics engines, the experiments the researchers ran were all executed on the same hardware. To examine if the hardware has an effect on the accuracy of a physics engine, in addition to running the test cases on various browsers, the games were tested on multiple PCs of varying specification, age and quality. The main aim was similar to that of the modifications of the two main tests: to see if the speed and specification of the PC the game was running on had an effect on the accuracy of the physics engines.

	High End PC	Mid Range Laptop	Low End Laptop
CPU	Intel Core i5-4690K	Intel Core i5 460M	Intel Core i3 370M
GPU	NVIDIA GeForce GTX	NVIDIA GeForce GT 420M	Integrated Graphics

	770		
Operating System	Windows 8.1	Windows 8.1	Windows 7

Table 2: Hardware used in these experiments

The High End PC was the main machine for these experiments. As the previous research used similarly high end specifications for their test hardware, this decision was made to eliminate the hardware variable from the initial base set of results, where the hardware was not being measured. The results from the High End PC were then compared with the Mid-Range and Low End machines to examine if they had an effect on the baseline results.

3 – Experiment

Here the experiments performed in this research are described in detail, with details of the real world data that the experiment is based upon provided.

3.1 – Static Friction Test

For the first experiment, we have used the test shown in previous research [?] that is the de-facto standard for testing static friction for real-world objects. Fig.4 shows the simple setup required for this test. A 2x2x2m³ cube, weighing 1kg, was placed on a plane. The plane was then rotated at 0.1 radians every frame until the box started to slide down the inclined plane. Once we have this angle at which the box starts to slide, we can compare it with the angle expected for the test using real world data. The current research in real world static friction shows that the friction angle expected can be calculated from the follow algorithm in Fig. 5:

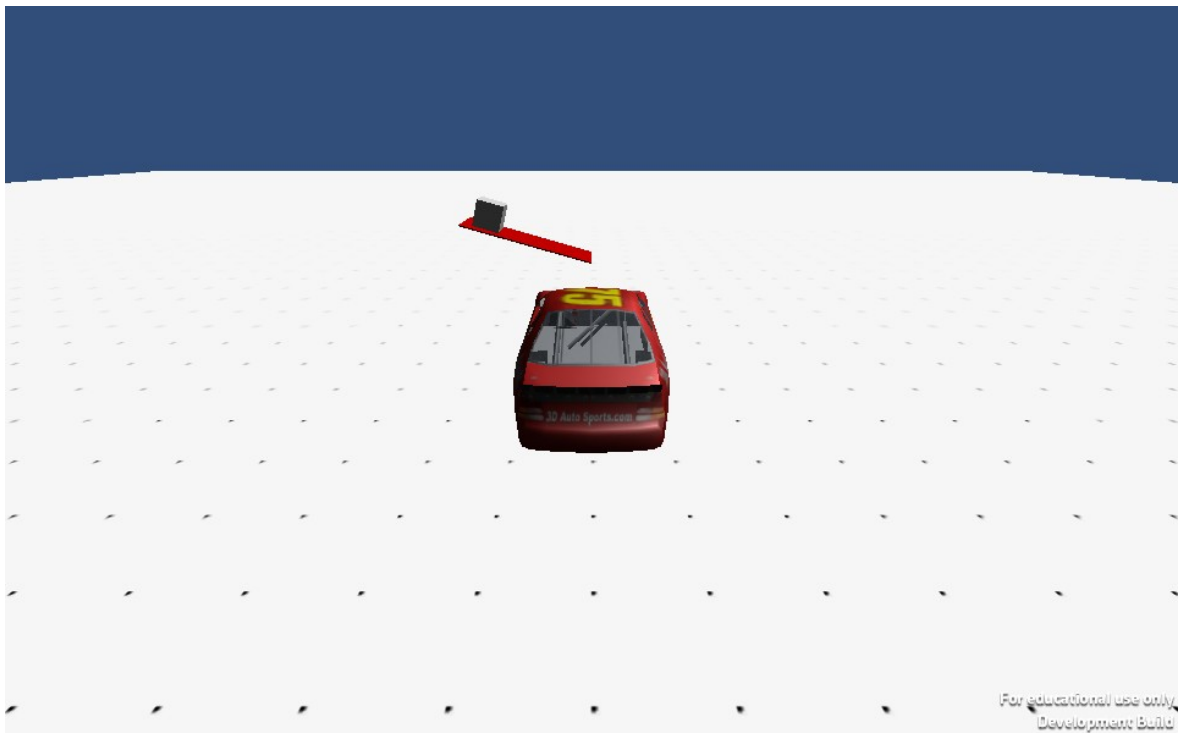


Figure 4: Screenshot showing the Static Friction Test

$$\Theta = \arctan(\mu)$$

Figure 5: Equation for the Static Friction Test, where Θ = Angle of the plane; μ = Coefficient of

friction

The Coefficient of Friction is defined as the value that shows the relationship between the force of friction between two objects and the normal force between the objects, or measure of how difficult it is to slide a material of one kind over another [16]. Within the game engines, we set the coefficient of friction on each object in the physics scene and once they interact, the engine will average the values to try to give a realistic physics representation of friction.

To expand upon this test, we modified it in two ways. Firstly, we duplicated the test multiple times, and executed them all at once. This let us find both where the limits of performance of the physics engine were, and allowed us to test the accuracy of the physics engine under this extreme load. The test was performed with the number of tests to the power of 10, up to 10,000 tests being executed at once. With these tests, we recorded the angle which the box starts to slide on Test 0, the first test generated and the FPS of the game during the test execution.

The second modification took the duplication test that was the first modification and executed the tests again, except first moving the camera so none of the tests that were being executed were in the view of the camera. By doing this, we were able to check if the performance of the game was being affected by the number of physics objects in the scene or by the need to render these objects on the screen. To ensure the physics tests were still being executed correctly, the angle which the box starts to slide on Test 0 was recorded as before, as was the FPS of the game.

With the results of this test, the data for the angle at which the object slides can then be graphed against the real world angles for each Coefficient of Friction, as defined by Gorst and Williamson [7] to give a visual representation of the correctness of each physics engine. The differences can also be displayed, for added validation. The results of the initial test are then compared with the equivalent results at each Coefficient of Friction level with the results generated from each of its modifications to find out if any of the modifications had an effect on the correctness of each engine.

3.2 – Car Test

For the second test, we wanted to use a scenario that was analogous to a scenario that would be directly applicable to a video game, but still allowed us to collect accurate results. From research,

car and vehicle based games can be some of the most intense physics based games, pushing many physics engines to their limits [12].

For the test, we place the car at the origin of the scene, with no rotation. We use the automated test suite to mimic user input to accelerate the car up to 25 Metres per Second along the Z axis, at which point the automated test suite applies the breaks to the car until it comes to an absolute stop. In the first stage of the process, we record the initial position of the car (in case it has changed from the origin before the test begins) and the position of the car at the point it hits a certain defined speed, for example twenty metres per Second. As the car is only accelerating forward along the Z axis, the position of the car in the X axis should not have changed. This first stage of the test lets us test this theory, as we record the change in position of the car in the X axis.

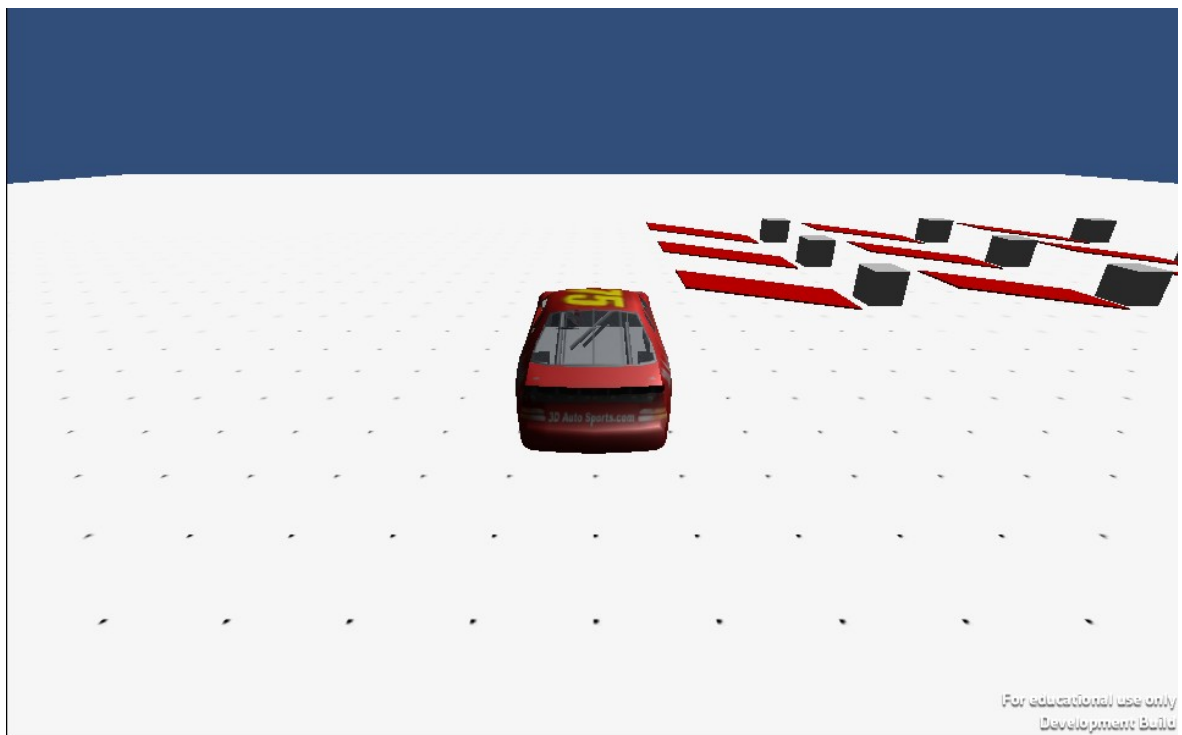


Figure 6: Screenshot showing the Car Test

In the second stage of the process, we record the exact speed at which the car is moving forward when we hit the brakes and the time it takes to decelerate from this speed until the car stops. During the braking time, we also record the deceleration of the car. When the car has stopped moving, we then calculate the average deceleration of the car. With this data, we can record the length of time it should have taken for the car to stop moving using the formula:

$$T = \frac{V - U}{A}$$

Figure 7: Algorithm used to calculate the time require to stop the car

Where $V = 0$ m/s; the final speed of the car

U = Initial Speed of the Car

A = Average deceleration of the car

This test is executed multiple times at multiple speeds, from 5 to 25 metres per second, at 5 metres per second intervals, on the same browser to ensure the results collected are repeatable for all speed levels.

For the modifications of this test, we used a similar duplication method of putting stress on the physics engine. We executed the main test the same way as before, recording the same data except we also have the multiples of the same Simple Friction Test running alongside the car test. We perform the Car test with the same numbers of Simple Friction Tests as previously used, with the number of tests being to the power of 10, up to 10,000 and then all being executed at once.

4 – Implementation

Here details of the implementation of the experiments is described, including the software used and specific issues that occurred in each framework and engine that had to be accounted for to keep the results separate implementations correct.

4.1 – Method

The following software was used to develop the implementations of the experiments:

Unity

- Unity Editor 5.0.1f1
- Microsoft Visual Studio Community 2013 12.0.31101.00

WebGL & Test Suite

- Brackets 1.3

Automation

- Eclipse IDE for Java Developers 4.4.1

4.2 – Development Parity

As the Unity and WebGL implementations had to be developed separately in separate languages, care had to be taken to reduce the number of variables in each scene's implementation to ensure there was no outside effect to the results apart from the variables we were explicitly testing.

In video games, the near and far clipping planes are the minimum and maximum distances at which an object in the scene can be seen, given the object would be within the view of the scene's camera. In these tests it is important as the effect of rendering a large number of objects is tested and if the clipping planes were not in sync, then it could affect the final results. The near and far clip planes in both Unity and WebGL were set to 0.3 and 1000 respectively.

Even with the differences in language between Unity and WebGL, it is still possible to implement some of the same algorithms in both frameworks. The framerate script is the same in both the Unity and WebGL games in order to keep the framerate measurements at the same level of accuracy between both games. All the experiments were first implemented in Unity. Once the experiment and any required scripts were tested thoroughly on this platform, it was then ported directly to

WebGL and PhysiJS, attempting to keep the implementation as close between the two platforms as possible.

4.3 – Unity

The real world Coefficient of Frictions of various objects is based on the material that the object is made from, as shown by Gorst and Williamson [7]. Unity's implementation of this attempts to emulate the real world, by allowing the user to define a "Physics Material", with a value of coefficient for static and dynamic friction for a single object, among other configuration settings [27]. In real world physics, however, the coefficient of friction is a value given for any two specific objects, as it is the ratio of the coarseness of the two items in contact. To emulate this fact, the physics material provides a "Friction Combine" setting, which allows the user to define how the two values of coefficient are combined to give a coefficient of friction for the simulation to use. The options they give are: Average, where the two coefficient values are averaged; Minimum, where the smaller of the two values is used; Maximum, where the larger of the two values is used; and multiply, where the friction values are multiplied with each other.

In these scenarios, both objects in question at any one time are given the same independent coefficient of friction and the default Average setting is used to ensure that the simulation is using the expected value that we define.

For the vehicle implementation, Unity provides a slightly different approach to the vehicle module than WebGL. Instead of defining a single Vehicle module, Unity provides the users with a WheelCollider object. Using this, the vehicle is constructed by simply creating four WheelColliders and placing them at the same position as the wheels on the car model. The variables for these objects are set independently to each other, so care had to be taken to ensure the variables were the same across each individual wheel.

4.4 – WebGL

With the Coefficient of Friction in WebGL, the physics engine works in a similar way to Unity. Bullet assigns a separate coefficient of friction to each object in the physics scene separately and combines them whenever a collision occurs. Bullet, however, doesn't allow the user to modify how the coefficients of friction are combined upon collision; instead it opts to multiply the two values together for every collision [30].

WebGL provides an implementation for a vehicle in a similar way to many physics engines, by providing a single vehicle module. The module is instantiated by passing the vehicle model to the module, as well as variables such as engine power and position of the wheels on the model. Care was taken to ensure these variables were the same as used in the Unity implementation.

4.5 – Test Suite

To extrapolate the data collection from the running game simulation, a browser based Test Suite was developed to show all the relevant data from the running game. As opposed to other methods of recording the results, this method was chosen to give the person viewing the games, either while the experiments are being executed or while manually interacting with the games, the ability to view the live data for each of the elements in the game. This live data could be surfaced in the games themselves, but this implementation allows the data collection to be abstracted and standardised between all implementations of the test in the various frameworks. The Test Suite was developed in HTML, JavaScript and JQuery.

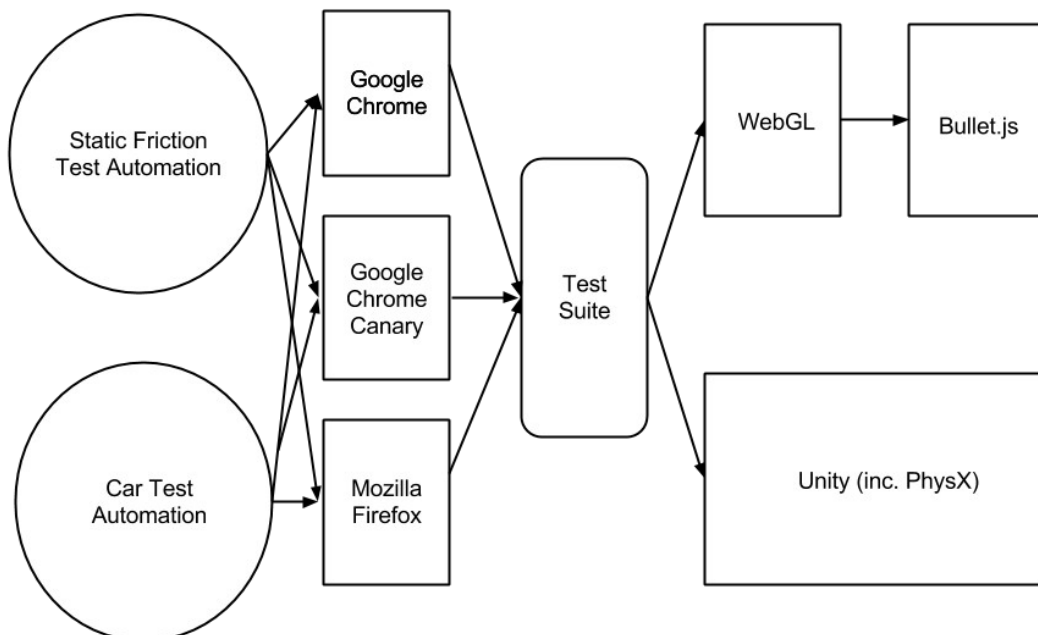


Figure 8: Diagram showing the basic flow of the automation, test suite and test cases

To pull the data from the game, the Test Suite must use two different methods for Unity and WebGL based games. For Unity, it uses the SendMessage function. This allows Unity to actively send output from the game to a manually defined JavaScript function. This function can be sent from the Test Suite to the Unity game on first load. Every frame update, the Unity game, in either plugin or WebGL form, sends the updated data to the Test Suite to display. To emulate this same process in WebGL, a similar system was developed, whereas the Test suite can define the HTML elements for the WebGL game to place the relevant data and every frame, the WebGL game fills out the page with the latest data available.

4.6 – Automation

To gather the data in these tests, a way to emulate user input into the games in a reliable and accurate manner was required. The decision was made to abstract this input emulation to outside of the game implementations themselves to ensure that there is no bias of implementation between the sets of results.

To implement this automated input, we used the Selenium Browser Automation framework [19]. This framework is primarily used for the implementation of automated test suites for websites, but we were able to use it for the browser inputs on the static webpage, the collection of data and other intricacies involved with automating website access. The Java bindings were used for development to create an executable JAR file that is able to execute all the tests accurately and generate the results without any user input. By using Java, we were able to make the automation application portable, so that the tests could be executed on non-Windows machines as required.

During development, a major issue occurred with the use of the Selenium framework. The framework is designed to access and interact with elements in a website's DOM (Document Object Model). With the Unity Plugin implementation of the game, the actual game is executed outside of the website itself, so Selenium was unable to interact with it. To solve this issue, the Java Robot class [20] was used for any inputs related to the games themselves. This class allows the developer to take direct control of the user's mouse and keyboard inputs, meaning that we could provide focus to the games in the browser and accurately emulate the user's inputs for our tests.

To simplify the testing process, the automation application will also start its own embedded Jetty server to serve the test suite and games to test. This allowed the automation framework to be portable, as it allowed the tests to be executed on any machine, independent of an internet connection. The Jetty server is also low overhead, only using a minimal amount of the computers resources, so it did not interfere with the executing tests. Upon completion of the two main tests, a CSV (Comma Separated Values) file for each main test is generated in the Output directory relative to the executable JAR file.

5 – Results

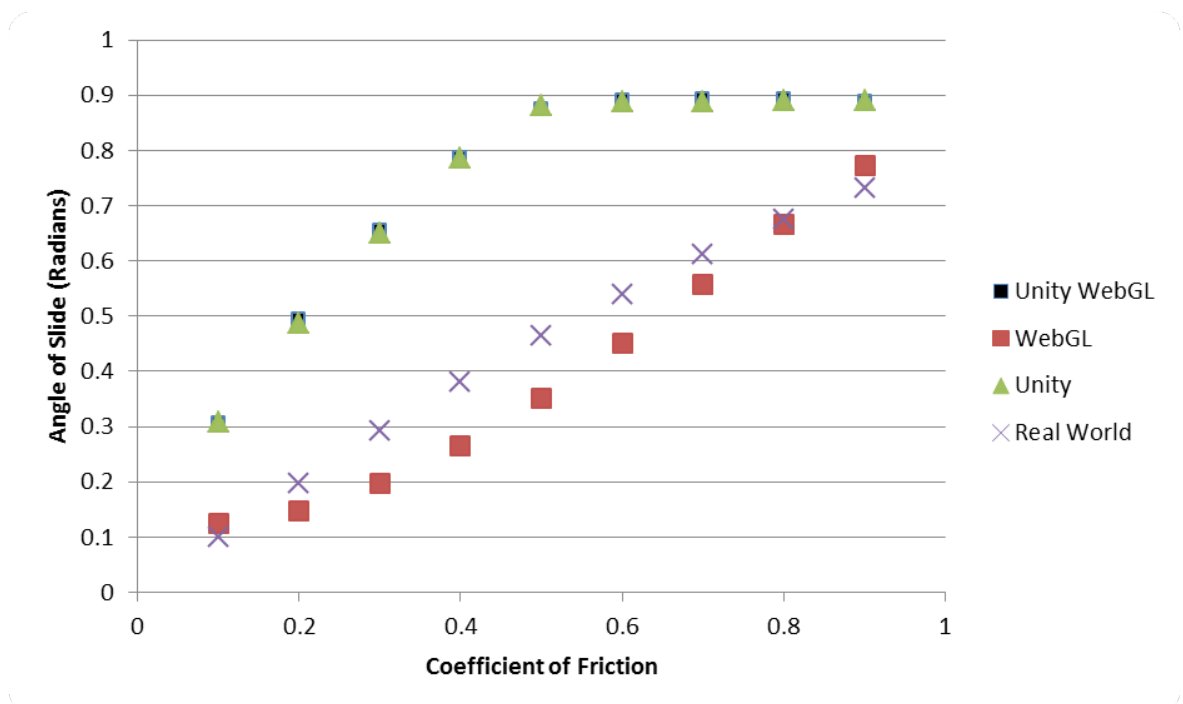
In this section, the results for each experiment and its modifications are interpreted in various forms to provide clarity and reasoning for the results generated.

In these results, *Unity* refers to the Unity Web Plugin version of the test cases; *Unity WebGL* refers to the WebGL build of the test cases in Unity; and *WebGL* refers to the native WebGL implementation using the Bullet/PhysiJS physics engine.

5.1 – Static Friction Test

5.1.1 – Correctness

The data represented in this section was executed on the High End Machine, detailed in 4.3, using the Firefox browser. The real world data comes from the formula detailed in 3.1, using the values for the Coefficient of Friction used in the execution of the simulation.



Graph 1: Correctness of the 3 physics engines compared against real world data - Firefox

This graph provides a surprising result. The results collected for the WebGL execution stayed the closest to the expected real world values for the angle of slide. The Unity executions, however, in both WebGL and Web Plugin forms, provided much higher results than expected. At the coefficient

of friction value of 0.5, the plane rotated to such an angle that the box began to fall rather than slide down the plane, giving flat values for the rest of the coefficients in the test thereafter.

To ensure the validity of these results, the configuration of the Unity implementations of the simulation were examined. As mentioned before in section 4.1, Unity uses a Friction Combine variable to set how the coefficient of friction between any two physics objects is generated. For these tests, we used Multiply to ensure that the correct value for the coefficient that we expected was used. To ensure this result was correct, we need to examine the results for this correctness test using the other variables.

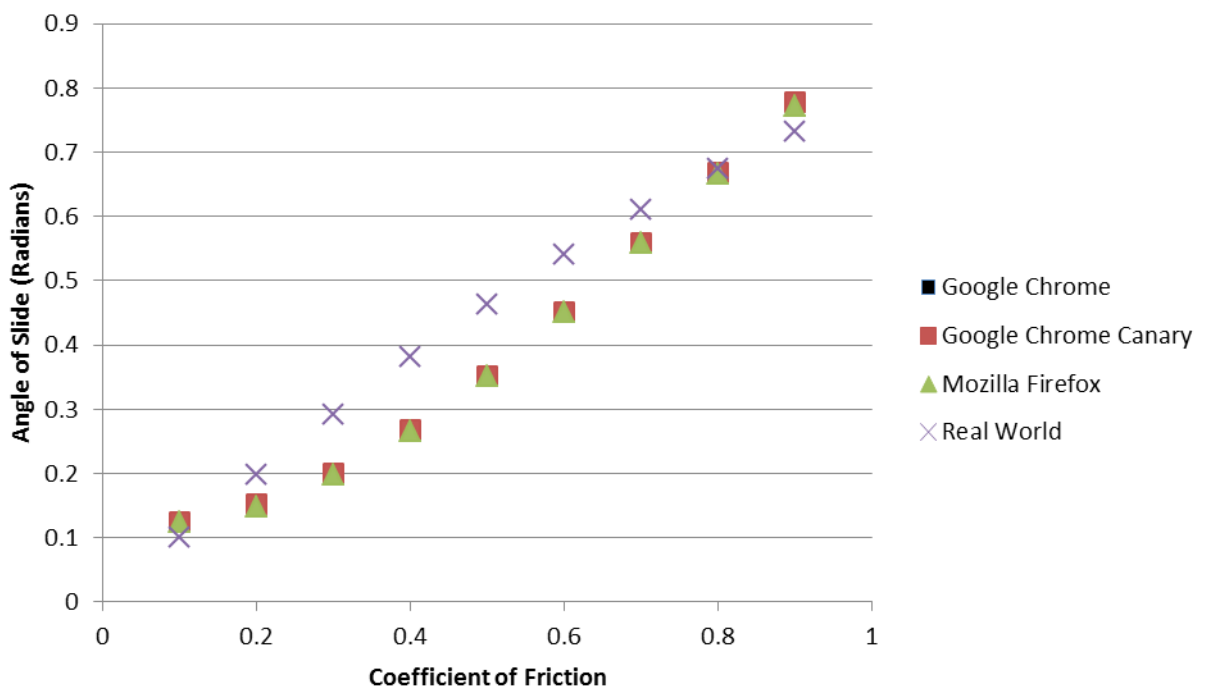
Despite the disparity of Unity's representation of friction from the real world frictional results, it is possible to see from the results if there are any differences in Unity's representation of physics in both the Unity Web Player game and the Unity WebGL game. From the results collected, it appears that any difference in the results between both the games is within the margin of error, therefore there is no difference between the Unity Web Player and the Unity WebGL builds of the game. This will be tested later in the Car Test to ensure this fact holds for all tested scenarios.

5.1.2 – Browser Influence

In these results, we will compare each scenario implementation, across each browser tested, separately. As before, the data was generated on the High End Machine.

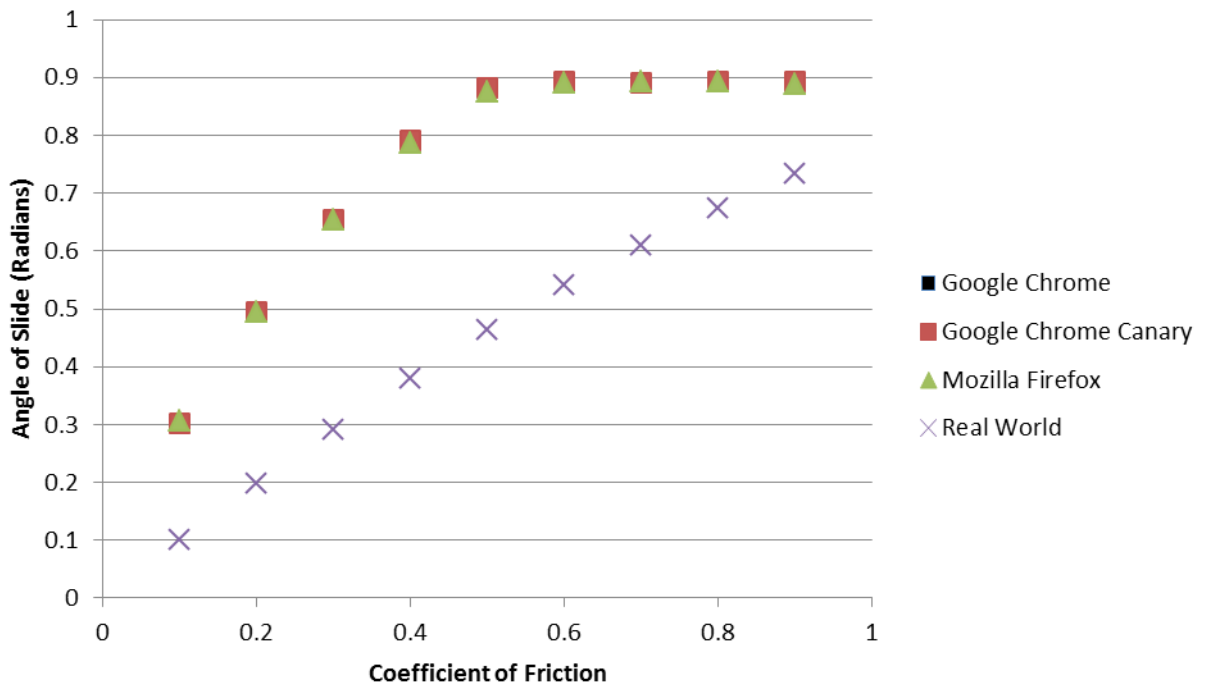
WebGL

As the WebGL implementation, with Bullet, proved to be the most accurate physics engine in the previous test, we will first analyse the WebGL results.



Graph 2: Correctness of the WebGL's physics engine compared on all tested browsers

From these results, with coefficient values below 0.8, when the test is executed in all 3 browsers, the scenario is generating results accurate to the real world data provided. Between the stable Chrome version and the unstable Chrome Canary, there is no discernible difference between the results.

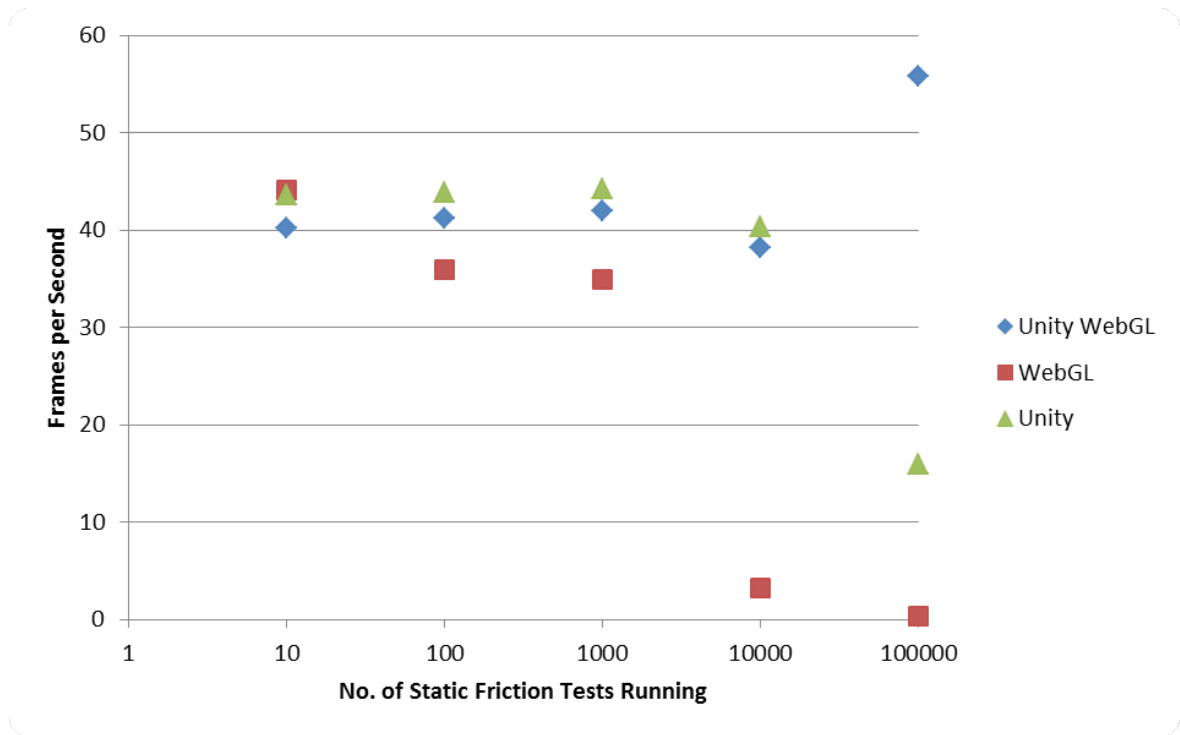


Graph 3: Correctness of Unity's physics engine compared on all tested browsers

With both Unity and Unity WebGL, the results echo the results of the WebGL implementation above. On the same scenario implementation, there is no discernible difference between the results collected between the three browsers.

5.1.3 – Performance

The data represented in this section was executed on the High End Machine, detailed in 4.3, using the Firefox browser.

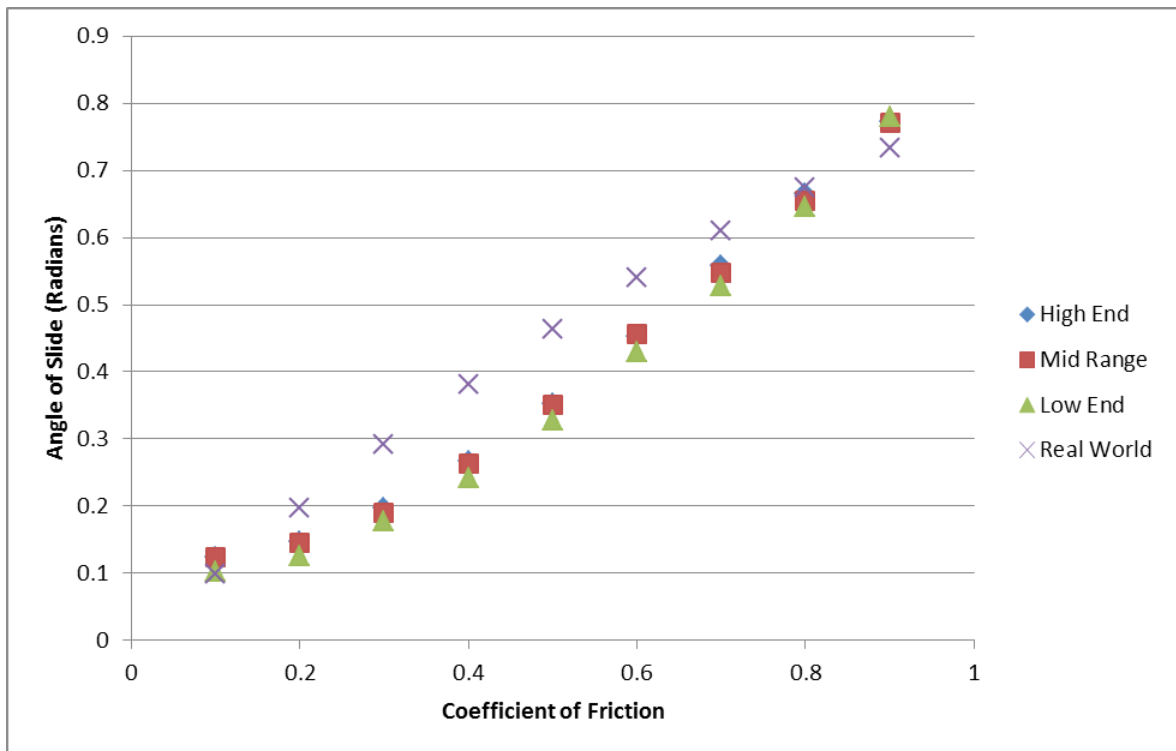


Graph 4: Performance of the tested physics engines with multiple executions of the static friction test

These results show that, for each number of simple friction tests carried out, the Unity Web Plugin implementation consistently provided the best performance from the three implementations tested. Unity WebGL was next, only losing on average 2 frames per second as compared to the Unity Web Plugin version. The WebGL implementation by far provided the worst performance out of all the implementations. At 100 and 1,000 static friction tests being executed, the framework averaged 10 frames per second less than the Unity implementations. At 10,000 the performance of the WebGL plummeted to unsustainable numbers, by recording an average framerate of 3 frames per second.

5.1.4 – Hardware Speed Influence - Accuracy

For these results, the data represented was generated on all hardware outlined in section 4.3. The results for the execution in the following graph were generated in Firefox, using the WebGL implementation due to its results in section 5.1.1.

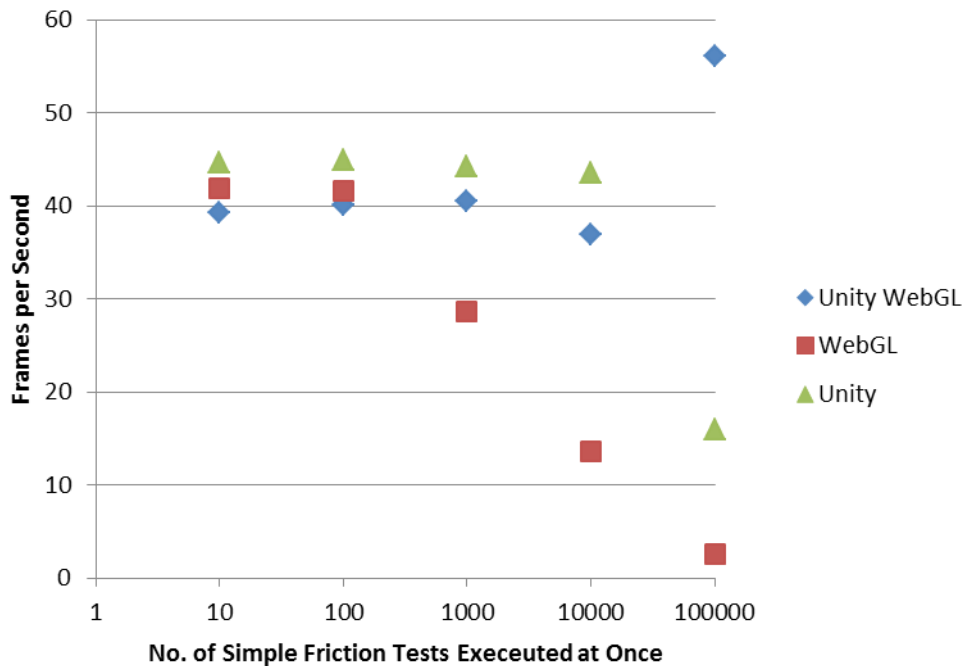


Graph 5: Correctness of the WebGL's physics engine compared on all tested hardware specifications - Firefox

From this, we can see that the results from all three hardware specifications, the hardware has had little – but not no - effect on the accuracy of the physics engines. All three sets of results follow a similar curved path; however the low end results appear to be slightly different than the other results. The results start lower than the Mid-Range and High End results, then follow a curve until the coefficient of 0.8, where the low end results cross the other result's curves and match up. Between the High End and Mid-Range machines, any difference in angles appears to be within the margin of error.

5.1.5 – Graphical Rendering Influence - Performance

The data represented in this section was executed on the High End Machine, detailed in 4.3, using the Firefox browser.



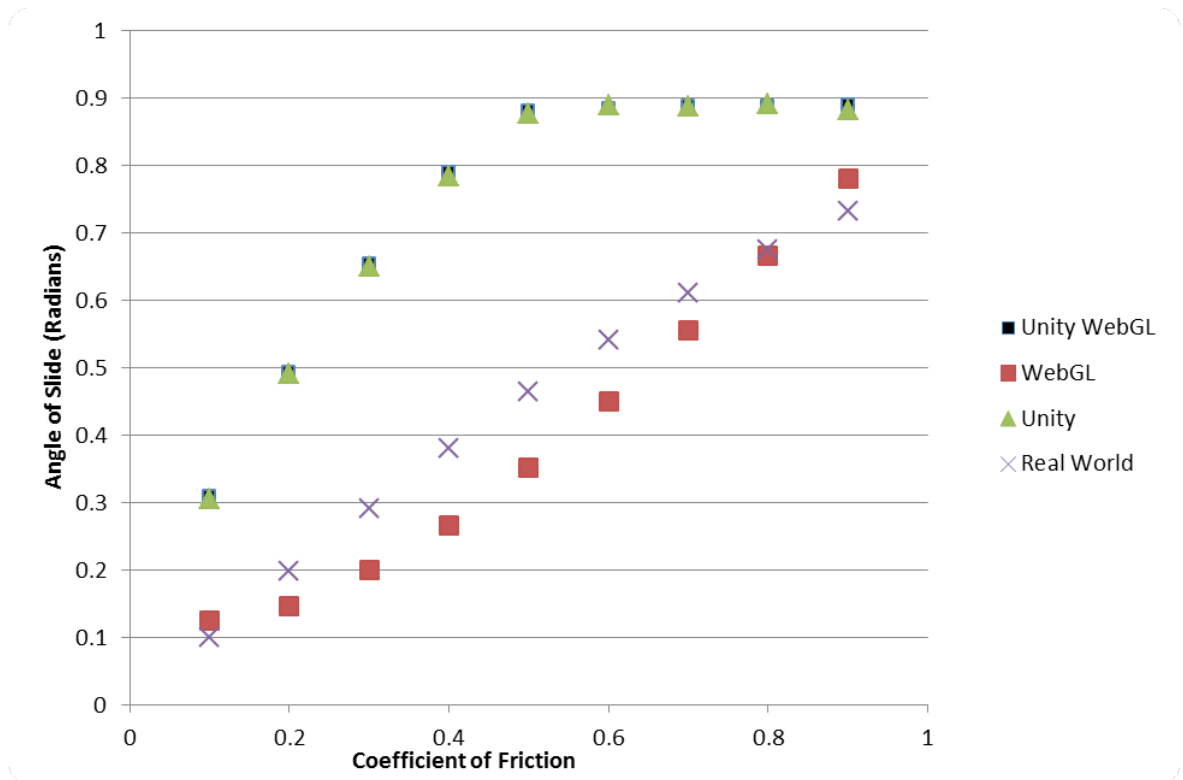
Graph 6: Performance of the tested physics engines with multiple executions of the static friction test – Camera pointed away from executing tests

From this graph, as measured by the framerate, it appears that the rendering of the objects on the screen had little effect to the actual performance of the scenarios. The data collected from the Unity Web Player and Unity WebGL implementations were within the margin of error range, by only fluctuating by a few frames on each number of static friction tests being executed. With the WebGL implementation, it appears that with 10,000 simple friction tests being executed, the performance is much better without the objects being rendered compared to the results from the objects being rendered. At 10,000, this finding appears to no longer apply, so another execution of the automated tests is required to ensure that this is not an erroneous result.

The result for Unity WebGL at 100,000 tests appears to be an issue with the framerate measurement script in the WebGL build. The same script in Unity provides the correct result. In viewing, the framerate at this level is similar to that provided by Unity.

5.1.6 – Graphical Rendering Influence - Accuracy

The data represented in this section was executed on the High End Machine, detailed in 4.3, using the Firefox browser.



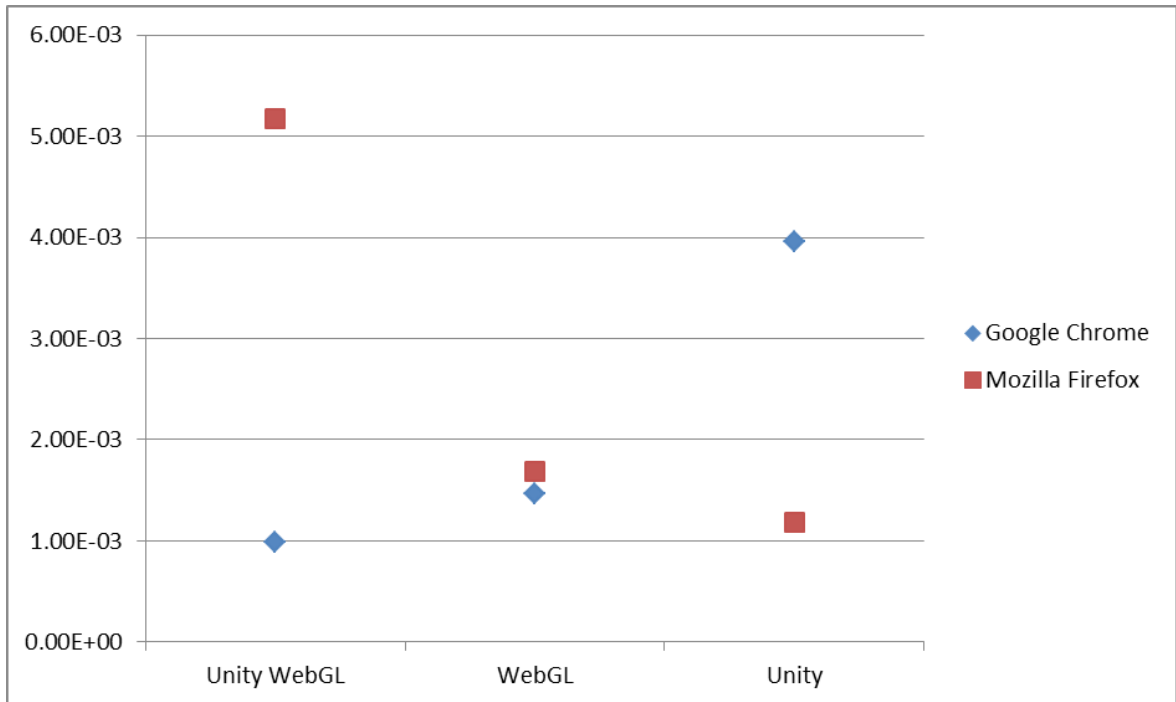
Graph 7: Correctness of the 3 physics engines compared against real world data – Firefox – Camera pointed away from executing tests

From the results shown here, it can be seen that there is no effect on the accuracy of each of the physics engines by removing the rendering of the objects on the screen.

5.2 – Car Test

5.2.1 – Car Drift Correctness

The data represented in this section was executed on the High End Machine, detailed in 4.3, in all three browsers and frameworks.



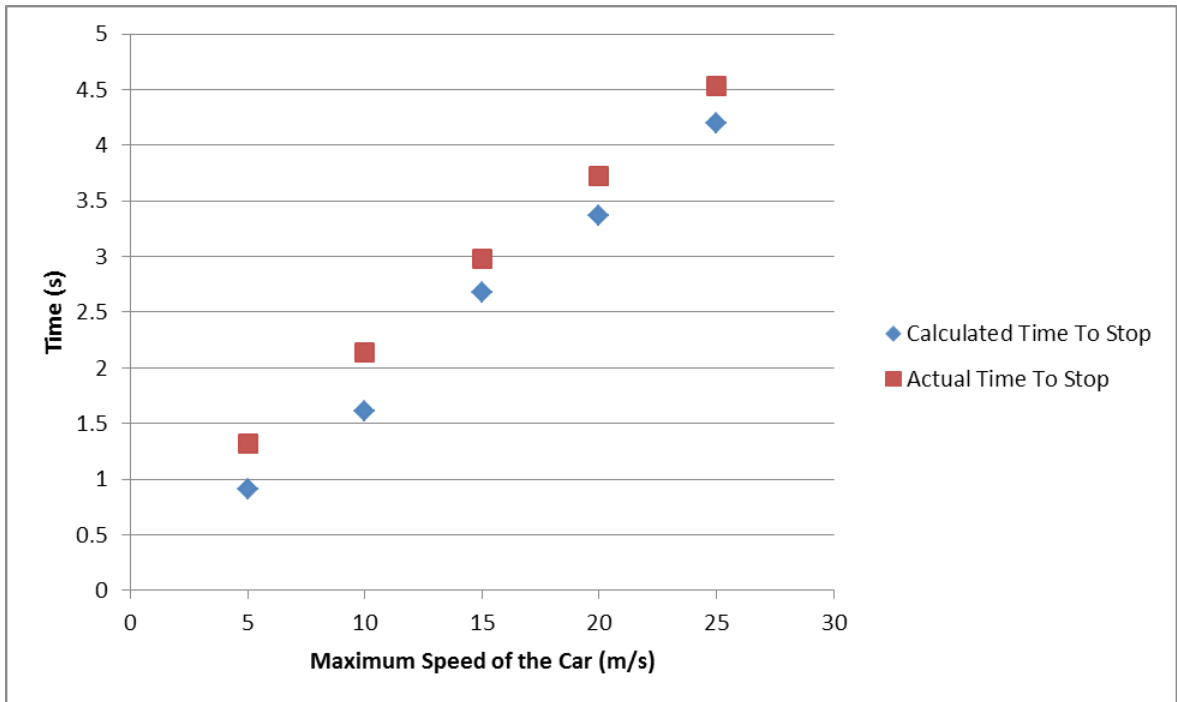
Graph 8: Number of units that the car has drifted from its original driving line

From the appearance of the data in this graph, it would appear that the results for this test are inconclusive, however examining the level of accuracy of the Y axis, it can be seen that the amount of drift that the vehicle has in each framework, in each browser, is remarkably small. While we can see that WebGL is more consistent than either Unity implementation, all three results can be claimed as being within the margin of error and therefore all three can be regarded as an accurate simulation.

5.2.2 – Stopping Time Correctness

The data represented in this section was executed on the High End Machine, detailed in 4.3, using the Firefox browser.

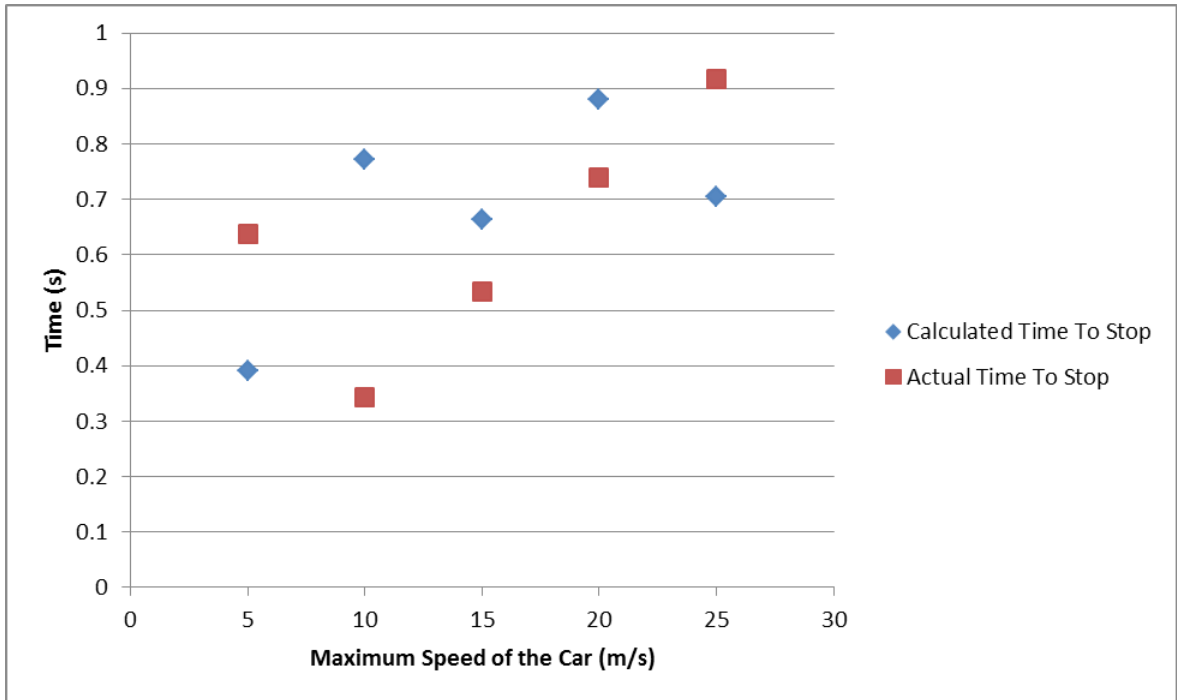
Unity



Graph 9: Graph showing the actual and calculated time to stop the car moving from various speeds

This data shows that there is consistency in Unity's handling of vehicle physics. The stopping times collected were consistently higher than the actual real world time to stop the car by around 0.4 seconds. While Unity's time to stop the car moving is different from the real world time, since it is consistent within its own ecosystem, it can be considered acceptable by the end user playing a game built in the engine.

WebGL

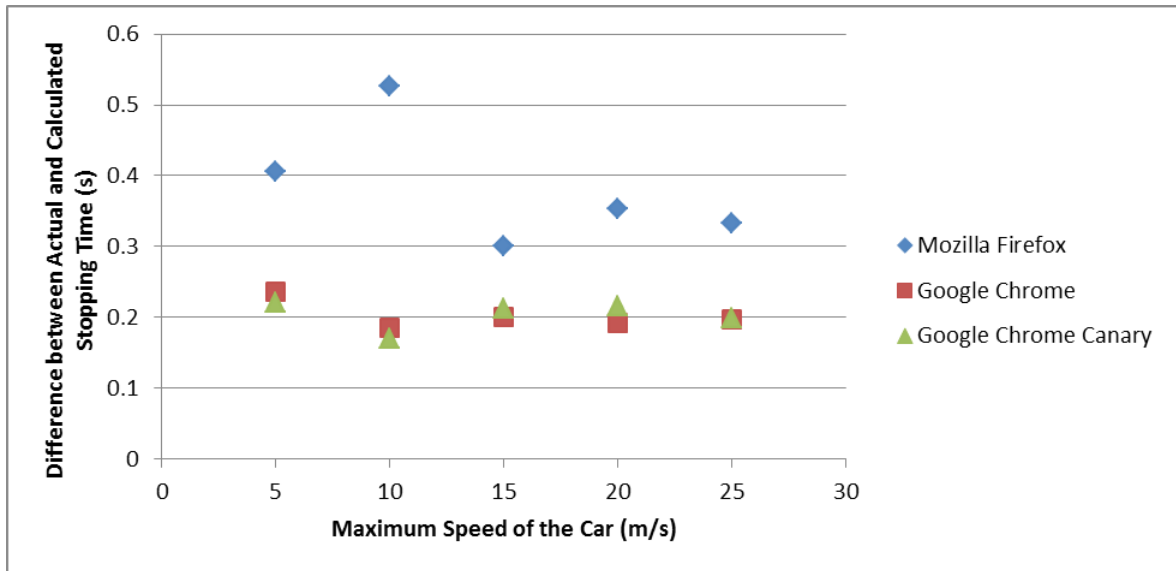


Graph 10: Graph showing the actual and calculated time to stop the car moving from various speeds

This graph shows strange results. The stopping time is not consistent, and neither is the actual time to stop. The actual time to stop is calculated from the acceleration provided from the game engine itself compared to the time it actually took to stop the car, so the actual results could be skewed in this result if the engine is reporting the wrong acceleration to the test suite.

5.2.3 – Browser Influence

The data represented in this section was executed on the High End Machine, detailed in 4.3, using the Unity WebGL data.

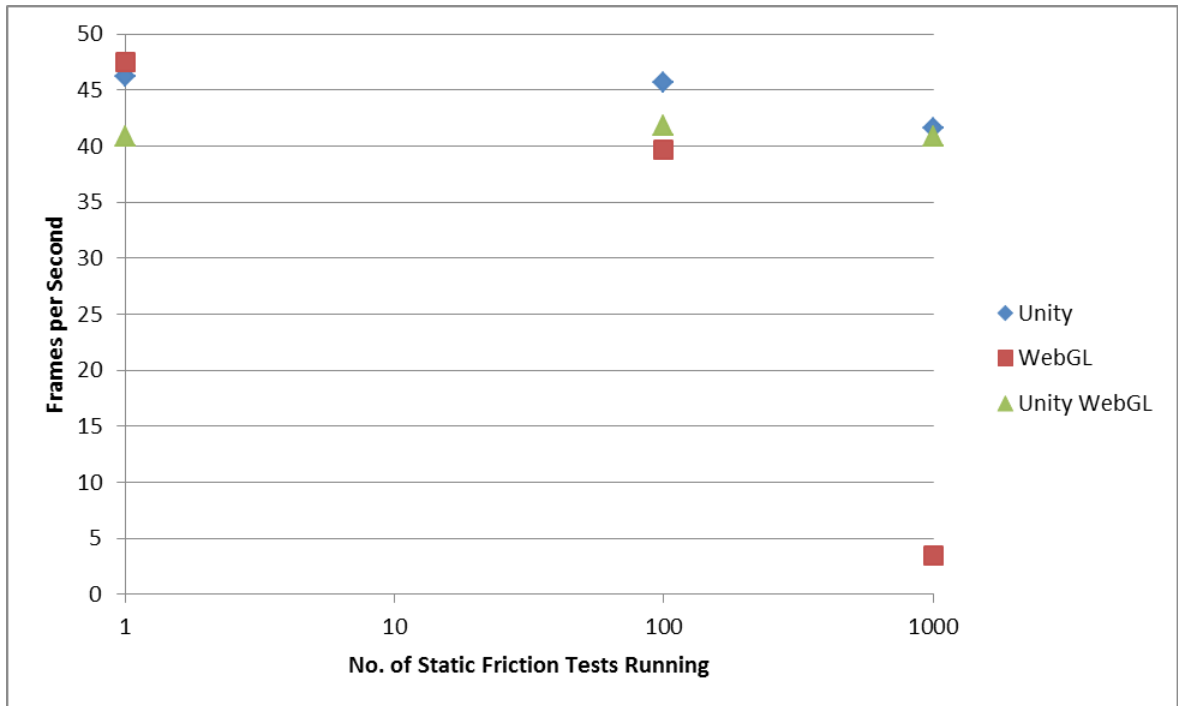


Graph 11: Graph showing the difference in actual and calculated stopping times; on each browser; for Unity WebGL; at multiple different speeds of the car

This data recorded here is interesting. Compared to the browser influence in section 5.1.2, where the browser used had virtually no effect on the results generated from the Simple Friction Test, the data collected from the car test shows a definitive difference between the results generated from Firefox and Google Chrome. While Google Chrome shows around a 0.2 second difference between the real world and the recorded stoppage times, Firefox proved to be much more varied, and consistently worse than the Google Chrome results. Firefox fluctuated between 0.3 and 0.5 seconds difference between the real world and the recorded stoppage times, with no consistency or correlation between the maximum speed achieved by the car and the differences in stoppage times. As has been noticed in prior results, both Google Chrome and Google Chrome Canary record very similar results.

5.2.4 – Performance

The data represented in this section was executed on the High End Machine, detailed in 4.3, using the Firefox browser.

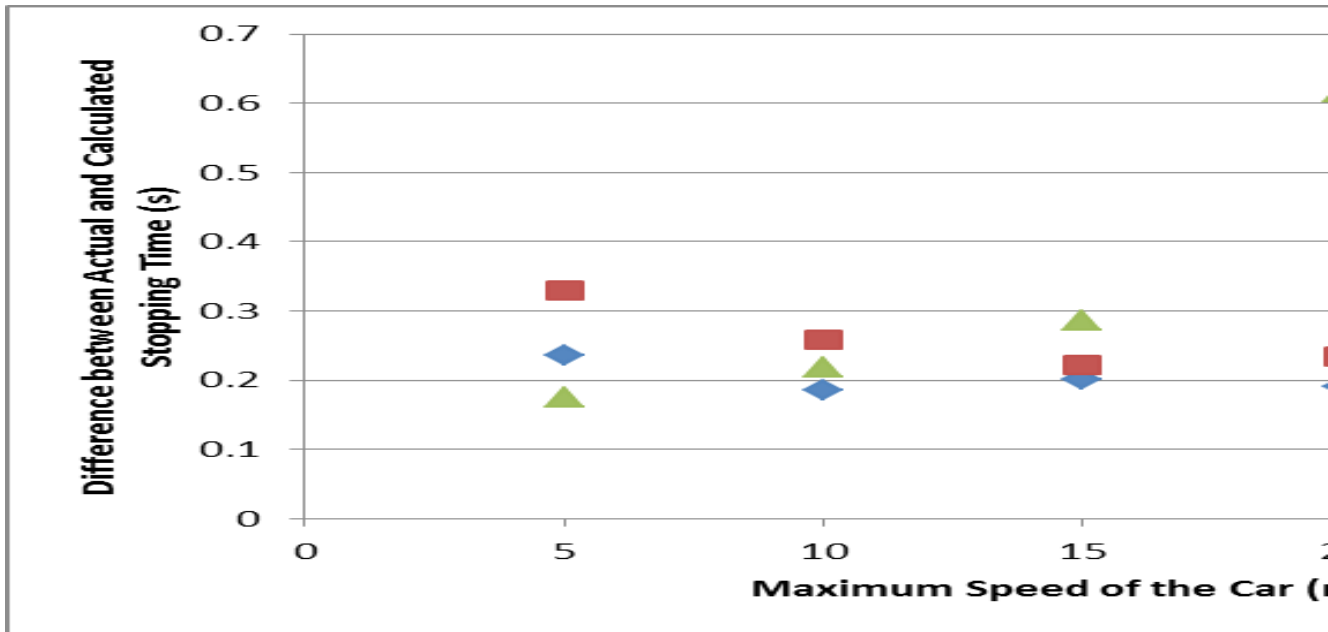


Graph 12: Performance of the tested physics engines with multiple executions of the Static Friction Test while running the Car Test

The results for the performance are similar to that measured in section 5.1.3. The results from Unity Web Player proved to be the most stable under a large number of objects in the physics scene. The framerates measured in this experiment were similar to the framerates in section 5.1.3, with the added overhead of the car test at the lower numbers of static friction tests having little effect. WebGL, however, had a similar but accelerated drop in framerate as compared to section 5.1.3. The framerate started on par with the Unity implementations, however it quickly fell to an unsustainable framerate at 1,000 running static friction tests. This is much worse than the original performance test data, where WebGL ran at around 35 FPS at the same level, and is much worse than its Unity counterparts.

5.2.5 – Hardware Influence – Stopping Time Correctness

The data represented in this section was executed on the High End Machine, detailed in 4.3, using the Firefox browser.



Graph 13: Graph showing the difference in actual and calculated stopping times; on each hardware specification; for Unity WebGL; at multiple different speeds of the car

This shows some interesting results. While the High End machine produced consistent results, the results generated by the Mid Range and Low End machines varied much more than expected. At certain points, the machines produced results similar to the High End machine, other times it produced a difference in real world and calculated times as large as 0.6 seconds. As compared to the results in section 5.1.4, the hardware had measurably more of an effect on the results for the car test than the static friction test.

6 – Conclusion

This section contains a short discussion on the results collected in this paper, as well as a description of further research that can be run, building on the foundation found here.

6.1 – Conclusion from Results

The ambitions of this report were to examine the accuracy and performance of web-based physics engines and solutions. As this conclusion has many different factors involved, it will be concluded using the four research questions asked at the start of the paper:

6.1.1 – Does the web browser have an effect on the performance/accuracy of the physics engine?

From the results in sections 5.1.2 and 5.2.2, the results on this are varied. While it appeared that the browser did not have an effect on the results in the Simple Friction Test, the results of the car test show that there can be a significant difference in the accuracy of a web-based physics engine. From the results here it appears that Google Chrome is the most accurate web browser for WebGL based games, however this is something that can be researched further in other aspects of web-based physics engine research to ensure this is accurate for all possible scenarios.

6.1.2 – Does either physics engine, running on Unity or WebGL, perform accurately to real world physics models?

In the accuracy tests, the physics engines performed with varying results. In the Simple Friction Tests, WebGL, using Bullet.js, proved to be the most accurate physics engine, with its results performing relatively close to the real world data. Unity, however, in both Web Plugin and WebGL forms, was not accurate to the real world data. While the static friction did increase with each Coefficient of Friction, it increased at a much steeper slope than the real world data.

In the Car Tests, the results generated showed that all physics engines in the research had varying levels of inaccuracy to real world physics. In the initial acceleration phase of the test, all three implementations showed signs of drift from the initial direction the cars were pointing in. None of the drift was deemed to be unacceptable in a video game sense, but compared to the accuracy of the real world physics model, the drift was noticed. The results for the stopping time experiments echo this evaluation, with the results for Unity's handling of vehicle physics being slight off from the real world data. From this, we can conclude that the physics engines do not perform at a level of

accuracy to be defined as a recreation of a real world physics model.

6.1.3 – Does the speed of the CPU and GPU of the machine have an effect on the performance and accuracy of the physics engines?

From the results collected, it appears that the speed of the CPU and GPU of the machine do have an effect on the accuracy of the physics engines. In section 5.1.4, it can be seen that the High End and Mid Range machines performed virtually identically to each other. The Low End machine appears to have skewed the angle of slide slightly, with the angle being slightly lower than the other two machines until a coefficient of 0.8. On the car test, however, some of the generated results showed a significant difference between the execution on the High End machine and the lower end machines.

6.1.4 – Do the physics engines run at an acceptable level of performance and accuracy for use in a video game?

Following from the conclusion in section 6.1.2, a conclusion was formed that the physics engines tested were not accurate to the real world physics model. In the Car Test conclusion, however, it was mentioned that none of the results generated were deemed to be unacceptable in terms of a video game. In a video game, some leeway in the real world results can be acceptable as long as the simulation is believable to the player. Often, as shown in the previous research, these inaccuracies in the simulation are to keep the performance of the games developed using the engine at an acceptable level. As shown in this research, Unity proved to be able to attain a constant framerate in all the stress testing that was done, whereas WebGL started to fail under the stress of a large number of objects in the simulation. From this we can state that Unity is the more stable of the physics setups, however WebGL is capable of running a smaller game with usable performance.

6.2 – Further Research

As is shown in the background research, there are many different experiments that can be performed on a physics engine to test its capabilities. As shown by Boeing and Bräunl (2007) [6], experiments can be carried out to examine constraint accuracy, collision detection/contact determination and bounce test to name a few. By carrying these experiments out on the same platforms and engines, while still incorporating the theory that one experiment may have an effect on the performance or accuracy of the other, a more robust conclusion on the physics engines tested can be formed.

A possible extension from this research is to examine how the web-based Unity builds compare to the desktop and console based builds using the same experiments as are used here.

It should be noted also that Unity's ability to build games for WebGL is currently, at the time of writing, in a beta state. As development of the feature continues, the WebGL build of games in the engine may (or may not) become more accurate to the other builds available. Once the feature comes out of beta, it would be worthwhile to rerun these tests to examine if the updates have improved the results measurably.

These experiments were also only run on desktop machines. The major mobile web browsers have full support for WebGL [34] and while they don't support the Unity Web Player, many native video games on mobile platforms are built in Unity also. It would be worthwhile to run these experiments on Android, iOS and Windows Mobile to examine if the results from those browsers and platforms are able to match up to the results collected here.

References

- [1] J. Joshi, W. Aref, A. Ghafoor, H. Spafford, "Security Models for Web Based Applications", ACM, 2001
- [2] J. Oentrich, "The German Gaming Industry: Europe's Biggest Gaming Market", Germany Trade & Invest, 2012
- [3] Science, Bina Nusantara University, Jakarta, Indonesia, 2014
- [3] J. Nielson, C. Williamson, M. Arlitt, "Benchmarking Modern Web Browsers", Department of Computer Science, University of Calgary
- [4] P. S. Paul, S. Goon, A. Bhattacharya, "History and Comparative Study of Modern Game Engines", Birla Institute of Technology, Kolkata Campus, India Institute of Engineering & Management
- [5] R. Yogya, R. Kosala, "Comparison of Physics Frameworks for WebGL-Based Game Engine", School of Computer Science, Bina Nusantara University, Jakarta, Indonesia, 2014
- [6] A. Boeing, T. Bräunl, "Evaluation of real-time physics simulation systems", Association for Computing Machinery, Inc. ., Perth, Western Australia, 2007
- [7] N.J.S Gorst, S.J. Williamson, "Friction in temporary works", The University of Birmingham, Birmingham, United Kingdom, 2003
- [8] N. Andersson, P. Hugoson, J. Jagelčák, S. Sökjer-Petersen, "Inclination tests to determine the static friction factor for different material combinations", MarITerm AB, Sweden, 2007
- [9] J. K. Martinsen, H. Grahn, A. Isberg "A Comparative Evaluation of JavaScript Execution Behavior", Sony Ericsson Mobile Communications AB, Sweden, 2010
- [10] A. Seugling, M. Rolin "Evaluation of Physics Engines and **Implementation** of a Physics Module in a 3d-Authoring Tool", Umeå University, Sweden, 2006
- [11] A. Mannerfelt, A. Schrab "Automatic Test Suite for Physics Simulation System", Linköpings Tekniska Högskola, Campus Norrköping, 2006
- [12] I. Milligan , "Introduction in Game Physics Engine Development", Morgan Kaufmann Publishers San Francisco, CA, 2007
- [13] M. Labschutz, K. Kroszl, "Content Creation for a 3D Game with Maya and Unity 3D", Vienna University of Technology, Vienna / Austria, 2010

- [14] WebKit, "SunSpider JavaScript Benchmark", <http://Webkit.org/perf/sunspider-0.9/sunspider.htm>
- [15] UptoDown, "Browser comparison: Chrome 34 vs Firefox 29 vs Explorer 11 vs Opera 20", <http://blog.en.uptodown.com/browser-comparison-chrome-firefox-explorer-opera/>
- [16] Physics Abstraction Layer, "PAL : Physics Abstraction Layer : Home", <http://www.adrianboeing.com/pal/index.html>
- [16] Boston University, "Friction", <http://physics.bu.edu/~duffy/py105/Friction.html>
- [17] CMO By Adobe, "ADI Report: Google Controls The Browser Worldwide", http://www.cmo.com/articles/2014/6/2/adi_2014_browser_war.html
- [18] StatCounter Global Stats, "Top 9 Browsers on Feb 2015", <http://gs.statcounter.com/#all-browser-ww-monthly-201502-201502-bar>
- [19] SeleniumHQ, "Selenium – Web Browser Automation", <http://www.seleniumhq.org/>
- [20] Java Tips, "How to use Robot class in Java", <http://www.java-tips.org/java-se-tips/java.awt/how-to-use-robot-class-in-java.html>

- [21] three.js, "three.js - Javascript 3D library", <http://threejs.org/>
- [22] Unity (Game Engine), Available: [http://en.wikipedia.org/wiki/Unity_\(game_engine\)](http://en.wikipedia.org/wiki/Unity_(game_engine))
- [23] Documentation, Unity Scripting, Languages and You, Available: <http://blogs.unity3d.com/2014/09/03/documentation-unity-scripting-languages-and-you/>
- [24] WebGL, "OpenGL ES 2.0 for the Web, Available" <https://www.khronos.org/webgl/>
- [25] Documentation, Unity Scripting, Languages and You, Available: <http://www.techradar.com/news/software/applications/why-microsoft-decided-to-put-webgl-into-internet-explorer-1167110>
- [26] Unity, "Benchmarking Unity performance in WebGL": <http://blogs.unity3d.com/2014/10/07/benchmarking-unity-performance-in-webgl/>
- [27] Chromium Blog, "The Final Countdown for NPAPI": <http://blog.chromium.org/2014/11/the-final-countdown-for-npapi.html>
- [28] Autodesk, "3D Animation and modelling Software | Maya | Autodesk": <http://www.autodesk.com.au/products/maya/overview>
- [29] Unity, "Unity – Get Unity": <http://unity3d.com/get-unity>
- [30] Thought Spike, "Friction Coefficients for Bullet Physics": <https://www.thoughtspike.com/friction-coefficients-for-bullet-physics/>
- [31] JetStream, "In Depth Analysis": <http://browserbench.org/JetStream/in-depth.html>
- [32] Unity, "High-performance Physics In Unity 5": <http://blogs.unity3d.com/2014/07/08/high-performance-physics-in-unity-5/>
- [33] Unity, "On The Future of Web Publishing in Unity 5": <http://blogs.unity3d.com/2014/04/29/on-the-future-of-web-publishing-in-unity/>
- [34] Dave Voyles, "The current state of WebGL – iOS 8, Yosemite, Android Lollipop": <http://www.davevoyles.com/current-state-webgl-ios-8-yosemite-android-lollipop/>

All Internet resources have been checked for availability on 11/05/2015

7 – Appendices

7.1 – Raw Data

7.1.1 – Static Friction Test

Coefficient of Friction	Angle of static slide
0.1	0.0996686525
0.2	0.1973955598
0.3	0.2914567945
0.4	0.3805063771
0.5	0.463647609
0.6	0.5404195003
0.7	0.6107259644
0.8	0.6747409422
0.9	0.7328151018

Table 3: Real world accuracy results for the Simple Friction Test, calculated using the formula defined in 3.1. Angles measured in radians.

Google Chrome				Google Chrome Canary		
	Unity WebGL	WebGL	Unity	Unity WebGL	WebGL	Unity
0.1	0.3054326	0.12400001	0.3054326	0.3019423	0.12400001	0.3054326
0.2	0.493928	0.15	0.4904376	0.4956734	0.15200002	0.4904376
0.3	0.656245	0.20000008	0.6527535	0.6544995	0.20000008	0.6527535
0.4	0.7906366	0.26800004	0.7871441	0.7906366	0.26800004	0.7871441
0.5	0.8813934	0.35200006	0.8796464	0.8813934	0.35200006	0.8796464
0.6	0.891865	0.44999993	0.8901184	0.8936104	0.45199993	0.8901184
0.7	0.8936104	0.55799997	0.8918633	0.89012	0.55799997	0.8883725
0.8	0.8953553	0.66799986	0.8918633	0.8936104	0.66799986	0.8918633
0.9	0.8936104	0.7760001	0.8883725	0.891865	0.77800006	0.8883725

Mozilla Firefox			
	Unity WebGL	WebGL	Unity
0.1	0.3054326	0.12400001	0.3071775
0.2	0.493928	0.14800002	0.4869468
0.3	0.6544995	0.19800007	0.6510081
0.4	0.7871463	0.26600003	0.7853987
0.5	0.8744116	0.35200006	0.8813918
0.6	0.89012	0.45199993	0.8883725
0.7	0.891865	0.55799997	0.8883725
0.8	0.891865	0.6659998	0.8901184
0.9	0.8883746	0.772	0.8901184

Table 4: Accuracy results for the Simple Friction Test, measured on the High End Machine. Angles measured in radians.

	Google Chrome			Google Chrome Canary		
	Unity WebGL	WebGL	Unity	Unity WebGL	WebGL	Unity
10	58.658543	56.571953	54.094025	58.76665	56.72146	55.990303
100	57.12614	57.03235	55.910862	55.746475	56.361034	56.247925
1000	57.06798	55.5812	55.631294	56.77624	48.040527	56.37349
10000	21.456516	6.149375	55.428116	20.04832	5.635737	55.56543
100000	21.9101	0.6519026	22.022934	24.276901	5.912758	21.609535

Mozilla Firefox			
	Unity WebGL	WebGL	Unity
10	40.197823	44.159977	43.553265
100	41.25895	35.890247	43.843822
1000	42.02257	34.937935	44.257935
10000	38.228786	3.213934	40.312588
100000	55.8553	0.38729668	15.939803

Table 5: Performance results for the multiple iterations of the Simple Friction Test, measured on the High End Machine. Numbers are measured in Frames per Second.

	Google Chrome			Chrome Canary			Mozilla Firefox		
	Unity WebGL	WebGL	Unity	Unity WebGL	WebGL	Unity	Unity WebGL	WebGL	Unity
No. Static Tests	0								
Stopping Speed	5								
Car drift from straight line	6.23E-04	-1.13E-04	1.97E-04	1.87E-04	-1.18E-04	2.17E-04	1.45E-04	-1.26E-04	2.14E-04
FPS	58.29789	57.43299	56.462627	56.376526	56.07276	56.49186	40.65913	42.294884	50.669918
Max Velocity (m/s)	5.030229	5.077167	5.055593	5.280895	5.4345484	5.33882	5.207492	5.6691637	5.299725
Average Acceleration Stopping (m/s ²)	-5.7369347	-13.148283	-5.79309	-5.7417727	-10.62313	-5.79049	-5.718818	-14.489223	-5.803741
Calculated Time Taken to Stop	0.8768148	0.38614678	0.8726937	0.9197326	0.51157695	0.92199796	0.9105888	0.39126763	0.9131567
Actual Time Taken to Stop	1.0481511	0.3114183	1.1092031	1.120224	0.24186055	1.1423565	1.2586585	0.6385904	1.319279

Table 6: Example of the results collected for the Car Test, at a specific number of static tests running and a specific stopping speed

	Google Chrome	Mozilla Firefox	Google Chrome Canary
Unity WebGL	9.88E-04	0.005178825	1.87E-04
WebGL	0.001459848	0.001687349	-1.18E-04
Unity	0.003953894	0.001187427	2.17E-04

Table 7: Table showing the number of units that the car has drifted from its original driving line in the 3 physic engines

	Calculated Time To Stop	Actual Time To Stop
5	0.9131567	1.319279
10	1.614758	2.1409702
15	2.681333	2.982682
20	3.372646	3.726395
25	4.203177	4.535835

	Calculated Time To Stop	Actual Time To Stop
5	0.39126763	0.6385904
10	0.77097815	0.34403422
15	0.66291064	0.5332499
20	0.88065976	0.74011654
25	0.70394665	0.91714025

Table 8: Tables showing the calculated and actual time for the car to stop at various speeds in Unity and WebGL respectively

	Mozilla Firefox	Google Chrome	Google Chrome Canary
5	0.4061223	0.2365094	0.22035854
10	0.5262122	0.185356	0.170681
15	0.301349	0.2008259	0.2128033
20	0.353749	0.1912029	0.2156379
25	0.332658	0.1971934	0.1992013

Table 9: Table showing the difference in actual and calculated stopping times; on each browser; for Unity WebGL; at multiple different speeds of the car

	Unity	WebGL	Unity WebGL
1	46.17353	47.4742	40.879086
100	45.64145	39.67759	41.770744
1000	41.58865	3.462861	40.88576

Table 10: Table showing the performance of the tested physics engines with multiple executions of the Static Friction Test while running the Car Test

	High End	Mid Range	Low End
5	0.2365094	0.3297093	0.1758185
10	0.185356	0.2582076	0.2186454
15	0.2008259	0.2222583	0.2866377
20	0.1912029	0.2336895	0.6164748
25	0.1971934	0.6254053	0.2377672

Table 11: Table showing the difference in actual and calculated stopping times; on each hardware specification; for Unity WebGL; at multiple different speeds of the car

7.1.2 – Minutes from Meeting