JANUARY 1, 2022

PROJECT REPORT

COMPARING PERFORMANCES OF HAND-BUILT AND GAME ENGINE DRIVEN PHYSICS SIMULATIONS

FERGAL BERNARD

Supervised by Dr. Andrew Hague

Abstract

Physics simulations are a foundation of all modern game engines, as they are indispensable for maintaining the illusion that a virtual environment is real. However, they are often among the most performance heavy systems involved. This project serves to determine whether this performance cost is efficiently managed by game engines, or if hand-built systems can be better. This is explored through the development of hand-built physics simulations and game engine driven ones, highlighting what optimisations are practical in the former but not in the latter, and comparing how well they perform. The findings show how such optimisations can lead to significantly better performances, even when compared to systems used by highly optimised game engines, showing that the use of engines should not be a default for simulating physics, but an option for when performance is not critical.

Contents

1 Introduction

Game engines, software frameworks that greatly facilitate the development of videogames and virtual environments, are used for more than 60% of existing videogame projects (Doucet, 2022), a percentage which is increasing as game engines implement systems that are increasingly out of reach for small development teams to design in-house. One such frontier of innovation is physics simulation. These are virtual scenes that are designed to mimic physical behaviour from the real world. The purpose of this project is to explore whether the performance provided by the game engines when simulating physics is out of reach for hand-built systems, or if such engines make performance sacrifices for the sake of generality or ease-of-use.

The goal of this project is therefore to implement a custom physics simulator, which can be augmented with optimisations that are not used by game engines, this system should be fairly general, and be one that game engines do focus on to an extent, such as rigidbody (objects that cannot be deformed) dynamics. A simulation implemented in this system could also be developed in a well-chosen game engine, with relevant metrics being analysed and compared. This should suffice to show whether it is worth considering developing a physics simulator from scratch.

2 Project Management

2.1 Management considerations

Before setting out specific methodology for project management, certain considerations must be made. Firstly, it is important to consider developer experience in the field, and how it factors into both research and the development itself. Due to a lack of experience in physics simulations, this project involved no small amount of learning, which should be factored into any time constraints. Moreover, this means, as a lot of knowledge can only be gained through practical work, that a significant amount of research for this project was performed during development. Because of this, adaptability should be a focus as an over-detailed plan should not get in the way of design changes that make sense considering new experience.

Another consideration is the fact that this is a single developer project, so teamwork and communication are not to be prioritised, and communication should only be considered as far as it factors into detailed reporting and proper documentation.

Lastly, the project should be resilient to external factors. It is important to consider that this project is a part of a university module amongst many others, meaning it cannot and will not be the sole focus. This means that external factors may arise that prevent project advancement for a period. Enforcing a strict timetable here would mean risking developer burnout, which would tremendously impact development and should be avoided at all costs.

2.2 Software development methodology

The methodology that best meets these requirements is agile (Fowler & Highsmith, 2001). Although the client facing aspects of agile do not apply so much here (as for all intents and purposes the client is the developer), the principle of "Responding to change over following a plan" makes a lot of sense and fits perfectly with the idea that new experience and research will shape the direction of development. Moreover, the agile principle "Individuals and interactions over processes and tools," coupled with the adaptability, would grant more flexibility and resilience toward external factors, enabling project advancement to be halted for short periods of time to avoid burnout.

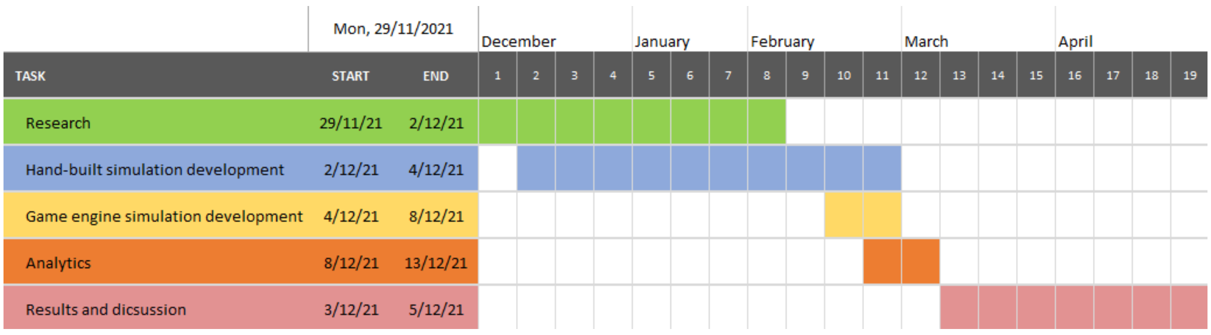
Each week can be considered a development cycle, such that progress can be assessed and planned out at the beginning of each cycle. This allows for new experience to be integrated into the plan on a weekly basis, as well as fitting in well with supervisor meetings.

This process is managed using a Kanban board, permitting progress to be easily tracked and weekly plans to be set out.

Moreover, version control grants a much safer and more efficient way of adding new systems to the project, which is why git is used. This permits simulation optimisations to be integrated and tested safely.

2.3 Project plan

At the time when the final project plan was formulated, after initial project research, it was as shown in figure 1.



	Mon, 29/11/2021		December				January			February				March				April			
TASK	START	END	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Research	29/11/21	2/12/21	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩											
Hand-built simulation development	2/12/21	4/12/21		🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦								
Game engine simulation development	4/12/21	8/12/21										🟨	🟨								
Analytics	8/12/21	13/12/21											🟧	🟧							
Results and dicsussion	3/12/21	5/12/21													🟥	🟥	🟥	🟥	🟥	🟥	🟥

Figure 1: Timeline of different phases of project work

Note that this was shifted slightly after the plan set out in the progress report, which detailed that development should begin a 2 weeks earlier than shown in this chart.

3 Research

3.1 General Systems

3.1.1 Rendering

Rendering systems are not the focus of this project. However, they are necessary for visualising the simulations. In order to avoid rendering having a large performance impact, it should be handled by the GPU. For this, graphics libraries are commonly used, as they provide a way to write shaders (i.e., programs that run on the GPU) and send them to the GPU. The video drivers generally provide these themselves, such as DirectX (exclusively for Windows) and OpenGL/WebGL. Some platforms such as JavaScript can operate at a higher level and do not require shaders to be written and managed directly.

3.1.2 Physics Object Types

In physics simulations, objects are generally assigned one of three types:

1. Static: Objects that have a physical presence in the scene, meaning they can be collided with by other objects, but are unable to exert forces via movement, or be affected by collisions and external forces. In other words, static objects cannot move or be moved, but other object types can collide with them.
2. Kinematic: Objects that can collide with other objects and exert forces upon them via movement but cannot be subject to external forces. These objects can move, but only in predefined ways, not as a result of forces within the simulation itself.
3. Dynamic: Objects that can collide with other objects, and be affected by such collisions, and any other external forces.

In the real world, all objects are dynamic. For example, a ball dropped to the ground will exert a force on the planet itself through collision (and gravitational attraction), but these forces are negligeable. Instead of simulating these forces, a large performance increase can be gained by simply ignoring collisions between objects that are considered immovable or unstoppable. This means that kinematic and static objects require significantly less computation.

Game engines such as Unity permits users to specify physics object types (Unity 3, 2022).

3.1.3 Object Movement

By far the simplest and most common way of representing movement in a physics-based system is the explicit Euler method. It involves describing each entity using three vectors of the same dimensionality as the environment itself. They are position, velocity, and acceleration. These values are updated every frame using the values from the previous frame. Position is updated as follows: $p_{t_n} = p_{t_{n+1}} + (v_{t_n} * \Delta t)$

Where p_{t_n} is the position at time t_n, v_{t_n} is the velocity at time t_n, and Δt is the difference between t_n and t_{n+1}. This can be thought of as moving an entity by its velocity scaled by the time until the next frame. If position is measured in metres, velocity in metres/second and time in seconds, the unit equation can be expressed as $m = m + \frac{m}{s} * s$ which can be simplified to as $m = m + m$, confirming that the equation is correct in terms of units. Velocity is updated similarly as follows:

$$v_{t_n} = v_{t_{n+1}} + (a_{t_n} * \Delta t)$$

Where a_{t_n} is the acceleration at time t_n. Intuitively, this works the same way as position, as velocity is set to the previous velocity plus the acceleration scaled by the time until the next frame. To verify correctness, the unit equation can be written as $\frac{m}{s} = \frac{m}{s} + \frac{m}{s^2} * s$ with acceleration measured in metres/second squared. This simplifies to $\frac{m}{s} = \frac{m}{s} + \frac{m}{s}$, confirming unit correctness.

Lastly, Newton's second law tells us that acceleration is equal to the sum of incoming forces divided by mass (i.e., $a = \frac{\sum F}{M}$). In most simple physics simulations, there are no incoming forces other than gravity that are considered, which is already represented as an acceleration (for example on earth gravity is 9.807 m/s^2). This means, acceleration can simply be set to a constant which controls the strength of the simulation's gravity.

This method is not without issues, however. The main problem that arises comes from the discretisation of the simulation. Because positions and velocities are computed directly for each frame using previous values, any force changes that occurred between frames will be inaccurately represented by this system. The area where this poses greatest issue is during collision, which can be solved using continuous collision as discussed in following sections. However, this can also cause issues when simulating soft-bodies (i.e., deformable objects such as springs), or any system where a movement is expected to repeat periodically, for example a ball bouncing without loss of energy. The reason for this is that inaccuracies produced by the explicit Euler method (because of how it simulates in discrete timesteps) can build up over time. This causes soft-bodies, which are held together by springs, to fall apart.

The widely used solution to this problem is the Runge-Kutta family, specifically Runge-Kutta order 4 (Butcher, 1964). These methods involve iteratively solving entity vectors over many sub-steps, producing a more accurate result. Because this project involves rigid-body simulation, and the discretisation problem surrounding collision can be solved by implementing continuous collision, this method provides little benefit compared to the added complexity of its implementation, so will not be further explored.

3.1.4 Collision Detection

Collision detection is the process of finding pairs of intersecting objects, which can then be fed into collision resolution. It is generally broken down into two phases: broad-phase and narrow-phase. The broad-phase is a fast but inaccurate system for finding pairs of potentially intersecting objects, false positives are tolerated so long as they provide a performance increase. The narrow-phase is instead a much more computationally intensive but highly accurate filter for the broad-results, false positives should never occur (LaValle, 2012).

3.1.4.1 Broad-phase

The most rudimentary broad-phase algorithm is the naïve broad-phase, which simply passes all unique pairs of objects to the narrow-phase. This is essentially assuming that every object is possibly intersecting with every other object and has an average time complexity of

$O(n^2)$. This running time is too inefficient for large scale simulations, especially considering that narrow-phase algorithms require more significant computation per object.

The algorithm used by Unity's implementation of Nvidia PhysX and Box2D (and most other physics engines) is SAP (Sweep And Prune) (Unity 1, 2020) (Unity 2, 2020).

For SAP, the AABB (Axis-Aligned Bounding Box) must first be computed for each object. This is a minimal (or close to minimal) box that entirely contains the object and is aligned with the global axes. This process has a linear time complexity in the number of vertices of a polygon. It involves finding the most extreme vertex coordinates for each axis (see figure 2).

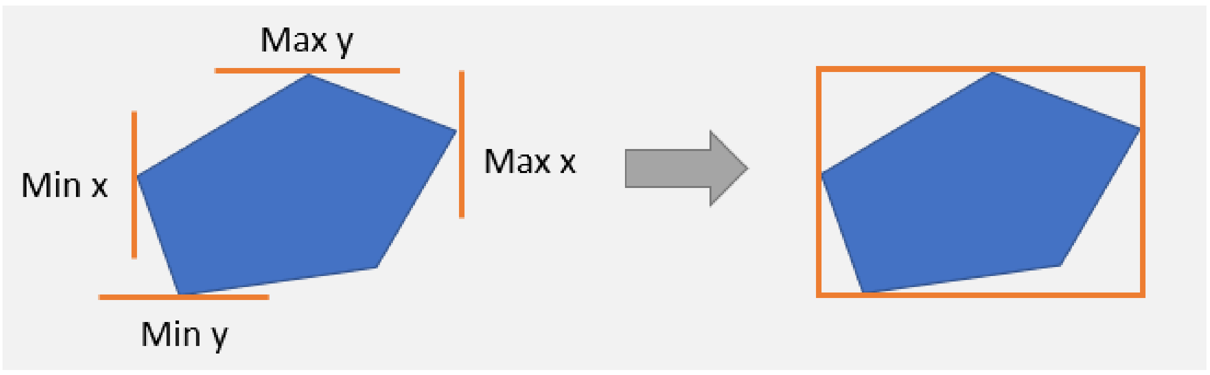


Figure 2: Axis-Aligned Bounding Box diagram for a simple polygon

The AABBs are then sorted along one axis, the result of which is traversed, identifying any intersecting intervals (for example along the x axis the interval would be from the minimum x of a polygon to the maximum x of a polygon). Any such intersections are flagged as possible collisions (see figure 3).

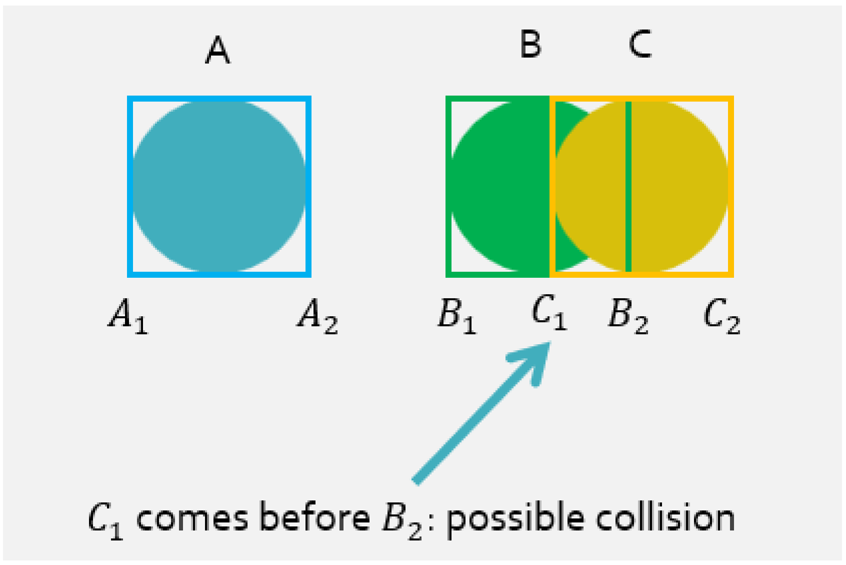


Figure 3: Example of intersecting intervals during SAP

The average time complexity of this algorithm is $O(n \log n)$, and is dominated by the sorting step. In cases where many objects in the simulation are not moving this can be improved by leveraging sleeping objects and maintaining a persistent sorted array of AABBs.

When an object's velocity is zero for a set length of time, it is set to a "sleeping" state. In this state, the object's AABB start and end points can remain in a persistent sorted array, as they cannot change while the object is not moving. This means that only AABB start and end points of objects that are moving need to be updated in such an array every frame. This can be performed even quicker when a sorting algorithm such as insertion sort is used, which has high efficiency for almost-sorted arrays. There are other techniques that allow moving objects to persist in the array (Tracy, Buss, & Woods, 2009), but generally these techniques provide speed increases for simulations when a significant portion of objects can be sleeping at any given time.

Another optimisation for SAP that is commonly implemented is one that fixes a major flaw with the algorithm: objects that are far apart can be flagged as possibly colliding if they are close along one axis (see figure 4).

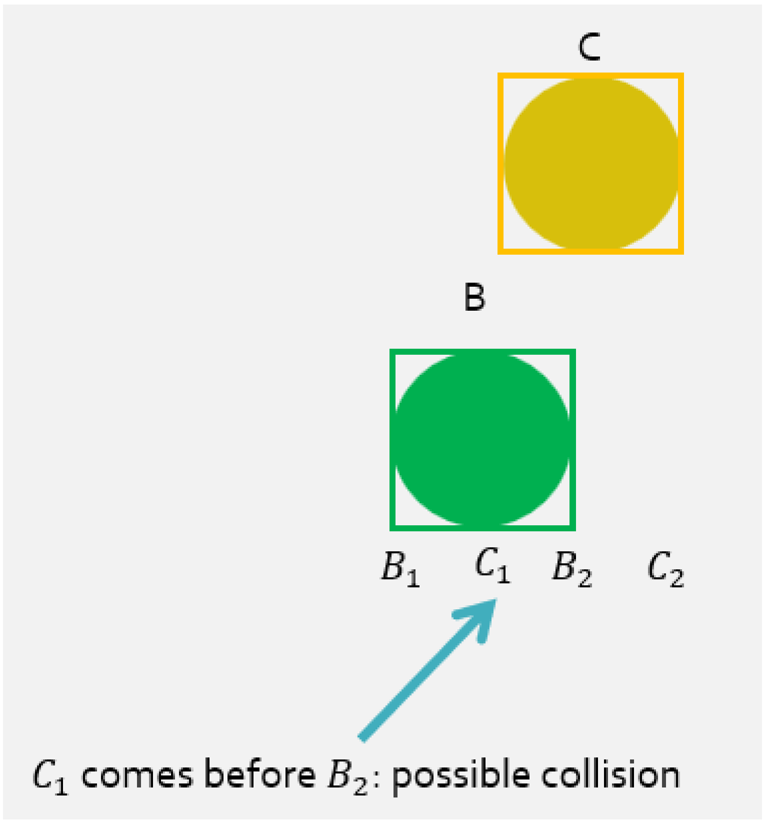


Figure 4: Example of far apart objects being found to be colliding using SAP

To reduce the number of false positives caused by this issue, AABBs can be sorted along multiple axes, providing significantly more accurate results. This is beneficial in simulations where the narrow-phase is very slow, meaning the cost of testing many false-positives in the narrow-phase outweighs the cost of performing SAP multiple times.

Other broad-phase algorithms generally fall under the umbrella of space-partitioning. The simplest of these is grid broad-phase. This technique involves dividing the simulation space into a grid, and only sending pairs of objects that share a cell to the narrow-phase, with objects overlapping grid cells considered to be in both cells. These techniques are not as widely used as SAP, but are sometimes used in conjunction with SAP, such as Nvidia PhysX's MBP (multi-box pruning) which divides the simulation space into a grid, and performs SAP in each cell (Unity 1, 2020).

3.1.4.2 Narrow-phase

For the narrow-phase, algorithms are chosen based on the types of objects being simulated. For convex polygons, GJK (Gilbert–Johnson–Keerthi) or MPR (Minkowski Portal Refinement) are two popular algorithms. MPR is faster and simpler to implement but does not efficiently compute penetration depth for a collision, whereas the more complex GJK does (Newth, 2013). Penetration depth can be used for highly accurate collision response, especially for soft-bodies, neither of which are the focus of this project, therefore only MPR will be considered.

For simplicity, the following steps are considered in two dimensions. The first step of MPR for two polygons is to calculate their Minkowski difference. The Minkowski difference of two polygons A and B, defined as A – B, is the region swept by A translated to every point negated in B (see figure 5).

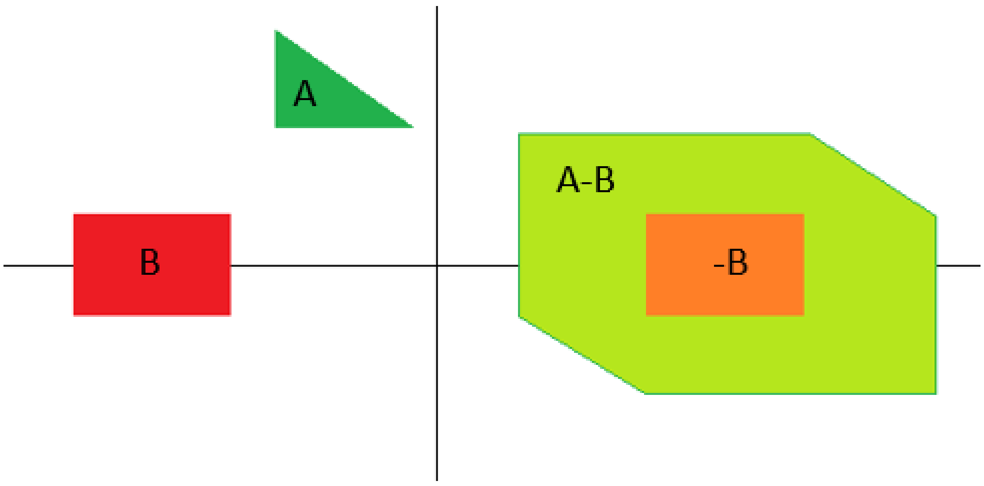


Figure 5: Minkowski Difference of two polygons

This difference has the useful property of containing the origin if and only if the two polygons are intersecting (Kockara, Halic, Iqbal, Bayrak, & Rowe, 2007). The rest of the process therefore involves detecting whether this is the case. This involves constructing a ray from an interior point of the Minkowski difference (usually the geometric centre) that passes through the origin. A "portal" is then found (a line segment that the ray passes though that connects 2 vertices of the polygon) which is then iteratively refined by being moved closer and closer to the surface of the polygon until either the origin is found to be between the portal and the interior point, in which case the origin is inside the polygon, or the algorithm encounters a minimal portal (one that is on the surface of the polygon that connects 2 adjacent vertices) with the origin lying on the opposite side (i.e. outside the polygon).

This narrow-phase works for all convex polygons, which for most use cases is enough. By limiting the use case further, much faster systems can be used. For instance, detecting whether two circles intersect is trivial. In this case, the narrow-phase simply involves checking whether the distance between both circles is greater than the sum of their radii. Therefore, game engines encourage the use of primitive colliders (i.e., sphere, cube, or capsule) for which the narrow-phases are all fast (Unity 1, 2022).

A common system to enable users to optimise simulations in game engines is collision layers. This allows a user to define which groups of objects can collide with each other, granting the system the power to effectively split simulations into multiple simulation worlds, one for each group, so collisions between objects of distinct groups do not need to be considered. This is implemented by Unity (Unity 2, 2022), Unreal Engine (Golding, 2014), and Bullet Physics (Coumans, 2012).

3.1.5 Collision Response

Once a collision has been detected, it must be handled appropriately. For rigid-body physics simulations, this usually means "bouncing" the objects involved. An important parameter to consider here is the elasticity of the objects, also known as the bounciness or the coefficient of restitution. This value, between 0 and 1 for realistic simulations, controls how much of the initial energy of the objects is preserved after the collision. For example, with perfect elasticity, which is one, a ball dropped on a flat surface should bounce forever, as no energy is lost after each bounce. In a world with no elasticity, the ball will not bounce at all, as all energy is lost after the first bounce.

During a collision between two objects, using the explicit Euler method, only their velocities should be affected. Their positions will indirectly be affected by their new velocities, and their accelerations remain the same as acceleration measures continuous forces (such as gravity) rather than the instantaneous ones of a collision. This means that the process of resolving a collision is calculating the resulting velocities of the objects involved. Assuming perfect elasticity:

Momentum is defined as the product of mass and velocity: $p = mv$

Using conservation of momentum (total momentum before the collision is equal to the total momentum after the collision) we get, for 2 object *a* and *b*: $m_a v_a + m_b v_b = m_a v'_a + m_b v'_b$ with v'_xbeing the velocity of an object *x* after collision.

Kinetic energy is defined as $KE = \frac{1}{2}mv^2$

Using conservation of kinetic energy we get: $\frac{1}{2}m_a {v_a}^2 + \frac{1}{2}m_b {v_b}^2 = \frac{1}{2}m_a {v'_a}^2 + \frac{1}{2}m_b {v'_b}^2$

Combining the two results, we can obtain the following:

$$v'_a = \frac{v_a(m_a - m_b) + 2m_b v_b}{m_a + m_b}$$

$$v'_b = \frac{v_b(m_b - m_a) + 2m_a v_a}{m_b + m_a}$$

This can be extrapolated to cases with non-perfect elasticity by multiplying the resulting values by the elasticity parameter.

3.1.6 Continuous Collision

So far, the movement and collisions that have been discussed have been discrete. This is a convenient way to simplify it as it works well with frame updates. This simplicity, however, comes at a cost to accuracy. For object movement this can be addressed by the Runge-Kutta method, but where this truly poses a problem for rigid-body simulations is during collision. Because collisions can only be detected at a specific point in time during the frame, any collisions that would happen between frames could be missed, this is called tunnelling (see figure 6)

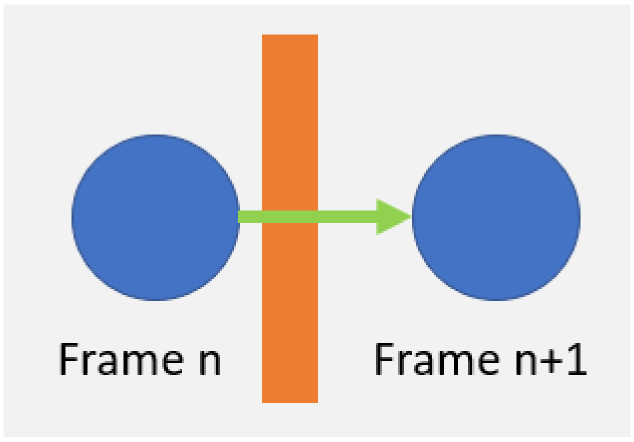


Figure 6: Example of Tunnelling

This can be prevented by either increasing the framerate, thereby increasing the time "resolution" of the simulation, making it more likely for the collision to be detected, or by enforcing a speed limit on physics objects, so that they are unlikely to pass through other objects between frames. These solutions only reduce the effects of this problem and can be impractical. A better system is required.

Continuous collision detection is the process of interpolating objects between positions and finding at what point a collision will first occur. This is practical when considering collisions between a moving object and a stationary one, as the moving object's position can be linearly interpolated along its velocity vector until a collision is found, at which point its position can be used for collision resolution. This process is no longer deterministic for collisions between moving objects. This is because, in a simulation, one object's movement must be processed before another's. This means the position at which a collision occurs depends on which object is moved first. Intuitively, both objects' velocities can be used to predict where a collision would occur, but this becomes impossible when considering the fact that another collision could occur before then, altering the position of any subsequent collision. Because of the butterfly effect, collisions between moving objects cannot be practically predicted, which is why continuous collision is most often used between dynamic and static objects, where the uncertainty is minimised (but still present).

3.2 Possible Optimisations

After initial research, the following novel optimisations were considered:

- Mutable physics object types: Physics objects can be "downgraded" from dynamic to static when no loss of accuracy would occur (when an object is assumed to no longer need to move), thereby increasing performance, as static objects have a significantly smaller performance cost, and possibly increasing accuracy, as it would allow for continuous collision to be used with newly static objects. This would require finding conditions governing when this can occur without loss of accuracy and ensuring that the cost of checking such conditions does not outweigh the performance and accuracy benefit of "downgrading" an object.
- Dynamic collision layers: Similarly to mutable physics object types, dynamic collision layers would involve changing object states at runtime. This would involve detecting which groups of objects will not be colliding and assigning them to groups accordingly. This would mean far fewer broad-phase checks and could result in a performance increase given that the layering process is fast and persistent.

Further research throughout the project presented some major flaws to these optimisations.

Firstly, both systems are already implemented to a certain extent at lower-levels of the physics engine. Although Mutable physics object types is not an existing technique in the way that is described previously, the concept of sleeping objects already fills that role. Instead of switching unmoving dynamic object to static, they are set to sleep, which yields the same performance increase, as they can be easily leveraged by the broad-phase for a significant performance increase (as discussed with SAP). There are specific situations where an object would not be set to sleep, but could be set to static using a system tailored to a specific simulation (see section 5.3.3), but these cases are highly specific, and are therefore not practical to be leveraged in generalised physics engines.

Furthermore, the work performed by the dynamic collision layers system should already be performed by a well-designed broad-phase, as assigning objects to different collision layers improves performance in the same way as discarding them efficiently by the broad-phase does. Having persistent collision groups that are only modified when objects from different layers become able to collide is the same principled as maintaining a persistent SAP AABB array, and therefore suffers from the same weaknesses (for example scenes where objects are moving fast and often). Similarly to mutable physics object types, there are only extremely specific cases where this system could be adapted to a specific simulation to provide a significant performance benefit and would therefore also not be practical for a more generalised physics engine.

Further research during development yielded more promising results. Because Unity does not provide a broad-phase that is highly efficient for scenes where most objects are moving (providing only SAP and variations of SAP for both 2D and 3D simulations), implementing one for this project would potentially yield positive results.

To get better performance with “noisy” simulations (i.e., ones with lots of movement), the broad-phase would have to not rely on persistent data structures, as the scene would evolve too fast for such systems to be beneficial. Therefore, a broad-phase with low overhead is required, but that still relatively accurately isolates objects in space. The Max Grid broad-phase was conceived during this project to fulfil these requirements.

The first step of this algorithm is to find the size S of the largest AABB of all physics objects in the scene. Every frame, the objects are mapped to a grid of cell size S, with objects only being added to the cell in which their AABBs centre is located. Lastly, for each object, a surrounding neighbourhood of size 3*3 is searched, any other objects found within this neighbourhood are considered to be possible colliders, at which point they are analysed by the narrow-phase (see figure 7).

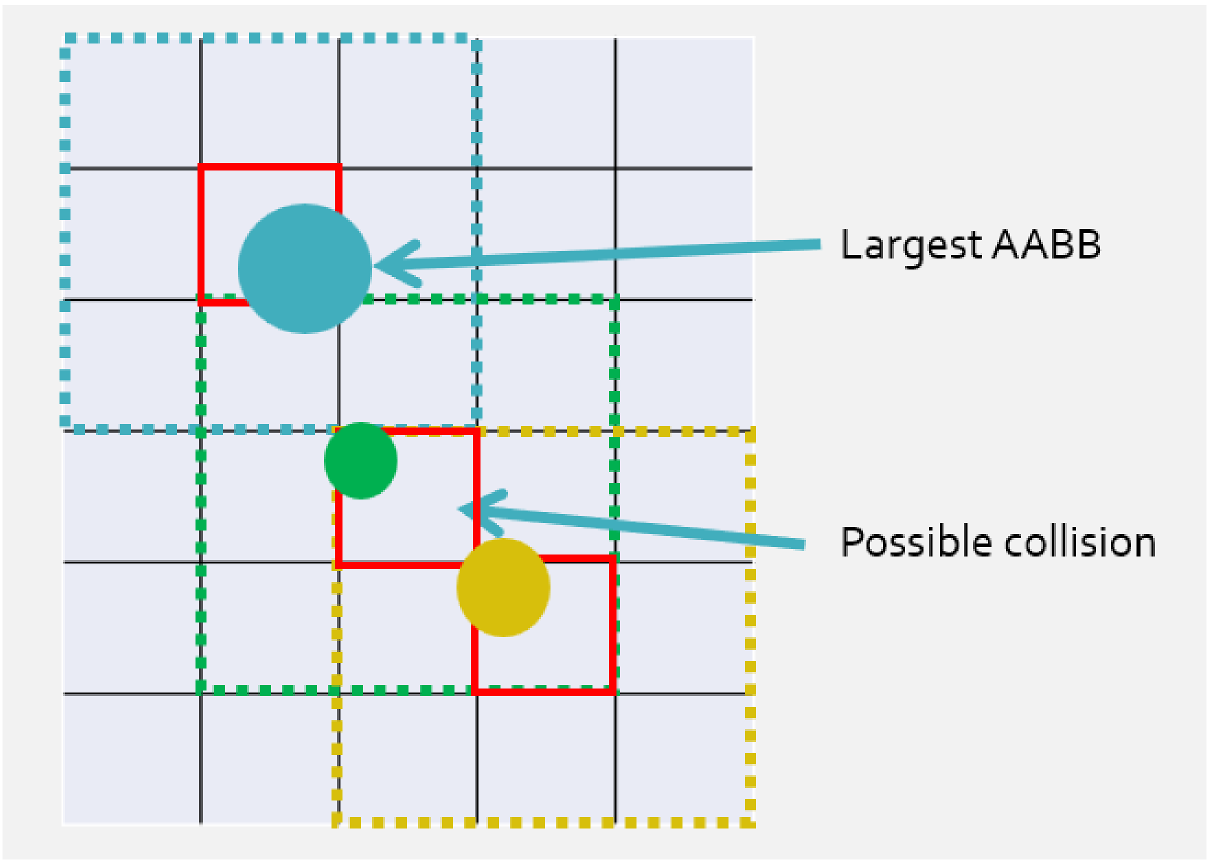


Figure 7: Max Grid Broad-phase showing cells occupied by objects in red and their neighbourhoods in dotted lines.

Because the cell size is set to the size of the largest AABB, this ensures that objects are colliding if and only if they are within one cell of each other. This provides a significant performance benefit, as it means a constant number of cells, 9, need to be checked for each object, whereas other grid based broad-phases would need to check an arbitrary number of cells. In theory, this algorithm performs well when object AABBs all have similar sizes, as it would minimise false positives. For example, if one object is significantly larger than all others, then the grid cell size would be much larger than most objects, meaning more objects can fit in each other's neighbourhoods without being in collision, producing more false positives. This condition is general enough for it to be acceptable for a fairly general physics engine. The time complexity of this algorithm is dominated by the process of searching each object's neighbourhood and mapping object to the grid, which can both be performed in linear time (given that the previous condition holds). Finding the size of the largest AABB can be performed in constant time when a new object is added assuming that the sizes of AABBs will not change. In the case that they can change (if polygons are permitted to rotate their AABB can change size), this process can also be performed in linear time. The average time complexity of this algorithm is therefore $O(n)$ in scenes with fairly uniform object sizes, and $O(n^2)$ in scenes object sizes differ by an order of magnitude, as the number of false positives cause it to become little more than a naïve broad-phase.

4 Development Plan

As previously discussed, the development cycles for this project would last a week, with work being set out at the beginning of each cycle using experience and research from previous cycles. For this reason, it is counterproductive to set out a detailed development plan or software specification. Instead, development plans will be discussed in the development timeline, in order to more accurately show how the development plan evolved with experience and research, allowing the project's goal to be fulfilled most effectively, even as the focus of development became more precise. Nonetheless, to ensure that the direction of development remained pointed at the project's goal, the general approach was set out as follows:

4.1 Working hand-built physics simulator

Firstly, a functioning physics simulator must be developed, able to simulate some rigid-body physics interactions to an accuracy comparable to a relevant game engine. This can be split into four sub-systems:

4.1.1 Rendering

The process of presenting the physics scene to a user should be the first system to be developed. If it works reliably, it significantly helps development of the other sub-systems, as it provides visual feedback that makes debugging simpler and makes new systems easier to think about. This process should ideally be GPU accelerated, especially for 3D systems, so that it does not affect performance in a significant way.

In case development of a rendering system is too slow, use of a library can be justified, as the rendering itself should not be the system that is the focus of testing for this project.

An alternative it to simplify the task by rendering scenes in 2D. This does not however prevent the simulations being 3-dimensional, as a 3D scene can be visualised to an extent in two dimensions.

4.1.2 Movement

This should be an implementation of the process described in section 3.1.3. This system can be simplified by removing consideration of external forces other than collisions (i.e., gravity) from consideration.

4.1.3 Collision Detection

This should be an implementation of processes described in section 3.1.4. Specific broad-phase and narrow-phase algorithms should be chosen during development and can be simplified for the purpose of accelerating implementation (a naïve broad-phase for example). If sufficient time is available, multiple broad-phases or narrow-phases can be implemented for the sake of comparing algorithms if this becomes the focus of optimisation.

4.1.4 Collision Response

This should be an implementation of processes described in section 3.1.5. It is worth noting that collision response of polygons that are permitted to rotate is considerably more complex and requires a strong understanding of many concepts relating to physics. If this proves too complex, objects can be given a fixed rotation. To do this without compromising accuracy, objects can be simplified to spheres/circles.

4.2 Optimisation

For the next step, if possible, optimisations that are not available in most game engines should be implemented in such a way that a comparatively better performance is achieved, or at least to highlight any potential benefits of building a custom system. Optimisation detailed in initial research (section 3.2) should be considered and refined if found to be relevant, as well as other techniques discovered during later research (such as the Max Grid broad-phase also discussed in section 3.2).

4.3 Testing & Analytics

Lastly, a similar simulation should be run on a relevant game engine as well as on the developed system, in order to compare performances and/or accuracy of both techniques. The simulation should be simple enough to avoid performance discrepancies that are irrelevant to the project, but complex enough that systems are used in ways that they would be in the real world (for instance a hyper-specific simulation is not a good indicator of how well simulations would perform generally). A testing environment for this process would be desirable, as well as ways of visualising analytics.

5 Development

5.1 C++ Development

The initial plan was to develop the physics simulator using C++. It is the industry standard for developing game engines and standalone performance intensive systems. This is because it offers high-performance due to a lack of garbage-collection, as well as memory control which can be useful for certain optimisations, but also it includes abstractions such as object-orientation, which is an incredibly powerful paradigm when it comes to game development, and more generally virtual environments (Gold, 2004). Other languages such as Rust offer similar benefits, but do not offer as many tools and examples/documentation as C++ does because of its widespread use.

To leverage the GPU, the OpenGL API was to be used to ensure such a task was manageable.

However, initial tests and attempts at rendering a simple shape in a window immediately proved more complex than previously anticipated. A lack of extensive experience with C++ combined with the steep learning curve of OpenGL would mean too much time spent learning, which would be too heavy a load considering physics simulation systems needed to be learned as well.

At this point, developing a working simulation needed to be a higher priority than developing the optimal one. Because of this, a switch to a language with which there was more experience was justified.

5.2 Switch to JavaScript

JavaScript was the language of choice with this new experience in mind. It enables significantly faster development because it handles some things automatically that C++ does not, such as memory and typing. There are of course drawbacks to this, for instance, the garbage collection used for automatic memory allocation and deallocation produces slower runtimes. However, it is unrealistic to strive for a system that runs as efficiently as Unity does at a low-level, as it is developed by a large team with extensive experience, which leverages specific architectures for optimal performance. For this reason, the focus of development would be towards optimising higher-level systems specifically involved in physics simulation. The fact that JavaScript uses dynamic typing, despite simplifying some aspects of development, does raise concern around safety. This causes significantly more unexpected runtime errors due to unanticipated typing. This was not considered relevant enough to justify the use of TypeScript (a strongly-typed superset of JavaScript) as the purpose of this project is not to produce a robust system but one that serves as a comparison to general purpose engines from a specific angle.

Other benefits with using JavaScript included significant pre-existing developer experience with it, as well as in-built tools for rendering. When run using the Chrome V8 engine (or similar implementations), JavaScript can render virtual scenes with robust performance using GPU acceleration using built-in libraries, such as Canvas (which is automatically GPU accelerated when required) or WebGL.

At this point, a decision had to be made about whether to build all systems from scratch, or to make use of external libraries, and build higher level optimisations on top of them. Both paths where subsequently explored to determine the best approach.

5.3 Simulation with Bullet

The first direction that was explored was the use of external libraries on top of which optimisations could be added. For this, bullet physics was chosen. It is the main open-source competitor to Nvidia PhysX and Vulkan (which are both proprietary), meaning the transparency of its documentation leads to more knowledge to be gained on key systems. To use this library with JavaScript, a pure JavaScript version of it was used called Cannon. This library was used in the hopes that if the high-level optimisations discussed in section 3.2 (mutable physics object types and dynamic collision layers) were not sufficient for the goal of this project, the experience gained would still be useful when constructing a physics engine from scratch.

Libraries are managed using the node package manager, which allows the use of a Node.js (a JavaScript runtime built on top of Chrome V8) local server for testing.

5.3.1 Rendering with three.js

In order to focus on the physics systems themselves, Three.js (a JavaScript 3D rendering library that uses WebGL) was used to handle 3D rendering using WebGL.

A common method for working with virtual environments is using scenes, which are containers for virtual objects. The first step to being able to visualise the scene is to add a camera to it. Various camera types are available, but for 3D the most common and intuitive is the perspective camera, which presents a 3D scene as it would appear to the eye in the real world, as opposed to orthographic cameras, which do not visualise depth.

Lights should then be added to the scene so that objects can be visible to the camera, prompting the use of a hemisphere light (one that simulates ambient light from a sky) and a directional light (one that shines from a specific direction, casting shadows). As lighting can be a fairly performance heavy process, no more lighting needed to be added, as a hemisphere and directional light suffice to provide accurate visual feedback (shadows that are cast by the directional light are useful for distinguishing distances).

Having added these elements, the scene was ready to be populated by virtual objects. This requires a mesh (which carries information about the shape of an object) and a material (which describes the colour and texture of an object). A cube mesh was selected and given a Three.js standard material, which can be lit but does not use more performance heavy rendering techniques.

The final step of rendering was to set up the animation loop. This involves creating a function which renders the scene, and recursively calls itself after a certain period, ensuring the scene is continually rendered so that movement is properly visualised. Instead of using a constant delay between frames, which would result in blocking code execution between frames as well as causing unexpected behaviour when performance heavy tasks cause frames to be delayed, the requestAnimationFrame function is used. This function, which is in-built to JavaScript, invokes the callback function provided whenever a new frame is available to the window, in an asynchronous manner. This means that execution can continue as the system waits for the next frame, as well as ensuring that the simulation's framerate is tied to that of the monitor and is reduced in a controlled manner when the system struggles to maintain a high framerate.

5.3.2 Simulating physics

The physics simulation provided by Cannon is represented in a similar way to the rendering. Instead of a scene, a world object is used to contain other objects, and is used to configure various parameters of the physics environment, such as gravity. This world can be populated by bodies, which require a mass, position, and shape (in this case a cube shape was used to match the rendering). A plane body was also created as a floor for the cubes to land on, preventing them from falling endlessly.

Once these bodies were added to the physics world (the cubes were positioned in a stack to test collisions), they needed to be matched to the objects being rendered in the scene. In other words, the objects being visualised needed to follow the objects in the physics world. This meant ensuring the sizes of the objects matched (to prevent overlap) as well as their quantity. Next, in the animation loop, the position of every object in the scene was set to the position of its counterpart in the physics world.

In order to test the system so far, simulations were run with objects in varying configurations to ensure the behaviour was visually indistinguishable from real world physics interactions.

Lastly, a performance monitor was added using components from three.js that displayed a graph of framerate according to time, enabling work to begin on performance. The resulting implementation is shown in figure 8.

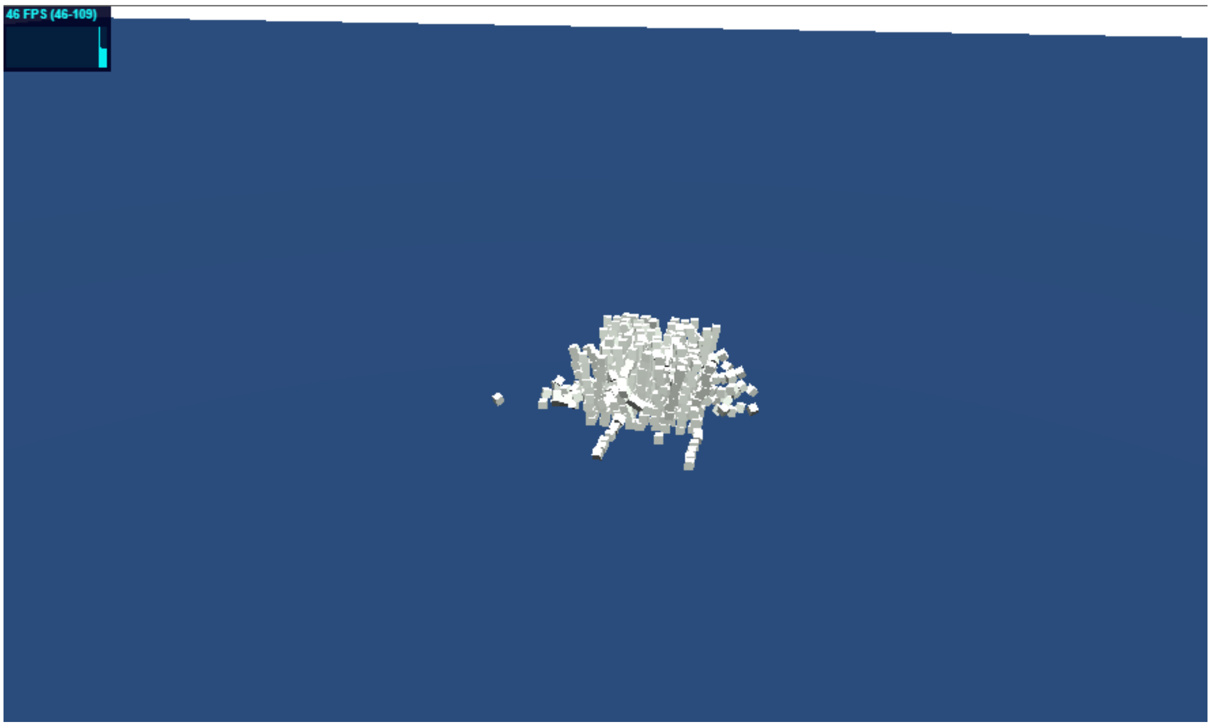


Figure 8: Simulation of a collapsing stack of cubes in three.js and Cannon with a performance monitor

5.3.3 Optimisation

The system was ready for optimisations. It was at this point of development that the concept of sleeping objects discussed in section 3.1.4 was fully understood and enabling it within the Cannon physics world immediately increased performance by a perceivable amount.

To compare this performance increase with that of mutable physics object types, a simple simulation was set up that would be able to highlight the benefits of this technique if they existed. This involved dropping layers of cubes on top of one another, setting objects in each layer to static once they landed on the previous layer (or on the floor for the first layer), as shown in figure 9.

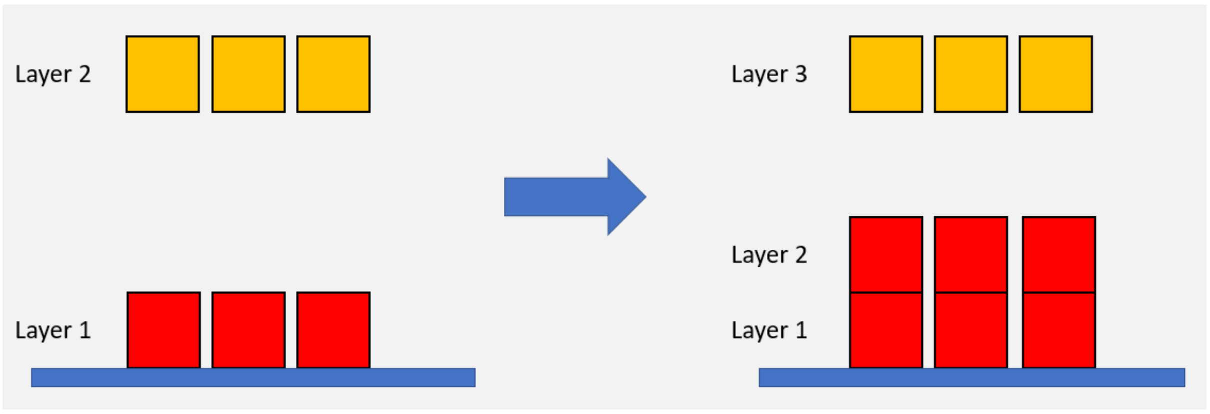


Figure 9: Diagram of simulation using mutable physics object types, static objects are in red and dynamic objects are in yellow

This simulation produced perceivable artifacts that were physically inaccurate, but these were temporarily set aside as performance was the initial focus. The simulation was run three separate times, once as a control experiment to provide some idea of the base performance without sleeping objects or mutable physics object types, once with sleeping objects, and once with the technique shown in figure 9 (i.e., mutable physics object types). Initial results suggested, as expected, that sleeping objects provided a better performance increase. This combined with the fact that sleeping objects is extremely versatile, while the other technique is highly specified as well as presenting inaccuracies, confirmed that these high-level optimisations that attempt to solve issues that are already solved to an extent are redundant.

Instead of building on top of already highly optimised systems, a better approach would be to build and optimise the systems themselves, meaning they could be better understood and utilised.

5.4 Switch to purely hand built

Using the experience gained in the previous sections, it was time to build an entirely custom physics engine from scratch. This would make it significantly easier to understand and implement meaningful optimisations and would more accurately fulfil the goal of the project, which is to compare a hand-built physics simulation to a game engine driven one. From this point, the plan set out in section 4 could be more accurately followed and would be implemented in pure JavaScript embedded into a static webpage.

5.5 Rendering

At this point it was clear that the focus needed to be more precise, with optimisations focused on key systems. To that end, rendering needed to be simplified as its optimisation is purely down to computer graphics and not physics simulation. The decision was therefore made to render a 2D scene. This can be done in two ways in pure JavaScript, using the canvas model, or directly using WebGL.

In theory, directly using WebGL could be faster as shaders could be optimised for this project, but this would again mean focusing on rendering optimisation, which was not relevant.

Instead, the in-built canvas 2D rendering system was used, which uses WebGL for leveraging the GPU given certain conditions are fulfilled. These conditions depend on the browser that runs the script, and are not easy to determine, but generally, as per the ECMAScript standard (i.e. the JavaScript standard that ensures interoperability between browsers) (ECMA International, 2021), GPU acceleration should be used when performance significantly benefits from it, which should be the case here.

To use it, the canvas object must be retrieved from the DOM (Document Object Model, and interface between the HTML webpage and JavaScript), which in turn is used to retrieve a “2d” context. This rendering context is used for any rendering tasks. To render circles, the arc function can be used, which draws a full circle if provided an angle of 2 pi.

Lastly, rendered objects, in this case circles, needed to persist between frames. For this, a Circle class was created, which holds data relevant to each circle, such as position, size, and colour (circles were assigned random colours upon construction in order to make them more distinguishable from each other). This class needed a draw function, which uses the arc function to draw a circle at the stored position. Lastly, if more than a few circles are to be simulated at a given time, a container is required. Instead of storing them in an array, an approach similar to the one three.js uses seemed more manageable, which is why a Scene class was created. This class contains an elements array to store the objects, which are added using an add method. On top of this, to easily draw all elements a Scene contains, an update function was added, which clears the canvas (so that circles are not constantly drawn on top of the previous frame's circles), and then calls each circle's draw function.

For the scene's update to be called every frame, an animation loop needs to be setup. A recursive call to requestAnimationFrame is the best way to achieve this, as discussed previously.

A renderer had now been constructed, capable of rendering circles of various sizes and colours, in any location on the canvas. The canvas is cleared, and the circles redrawn every frame to ensure that movement is correctly animated when implemented.

5.6 Movement

The next step of development was movement, which would require an implementation of the explicit Euler method. To achieve this, a 2D vector class was needed, so that position (which was currently stored in the Circle class as an array), velocity, and acceleration could be easily stored and manipulated. Knowing that many vector operations would be required, they were implemented as functions, including vector sum, subtraction, scaling, normalisation, and dot product, as well as methods to calculate vector magnitudes and distances. The last two functions were implemented in two ways, one that returned the exact vector magnitude or distance between two vectors, and one that returned the square of these values. This second approach is useful because it means avoiding the use of a square root, which is a costly operation. The fast versions of these functions would be used when possible.

A global variable was created that stored the time passed between the previous frame and the current frame, which is updated in the animation loop. This is used when calculating change in position and velocities as outlined in the explicit Euler method formulas as Δt. Using this, each circle's position and velocity could be updated in an update function according to the position and velocity formulas. The circle's acceleration field was set to a 2D vector of which the x component was 0 (no horizontal acceleration) and the y was set to a global variable storing the amount of gravity. This would enable gravity to be easily adjusted or be disabled (set to 0) altogether.

Each circle's update function now needed to be called within the scene's update function before the draw function. At this point, the initial movement system was complete, and could be tested by creating an arbitrary quantity of circles, each given random sizes, positions, and velocities, and adding each one to a scene object, which was then updated every frame in the animation loop. The circles were correctly animating and would move across the screen until they disappeared past the boundaries of the canvas. This would need to be fixed in the next section: collision.

5.7 Plane Collision

Before focusing on collisions between circles, the circles need to be contained in some way. For this, circle/plane collision is required. By arranging four planes around the edge of the canvas, and making the circles bounce off them, they would be suitably contained.

The first step is to detect when a circle is passing through a plane. Because the borders of the canvas are axis-aligned, this becomes very easy. In a general case, coordinates can be used to test collision with a plane, or with an area denoted by a plane, if they fulfil the plane's equation. In this case, that simply means testing whether the x coordinate is less than the canvas's minimum x or greater than the canvas's greatest x, similarly for the y coordinate. However, because each circle's position is its centre, this means that a collision with a plane would only be detected when its centre crosses the boundary. To detect when the first intersection occurs, the circle's radius can simply be added to or subtracted from whichever coordinate is being checked.

Now that collisions with the borders of the canvas could be detected, they needed to be resolved, which means computing post-collision velocities. Because the planes are axis-aligned, this process is straightforward, and involves reflecting the velocity coordinate along which the collision occurs (see figure 10).

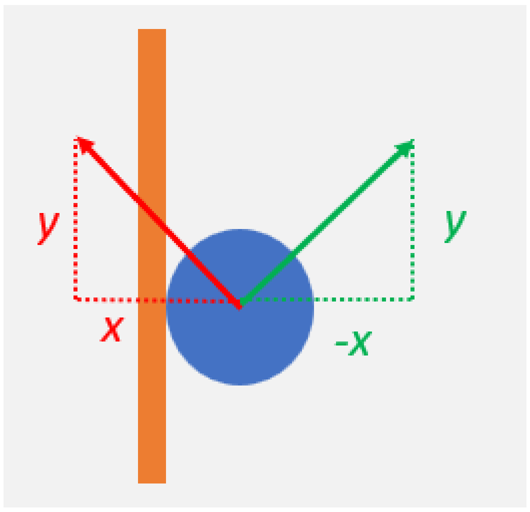


Figure 10: Diagram of initial velocity (in red) and resulting velocity (in green) of a circle and axis-aligned plane collision

Lastly, the resulting velocity should be scaled according to the elasticity. For this a global elasticity variable was defined, which was initially set to 1 (i.e., no energy is lost) for testing.

With this addition, the circles were correctly being contained by the planes, and were bouncing as expected. However, at high speeds, the circles would visibly leave the canvas before bouncing back. This is because of the tunnelling issue described in section 3.1.6. Although continuous collision would be impractical for circle/circle collision for reasons described during research, it could be simulated for circle/plane collision. The way this was achieved was by checking for collision before drawing the circles. During the collision check, the penetration depth of the circle in the plane was determined (by coordinate subtraction) and the circle was moved that distance away from the plane (see figure 11).

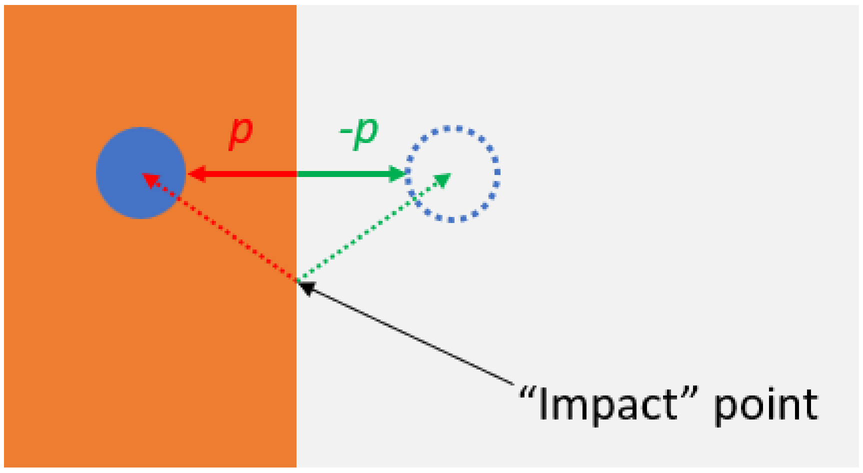


Figure 11: Circle being reflected from a plane by a penetration distance p, with the trajectories as dotted arrows

This would mean, after adding the velocity to the circle's position, that the circle would be drawn at the position that it would have reached if the collision had been detected when it first occurred, at the so-called "impact" point.

5.8 Testing Environment

Before implementing more complex systems, a testing tool was constructed, which would allow various simulation parameters to be easily configured. Such parameters would include gravity, elasticity, circle quantity, circle sizes, and circle initial velocities (useful in cases where gravity was set to 0). The parameters would be controlled using sliders in the simulation webpage itself for simplicity. The sliders could be easily created using HTML and CSS and were presented as shown in figure 12.

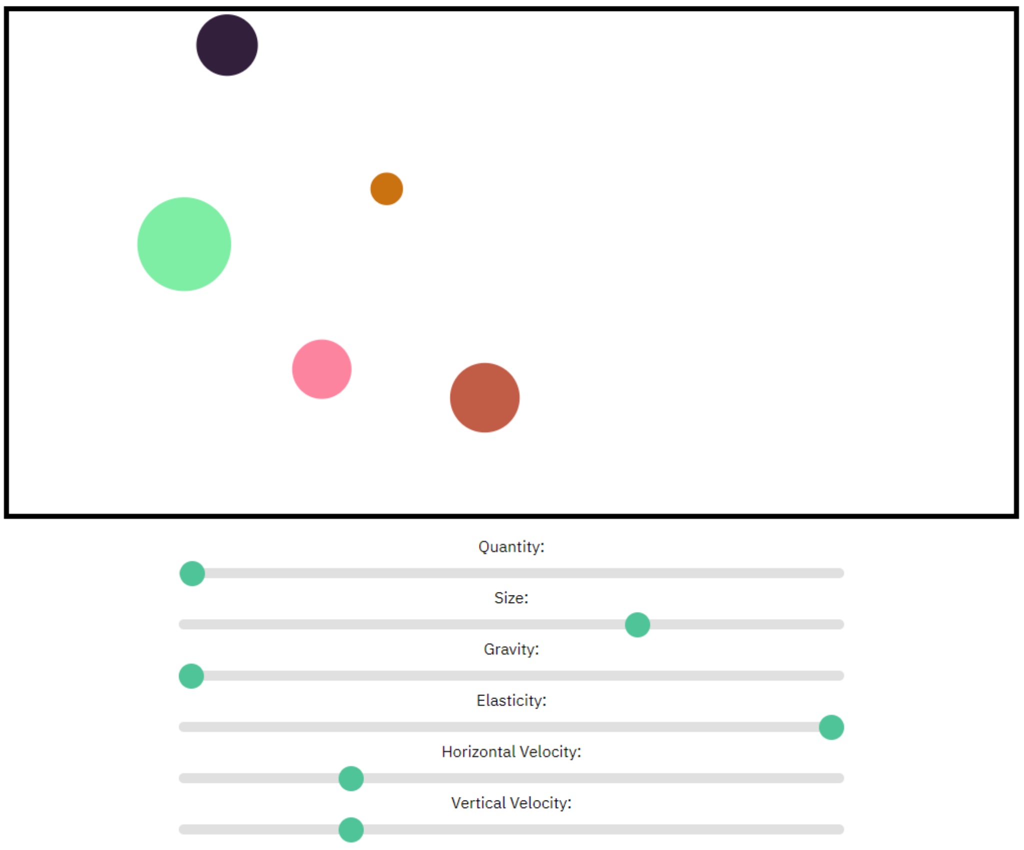


Figure 12: Simulation canvas with parameter control sliders

The value of these sliders could be accessed in JavaScript using the DOM, and a callback function could be invoked whenever a slider was updated. This function would ensure all the parameter values were updated when this happened, as well as calling a scene refresh function, which would clear the scene of all circles (using a clear function) and recreate circle objects based on the adjusted parameters. A certain amount of random noise was added to the size and velocity fields to ensure each circle was not identical, allowing for more edge cases to be covered by each simulation.

With this testing environment available, more complex systems could be easily debugged and tested.

5.9 Collision Detection

The first stage of implementing collision detection was to develop a simple but functional broad-phase and narrow-phase. To that end, a naïve broad-phase was initially implemented, and more complex ones would be added during the optimisation stage. Because a naïve broad-phase is the same as essentially having no broad-phase, this stage of development comes down to developing a narrow-phase. Thankfully, this is much simpler for circle/circle collisions than polygonal narrow-phases, as it just requires comparing the sum of radii of the circles to the distance between them.

Initially, two nested for loops were used such that each iterated over every circle in the scene, comparing every circle to every other circle. As collision response was not implemented yet, detected collisions were simply logged instead. The narrow-phase would output an array of circle pairs. Because of how the for loops were implemented, each collision pair was being added to this array twice, as each circle was being tested against each other circle, the pair would be added once for the first and once for the second. Initially, this would not be an issue, but with a functioning collision response system, this would resolve each collision twice, causing the circles to bounce back and forth every frame once they entered collision. This was initially solved by passing the collisions pairs array through a filter that removed duplicates. Later on, a better solution was implemented, that prevented duplicates from being added in the first place. This involved starting the second for loop at the circle that followed the one reached by the outer for loop, which would on average half the processing time for the second for loop as well as preventing duplicates. This still results in a quadratic average time complexity (because $O\left(\frac{n^2}{2}\right) = O(n^2)$), but the speed increase was still relevant. Based on visual feedback, this system was working, but to fully test it, a collision response system was required.

5.10 Collision Response

Responding to detected collisions is a straightforward implementation of the formula described in section 3.1.5. Because angular velocities of the circles are generally irrelevant, this means the highly complex task of calculating such velocities can be ignored.

A function for computing outgoing velocities was used for this, which utilised the vector operation functions defined previously. Before returning each velocity, it was scaled by the elasticity to simulate energy loss. This function was called in the scene update function, after the collision detection function was called. It is worth noting that for circle mass, the size, i.e., the radii, for each circle was used as per the formula for circle area ($A = \pi r^2$) which is directly proportional to mass according to the density volume formula (or area volume for 2D).

With this implementation, both collision detection and collision response could be tested using the testing environment. After a significant amount of debugging, they were working as expected. However, at high speeds, a significant amount of tunnelling was occurring, which was negatively affecting accuracy.

Although continuous collision could not be implemented the same way as for circle/plane collision, its results could be imitated with lower accuracy. This involved, before resolving collision, moving the circles away from each other so that they were exactly separated by the sum of their radii (i.e., they were tangent). This involved finding the vector from A to B for two colliding circles A and B, normalising it, and moving each circle along it (or inversely along it) by half the sum of their radii. A way the accuracy could be improved is by scaling the amount they each moved by their masses. This would involve, for example, if circle A had twice the mass of circle B, scaling that amount by 0.33 for circle A and by 0.66 for circle B, such that B moves twice as far, but the total distance moved by both is still exactly the sum of their radii.

5.11 Collision Optimisation

The obvious choice for optimisation is the broad-phase. In order to produce a performance that is comparable to that of a game engine, the broad-phase should be at least as efficient. To set a standard for comparison, the SAP broad-phase discussed in section 3.1.4 was first implemented.

5.11.1 SAP

Firstly, the AABB of the circles needed to be determined. For circles, this is trivial, as it would just mean adding or subtracting the radius from the x or y coordinate. Because only one axis needed to be swept, only the x coordinate mattered (the x axis was chosen because the canvas was wider than it was tall, meaning sweeping along the x axis would net fewer false positives). The start and end points of the AABB along the x axis were therefore the x coordinate minus the radius and the x coordinate plus the radius. To simplify handling these start and end points, an interval class was created, that stores these points, as well as the circle that they belong to and its index in the scene array. Knowing how to find the interval for each circle, the next step was to sort them (by the start point) and to find intersections in the sorted array. Because all circles in the scene would be moving, sleeping objects could not be leveraged for this, meaning that a persistent array was not useful. This means that there is little benefit in using insertion sort, so the inbuilt sorting function was used, and provided a function that could be used to compare Intervals. The resulting array could then be traversed, with pairs of intervals that intersected being added to a resulting array, which could then be passed to the narrow-phase which works as previously described.

To optimise this system, the sweep was performed on both axes independently, and an intersect function was used to find intersections that were in both resulting arrays. This function utilised a hash map, with the first array having its values added to the map. The second array could then be traversed, and if a pair were already in the map, it would be added to an output array, as it meant that the pair was present in both. In order to effectively hash each intersection pair, they were converted into strings, with the index of the circle that was largest first in the string, to ensure the pairs were always in the same order, so the hash would be the same for both pairs.

The performance increase of using multiple axis was not expected to be too significant, as it serves to reduce the number of false positives so that the narrow-phase would not be invoked as often, and the narrow-phase here is very fast. However, for an accurate comparison it was still worth getting the best possible performance out of this algorithm.

5.11.2 Max Grid

After implementing SAP, the search began for a novel algorithm that could outperform it, showing the benefit of using a hand-built physics simulator. It was at this point that the idea for the Max Grid (section 3.2) broad-phase was conceived.

To find the largest AABB, or in this case, the largest radii (which when doubled produces the largest AABB), a global cell size variable was created, and initialised as 0, Whenever the scene’s add function was called (i.e. when a circle was added to the scene), the size of the new circle was compared with the cell size, and if the new size was larger, cell size was set to that value.

A grid was created as a 2D array, the width of which was the width of the canvas divided by the cell size, and the height of which was the height of the canvas divided by the cell size. Because an integer value is required for array length, these values were rounded up. For debugging purposes, a draw grid function was created to visualise the grid, which could be called during the animation loop.

In a new broad-phase function, this grid would be cleared, and populated using the current circle positions. The initial plan was to check each circle's neighbourhood after this step, but that would result in each collision pair being added to the collision pair array twice. To solve this, each circle's neighbourhood was checked when it was added to the grid. This meant that each collision pair would be detected when the second circle of the pair was added, ensuring each collision was reported once, and slightly improving performance at the same time.

Once again, these pairs were then sent to the narrow-phase for accurate analysis.

5.12 Unity Simulation

Having completed a hand-built physics engine, it was time to build the same, or at least similar, physics simulation within Unity. The reason Unity was chosen as the game engine to compare against is because it fitted well with how the hand-built one had turned out. For instance, the fact that the focus was on collision detection rather than 3D rendering means that the other main contender, Unreal Engine, was not as relevant, because of its focus on real-time photoreal 3D rendering. Unity, on the other hand, is by far the most common engine used for 2D projects, as according to an online database of video games, 6.7 thousand catalogued 2D games are built using Unity, with the closest contender, GameMaker, having only 1.7 thousand (Doucet, 2022). Unity is also still relevant and has a highly active ecosystem, and for these reasons is by far the best candidate for comparison. Moreover, Unity projects can be built for WebGL using web-assembly, which is a closer comparison to JavaScript than most other build targets.

The implementation details of the Unity simulation will not be discussed extensively as this project serves to discuss performances, not the ease of use, nevertheless the general process was as follows.

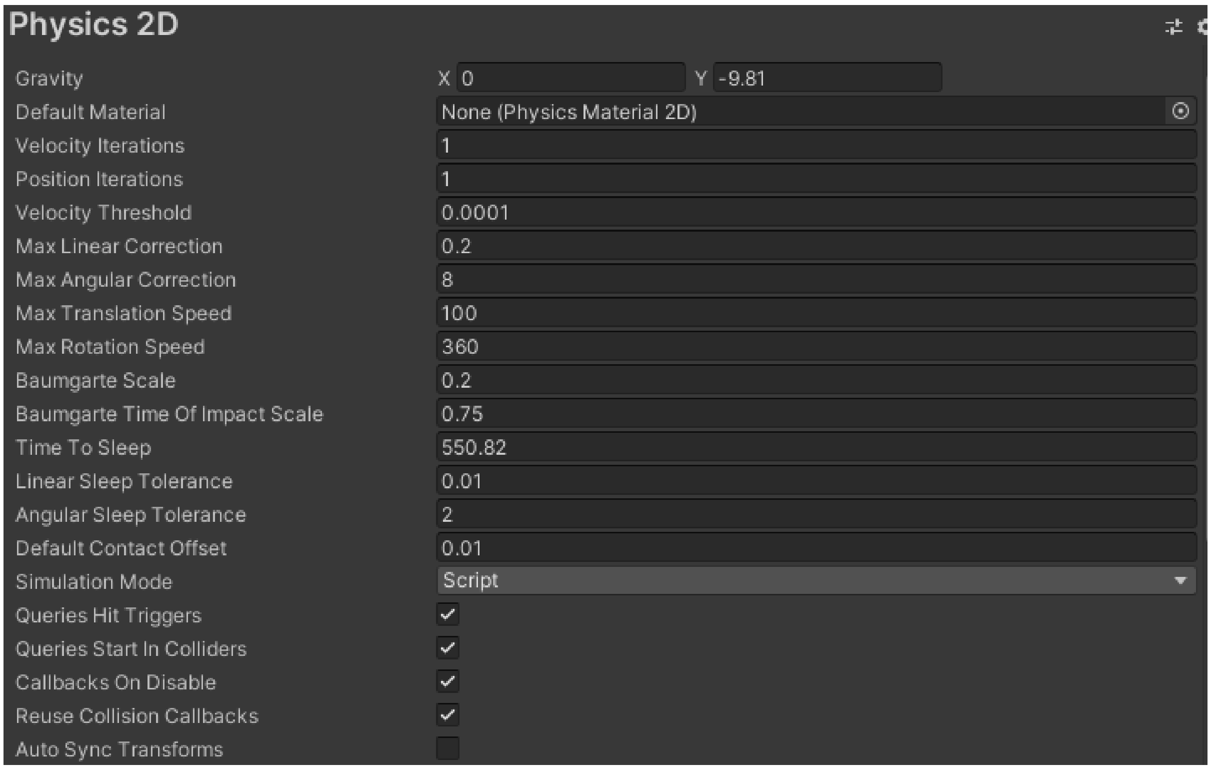
Firstly, after creating a blank 2D project, the required prefabs needed to be created. A prefab is essentially a blueprint for an instance of a class. In this case, the only necessary prefab would be a circle prefab. This would need several components: a rigidbody 2D component, which is the object that is used in the physics simulation, a collider, which defines what shape should be used for collision, in this case a circle collider was used, and lastly a sprite was required, which is the image that is drawn on screen to represent the object, in this case a circle.

A script was created and given an add Circles function. This function would instantiate a given number of circles using the circle prefab, each of which would be given a random (between set limits) initial force, scale, position, and colour.

To contain the circles, four walls were added to the scene, similarly to the planes in the hand-built version. Each was given a rigidbody 2D component, which was set to static, a box collider, and a sprite so that the borders are visible.

Lastly, the rigidbody component for the circle prefab was set to dynamic because the circles needed to be able to collide with each other, and was assigned a physics material 2D, which controls the elasticity (called bounciness) of an object as well as its friction (when relevant). The bounciness was set to 1 and the friction was set to 0 (as was assumed for the hand-built version).

The final step was to adjust the Unity project physics 2D settings so they would be similar to that of the hand-built version (see figure 13).


Physics 2D

Gravity	X 0 Y -9.81
Default Material	None (Physics Material 2D)
Velocity Iterations	1
Position Iterations	1
Velocity Threshold	0.0001
Max Linear Correction	0.2
Max Angular Correction	8
Max Translation Speed	100
Max Rotation Speed	360
Baumgarte Scale	0.2
Baumgarte Time Of Impact Scale	0.75
Time To Sleep	550.82
Linear Sleep Tolerance	0.01
Angular Sleep Tolerance	2
Default Contact Offset	0.01
Simulation Mode	Script
Queries Hit Triggers	✓
Queries Start In Colliders	✓
Callbacks On Disable	✓
Reuse Collision Callbacks	✓
Auto Sync Transforms	☐

Figure 13: Unity project physics 2D settings

The details for these settings can be found in the Unity Manual (Unity 2, 2020). The relevant fields that were altered were velocity iterations, position iterations, velocity threshold, time to sleep, linear sleep tolerance, and simulation mode. Velocity and position iterations control how many sub-steps, or iterations, are used for the Runge-Kutta method (see section 3.1.3). By setting these values to one, Unity would effectively be using explicit Euler method, and in this respect would be compared more fairly to the hand-built version. The velocity threshold is a value below which collisions are treated as inelastic (i.e., lose all energy). This helps when two objects should be in contact without movement (for example two boxes stacked on top of each-other) and prevents them from jittering from constant intersections. This feature would not be useful here as the performance tests would be run with perfect elasticity and therefore this situation should never occur, as it would mean an inaccurate simulation, which is why this value was set to its minimum amount. The time to sleep determines how long an object should be stationary before being set to sleep. Again, this should not occur, so was set to the maximum amount. The linear sleep tolerance also determines how easily an object can sleep, and so was also set to the maximum amount. Finally, the simulation mode allows the user to select if the physics step should be performed automatically every frame or should be called explicitly in a user-defined script. For the purposes of performance analysis, this was set to script (as discussed in section 5.13.2).

The resulting simulations (from Unity and hand-built) were at this point almost identical and were ready to be compared.

5.13 Analytics

Firstly, accuracy would be measured as able. The results would indicate whether comparing performances would be fair (a very inaccurate simulation can outperform a highly accurate simulation without much relevance). After having established this, the performances of both techniques should be measured. For this purpose, the testing environment was given a button that would allow simulation data to be downloaded.

5.13.1 Accuracy

Accuracy is hard to measure for physics simulations, beyond how “realistic” they appear to an observer, which was perfect for both simulations. However, there are some tests that can be performed to gain an estimate of the accuracy involved. The simplest one is to set the elasticity to one, to “drop” a ball, and to check how well it maintains its bounce over time. Because the simulation is perfectly elastic, no energy should be gained or lost per bounce. To measure this accurately the energy of the circle needed to be measured over time. This can be done using the formula for total energy $E_T = E_P + E_K$ where E_P and E_K are respectively the potential and kinetic energies defined as $E_P = mgh$ (where m is the mass, g is the gravitational field, and h is the height) and $E_K = \frac{1}{2}mv^2$ (where m is the mass and v is the velocity). Because only the evolution of these measurements mattered not the exact values, constants such as gravitational field and mass were not relevant and could be set to any constant value.

To implement this, total energy can be calculated every frame using the given formula in the scene’s update function. For multiple bodies, the total energy of all bodies can be summed and averaged. Every frame this value can be saved to an array.

5.13.2 Performance

Performance in terms of efficiency of the systems used is more straightforward to measure than accuracy. As the focus of this project was on the collision detection, it is the performance of this that should be measured. However, this is not directly manageable in Unity. Instead, using the script simulation mode as discussed previously allows the time it takes to perform an entire physics step to be determined. This includes collision response, but should still be dominated by collision detection, at least in situations where many objects must be checked for collisions, which is why it is a valid metric. For this reason, the time to process the entire physics step was also to be measured in the hand-built version.

Both systems’ performances needed to be tested according to the number of entities simulated, which is why they were automated in such a way that each simulation would run for 10 seconds, the average physics step time was calculated over that period, then the number of entities was increased by ten.

Having added the systems for measuring these times, the development for the project was complete, and what remained was for the results to be analysed.

6 Results

6.1 Accuracy

For the hand-built simulation, the results are presented in figure 14.

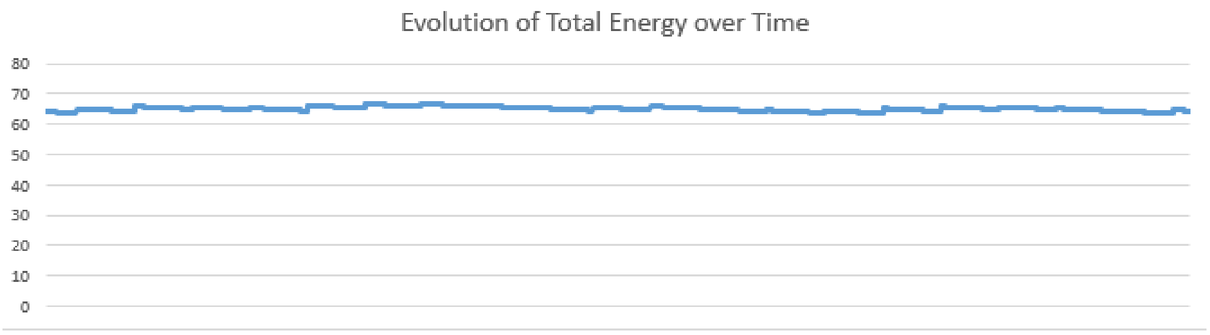


Figure 14: Total Energy of hand-built simulation of a single bouncing circle over 5 minutes

No unit is provided as the velocity and mass of the object in question is arbitrary (and cannot be unified across both systems), but if mass is in kg and velocity is in m/s, energy can be measured in Joules.

This shows, despite some variation, that the value remains approximately the same. Because velocities vary depending on the framerate (i.e., Δt in the explicit Euler method), this could have had an impact on the energies. For Unity however, the total energy evolved as presented in figure 15.

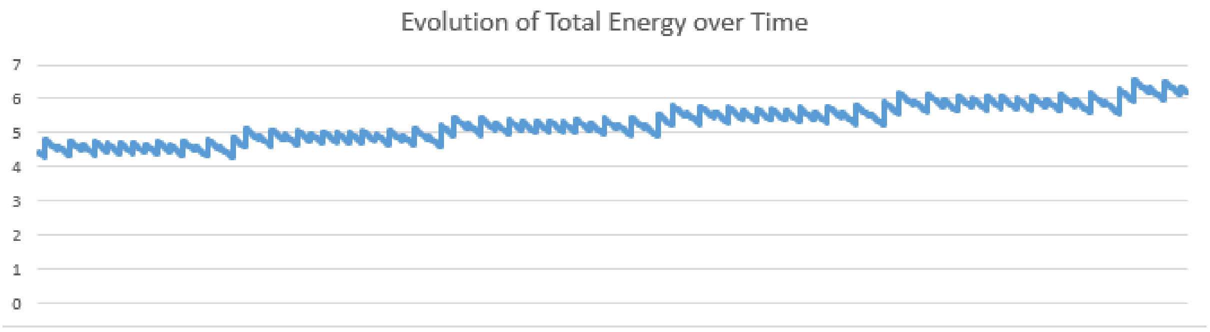


Figure 15: Total Energy of Unity simulation of a single bouncing circle over 5 minutes

This shows a distinct growth over time. It is fair to say that in this respect, the accuracy of the hand-built version is superior.

It is hard to determine other measurable metrics for accuracy. There is no way of comparing each simulation to a simulation with "perfect" accuracy as no such model exists, and if it did, a noisy simulation as the one used here would diverge almost instantly from the "perfect" simulation because of the butterfly effect. The measurements that were able to be made show at very least that the hand-built simulation is not sacrificing accuracy for speed any more than Unity is and establishes that a performance comparison between the two systems is fair.

6.2 Performance

Performance tests were performed in both systems. For the hand-built system, tests were run for each broad-phase type. Each test was run ten times. Any outlier tests were rerun to ensure external factors did not affect the measurements. Figure 16 shows the hand-built naïve physics step times of all ten runs according to number of circles, and figure 17 shows the average of these runs.

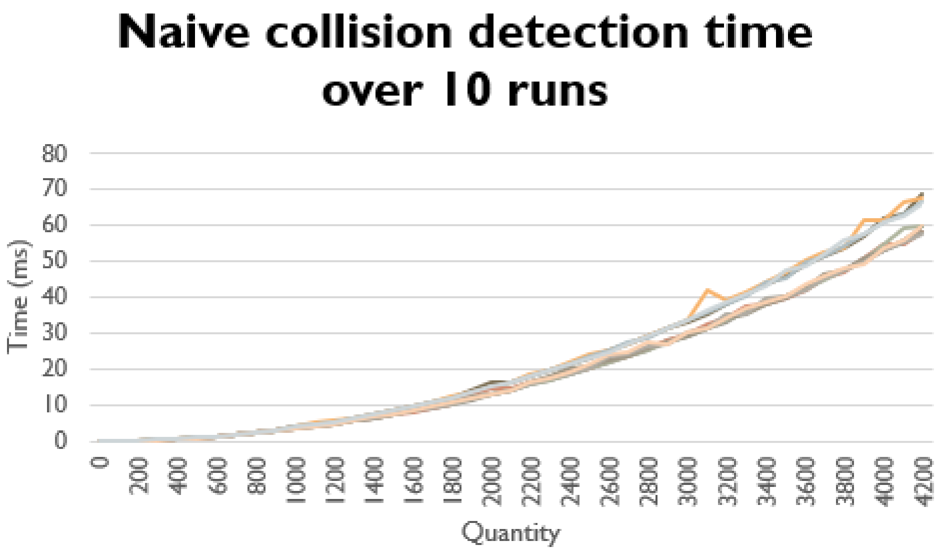


Figure 16: Physics step times using naïve collision detection over ten runs

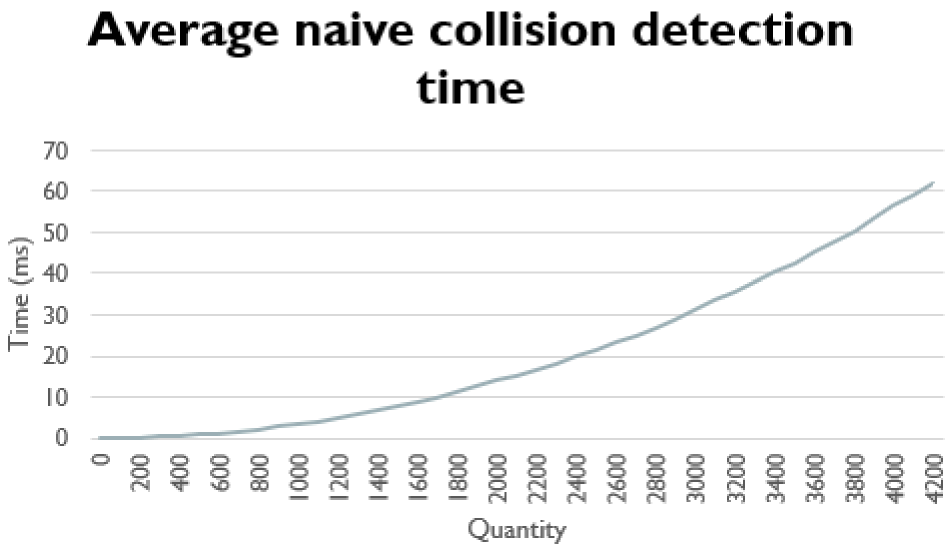


Figure 17: Average physics step times using naïve collision

These results show that when the quantity exceeds roughly 3000, the time it takes to process each step is more than 30 milliseconds, which means the framerate would be dropping below 30 frames per second, the approximate threshold below which an animation becomes visibly jittery to the human eye (Holcombe, 2009). On top of this, a lower framerate means less accuracy, as each circle travels more between frames, causing it to detect new collisions later, or even miss them altogether. For quantities above 4200, it was no longer practical to run further tests, which is why all future measurements will also remain under this quantity.

The results from testing the hand-built implementation of the SAP broad-phase are given in figures 18 and 19.

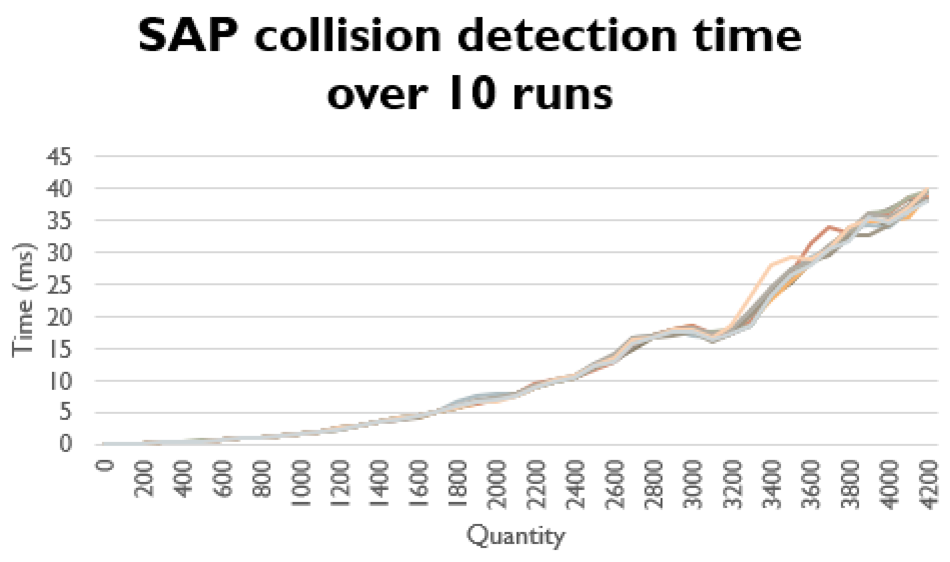


Figure 18: Physics step times using SAP collision detection over ten runs

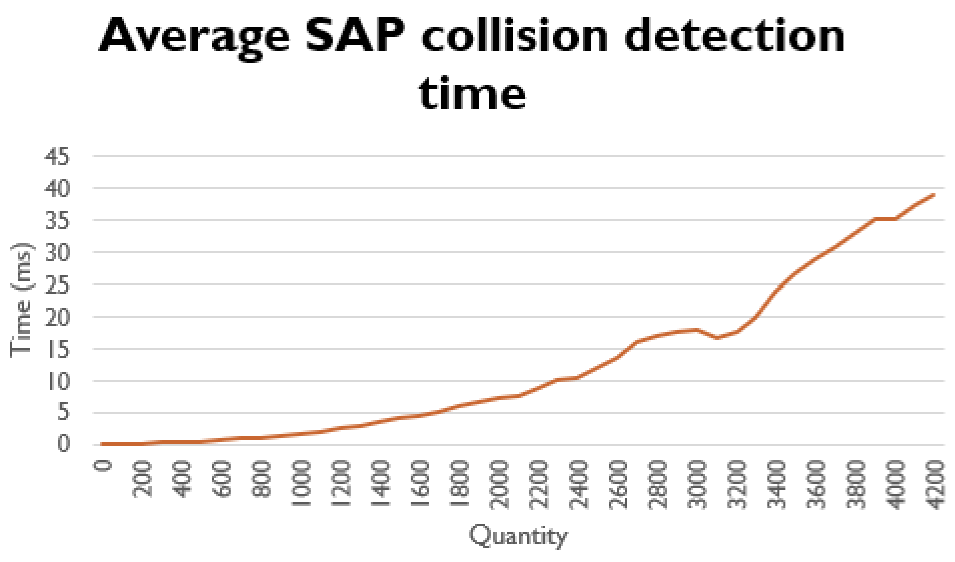


Figure 19: Average physics step times using SAP collision

The results for SAP collision detection are noticeably better, which maintains framerates over thirty for up to roughly 3700 entities. There is also a noticeable increase in performance as the quantity approaches three thousand. This simulation was tested in Google Chrome, Microsoft Edge, Mozilla Firefox, and Safari, and all but Safari presented this anomaly consistently. Moreover, the test was run backwards (i.e., starting from 4200 and decreasing the circle quantity) and the results were the same. It is worth noting that Google Chrome and Microsoft Edge are both based on the Chromium project, but Firefox is not, so it is hard to determine what exactly is causing the anomaly, and why it does not occur on

Safari. This is not hardware or OS dependent, because the same performance increase was noticeable when tested on a Linux machine, and on Mac. Nevertheless, because Safari is only available on Mac, it could not be tested on a different OS. The most probable cause is memory allocation, and that Chromium and Firefox's Quantum engine share a similar principle that Safari does not.

The results of the final hand-built system tests, using Max Grid broad-phase, the results are provided in figures 20 and 21.

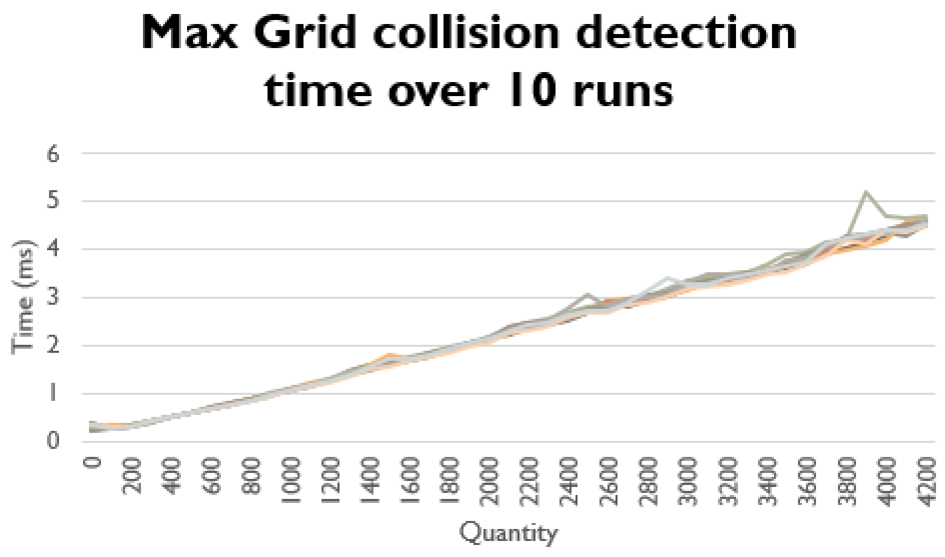


Figure 20: Physics step times using Max Grid collision detection over ten runs

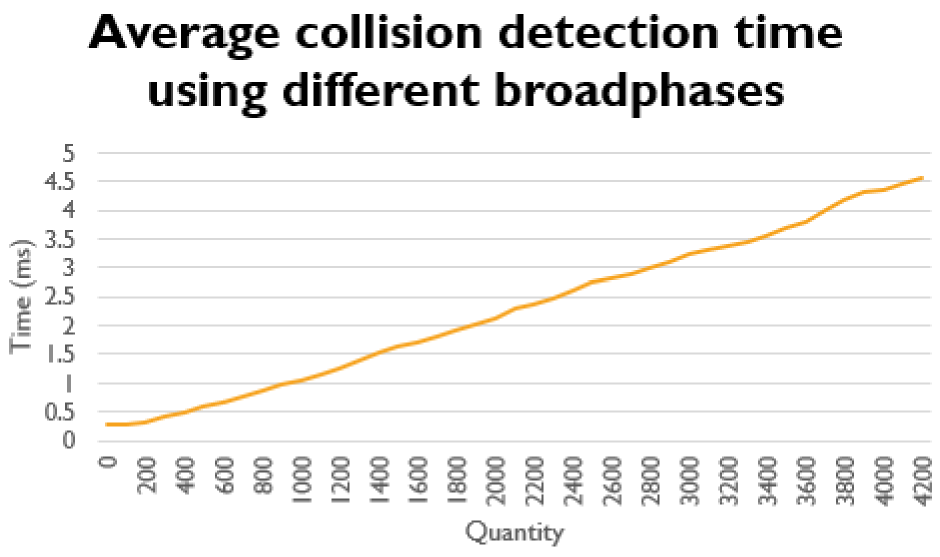


Figure 21: Average physics step times using Max Grid collision

The physics steps for this broad-phase are considerably faster. The time complexity for this system appears to be linear, which matches the analysis in section 3.2. Using this method, the quantity could be increased past ten thousand before the framerate got visibly slower, at which point physics step time was still below 30, indicating that the rendering, not the physics step, was affecting performance most significantly.

The final tests were performed in Unity, for which the results are given in figures 22 and 23.

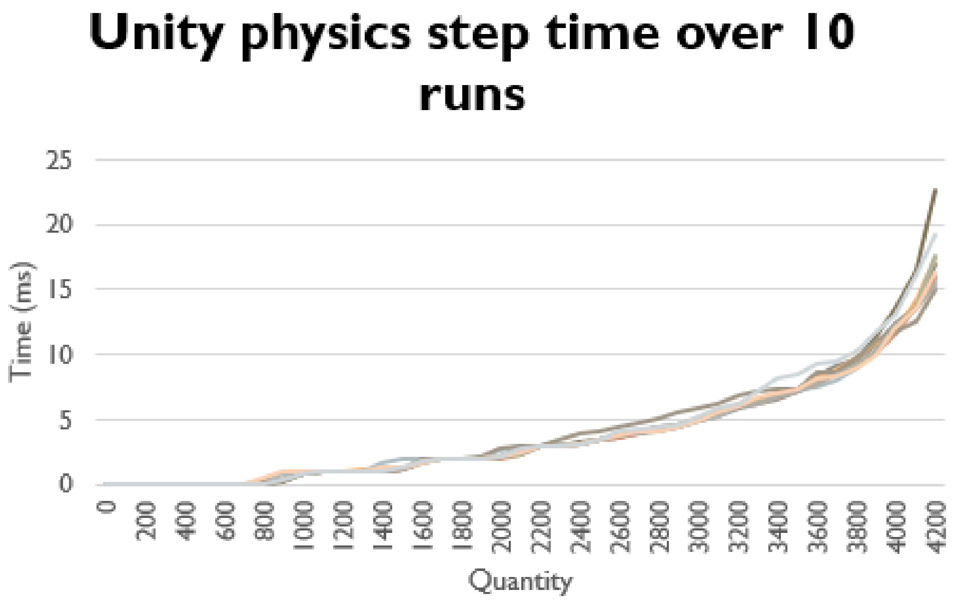


Figure 22: Physics step times in Unity over ten runs

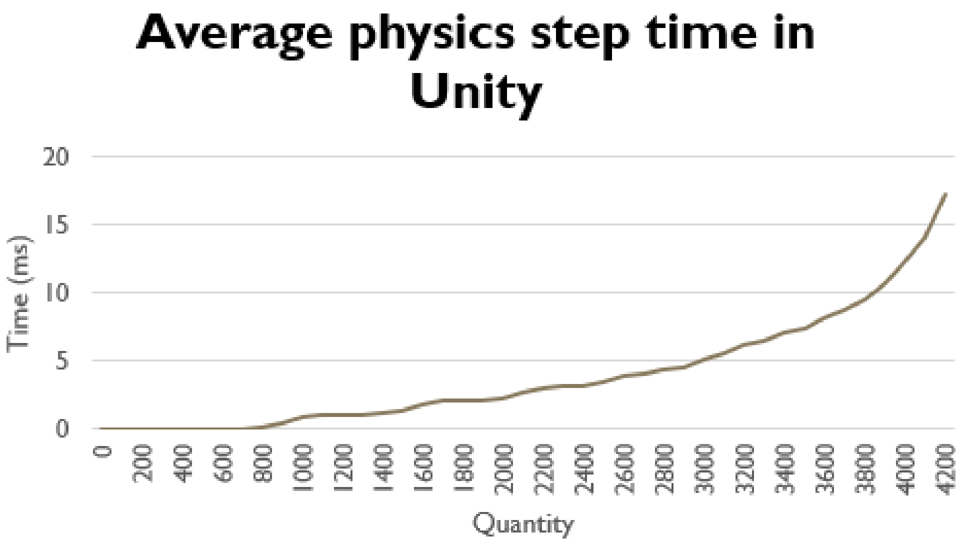


Figure 23: Average physics step times in Unity

This also allows for above 30 framerates for over 4200 circles. There is a considerable spike in physics step time for these larger values, but because the spread for the measurements (as seen in figure 22) is significant, it is hard to analyse this effectively.

Figure 24 shows all results grouped into a singular graph.

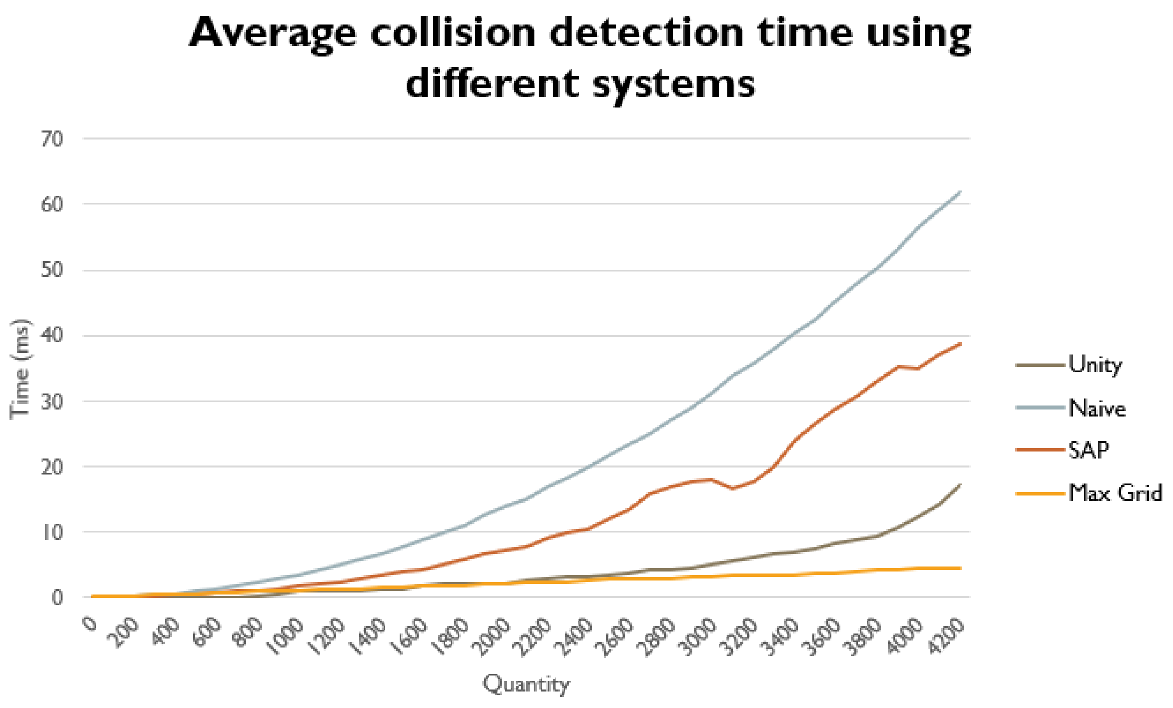


Figure 24: Average physics step times of different systems

This shows that the hand-built system using the Max Grid broad-phase was faster than Unity, highlighting the fact that the time complexity of Max Grid is significantly better than Unity's system for the given simulation. The performance of the Max Grid system exceeds that of Unity for quantities between 1500 and 1600. However, Unity's performance is never surpassed by the hand-built implementations of the naïve or SAP broad-phases.

7 Conclusions

7.1 Findings

Opting to simulate a scene without using a game engine can be a worthwhile choice if performance is a factor to be considered. Although, as previously discussed, the Max Grid broad-phase outperformed Unity because of the nature of the simulation, many constantly moving and bouncing circles, this does not make it a hyper-specific case. This broad-phase could for example be very useful when simulating certain particles. Often particle simulations consist of many entities that are roughly the same size, and that are contained, or at least limited by, a small quantity of static objects.

The superior performance is even more significant considering that Unity is highly optimised, and builds to web-assembly, which runs noticeably faster on all platforms (Herrera, Chen, Lavoie, & Hendren, 2018).

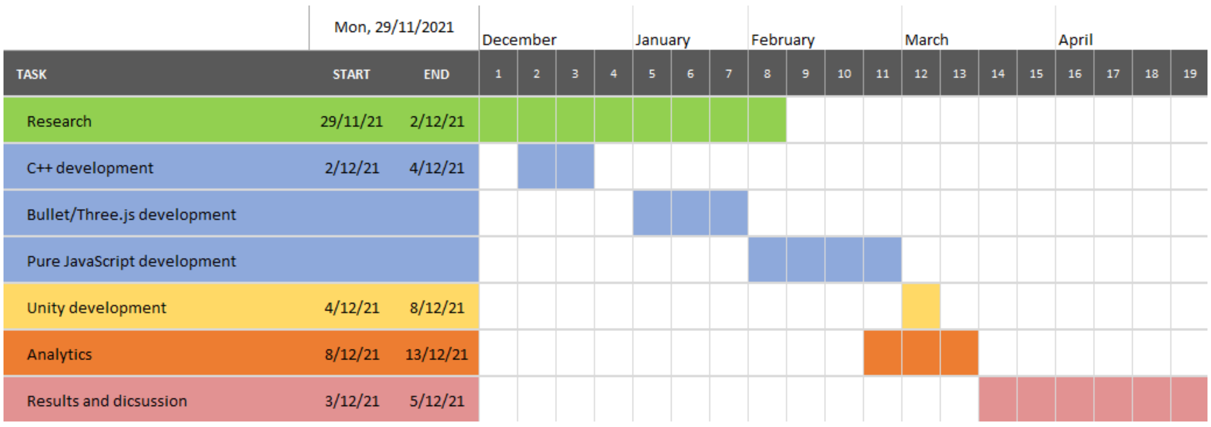
Finally, considerations should be made for the effort required for the development of these simulations. The private repository used for the hand-built pure JavaScript simulation saw its first commit on January 31st 2022, and saw its last commit before testing on March 13th 2022, which equates to 6 weeks of development (bearing in mind that this project was not an exclusive focus, and a more experienced developer would have taken significantly less time). The Unity simulation, on the other hand, was completed in a day. This highlights why game engines are still overwhelmingly popular.

However, the goal of this project was to find out if a hand-built physics simulation was still worth developing in terms of performance in a world dominated by general-purpose game engines. The project was a success in that respect, as it showed that there are situations where a hand-built simulation can provide the transparency and control required to achieve better performances than a game engine driven simulation and is an option that should be considered.

7.2 Project Management

7.2.1 Actual timeline

It is worth considering how well the project management helped, or hindered, the project. The actual development timeline was as shown in figure 25.



	Mon, 29/11/2021		December				January			February				March				April			
TASK	START	END	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Research	29/11/21	2/12/21	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩											
C++ development	2/12/21	4/12/21		🟦	🟦																
Bullet/Three.js development							🟦	🟦	🟦												
Pure JavaScript development										🟦	🟦	🟦	🟦								
Unity development	4/12/21	8/12/21												🟨							
Analytics	8/12/21	13/12/21											🟧	🟧	🟧						
Results and dicsussion	3/12/21	5/12/21														🟥	🟥	🟥	🟥	🟥	🟥

Figure 25: Actual timeline of different phases of development

This shows when the direction changes occurred. It is worth noting that the pure JavaScript development was significantly more work than the previous two phases and is not properly shown in this figure. This is because for the month of march this project became a much more significant focus of work than all other modules. Moreover, research was no longer being performed shortly after beginning this phase, as shown in the graph, meaning effort was concentrated on the development itself. The final timeline does show the final phase, results and discussion, as starting a week later than the initial timeline intended. However, this was an acceptable delay and did not prevent a full project to be completed in time.

7.2.2 Discussion

Firstly, the decision to not set out a detailed software design plan prior to development was hugely beneficial. In the best case, it would have been ignored when the development complications of section 5.1 where met, but in the worst case, it would have encouraged the development to follow a path that was not achievable, forcing a shift in direction later on with less time remaining, causing systems to not function at all, or simply leading to developer burnout. This taught the value of allowing implementation details to be more open-ended when experience is lacking, so that future experience and research can be trusted to guide development more accurately. This allowed the focus of the project to become increasingly precise, granting an accurate and deep understanding of certain processes, in this case collision detection, rather than a shallower understanding of more general systems, which was the case after initial research.

However, although the development with Bullet did grant valuable experience that would aid the development of the pure JavaScript simulation, it can be seen as a shortcoming of research, as the product of that phase was not directly useful to the goal of this project. A better trajectory would have been to use Unity or Bullet/three.js to simply experiment with creating physics simulations, using the documentation to gain a better understanding of what can be optimised and how. Instead, possible optimisations were conceived before adequate knowledge had been gained of existing systems, which led to a situation where the optimisations that were worked on turned out to already exist in a different form. This taught the value of understanding existing systems well before attempting to develop better ones and will encourage more research and experimentation to be conducted prior to committing to specific systems in future projects.

Despite this shortcoming, the project management philosophy used did serve as a solid backbone to development, and provided a clear path forward, while also being flexible enough to allow new experience and understanding to be incorporated into development and taught valuable lessons both with its successes and flaws.

7.3 Experience gained

This project has been a significant source of experience in multiple capacities. Firstly, it has been a source of learning for physics simulations, concepts for which can be applied to many different fields.

Secondly, it has provided significant JavaScript experience, which was initially thought of as a scripting language that sacrifices simplicity for speed, without much merit for serious projects. This has been proven wrong, and, although the algorithms themselves were the main cause of increased performance, the comparable, and even faster, speeds show that it is a robust language. The larger lesson learned here is that a programming language that is easier to use is not necessarily "worse" and should factor into the equation. On top of this, developer experience must also be considered when choosing a language, not just the language's objectives benefits and drawbacks, as was shown in section 5.1, when C++ was initially used, because its features seemed the most beneficial, without much consideration for existing developer experience. Had the switch to JavaScript not been made, the project would likely not have resulted in such success.

This leads to the final capacity in which experience was learned, project management. The benefits of a well-chosen management strategy were highlighted in the previous section. This shows how important such a strategy can be even for projects with a single developer, as it shapes the direction development goes in, making sure it remains on a stable path toward the project goal.

7.4 Future work

The obvious direction for future work would be to generalise the simulation, to show at what point the Max Grid broad-phase can be outperformed by Unity. The dimensionality could be increased to 3D, but this would not add much complexity, as the changes would mainly be to the rendering which was never the focus of this project. A better generalisation would be the introduction of general convex polygons. This would add a significant load to the narrow-phase, raising the cost of false-positives from the broad-phase, which could close the gap in performance between Unity and Max Grid. An analysis of the proportions of false positives of each technique could also be useful here. This could be simply achieved by comparing the length of the input and output arrays of the narrow-phase.

The introduction of polygons could also mean adding angular velocity to the system. This would mean a significant added complexity to the collision response system. This would shift the focus away from collision detection, and it would be interesting to see how collision response systems could be improved. However, the problem with this is the skillset required would shift away from that of a computer scientist and more toward that of a physicist.

Lastly, the possibilities considered so far have been constrained to rigidbody simulations. The systems used for this project could potentially be expanded toward softbody simulations, as they are also commonly used in game engines. This would involve developing a new movement system (as the explicit Euler method often fails for softbodies) such as Runge-Kutta.

8 References

Butcher, J. C. (1964). Implicit Runge-Kutta Processes. *Mathematics of Computation, 18*(85), 50-64.

Coumans, E. (2012). *Bullet Physics Manual.* Retrieved 04 18, 2022, from Bullet Physics: http://www.cs.kent.edu/~ruttan/GameEngines/lectures/Bullet_User_Manual

Doucet, L. (2022, 04 07). *Top Games*. Retrieved from GameDataCrunch: https://www.gamedatacrunch.com/

ECMA International. (2021, June). *ECMA-262, 12th edition, Language Specification*. Retrieved from ecma-international: https://262.ecma-international.org/12.0/

Fowler, M., & Highsmith, J. (2001). The agile manifesto. *Software development*, 28-35.

Gold, J. (2004). Software engineering for games. In J. Gold, *Object-oriented game development* (pp. 23-67). Pearson Education.

Golding, J. (2014, April 14). *Unreal Engine blog: Collision Filtering*. Retrieved 04 18, 2022, from Unreal Engine: https://www.unrealengine.com/en-US/blog/collision-filtering

Herrera, D., Chen, H., Lavoie, E., & Hendren, L. (2018). WebAssembly and JavaScript Challenge: Numerical program performance using modern browser technologies and devices. *University of McGill, Montreal: QC, Technical report SABLE-TR-2018-2*.

Holcombe, A. O. (2009). Seeing slow and seeing fast: two limits on perception. *Trends in cognitive sciences, 13*(5), 216--221.

Kockara, S., Halic, T., Iqbal, K., Bayrak, C., & Rowe, R. (2007). Collision detection: A survey. *2007 IEEE International Conference on Systems, Man and Cybernetics*, 4046-4051. doi:10.1109/ICSMC.2007.4414258

LaValle, S. M. (2012, 04 20). *Planning Algorithms*. Retrieved 04 06, 2022, from Cambridge University Press: http://planning.cs.uiuc.edu/node214.html

Newth, J. (2013). *Minkowski Portal Refinement and Speculative Contacts in Box2D.* San Jose State University.

Tracy, D. J., Buss, S. R., & Woods, B. M. (2009). Efficient Large-Scale Sweep and Prune Methods with AABB Insertion and Removal. *2009 IEEE Virtual Reality Conference*, pp. 191-198. doi:10.1109/VR.2009.4811022

Unity 1. (2020, June 05). *Unity - Manual: Physics Manager*. Retrieved April 05, 2022, from Unity Documentation: https://docs.unity3d.com/2017.4/Documentation/Manual/class-PhysicsManager.html

Unity 1. (2022, 04 09). *Unity - Manual: Colliders*. Retrieved 04 17, 2022, from Unity Documentation: https://docs.unity3d.com/Manual/CollidersOverview.html

Unity 2. (2020, June 05). *Unity - Manual: Physics 2D Settings*. Retrieved April 05, 2022, from Unity Documentation: https://docs.unity3d.com/2017.4/Documentation/Manual/class-Physics2DManager.html

Unity 2. (2022, 04 09). *Unity - Manual: Layer-based collision detection*. Retrieved 04 18, 2022, from Unity Documentation: https://docs.unity3d.com/Manual/LayerBasedCollision.html

Unity 3. (2022, 04 09). *Unity - Manual: Rigidbody*. Retrieved 04 11, 2022, from Unity Documentation: https://docs.unity3d.com/Manual/class-Rigidbody.html