



UNIVERSITAT POLITÈCNICA DE CATALUNYA  
BARCELONATECH

Centre de la Imatge i la Tecnologia Multimèdia

# RedEye 3D Particle Physics

Final Degree Project

Videogame Design & Development Degree

Author: Sardón Roldán, Rubén Gonzalo

Academic Plan: 2014

Director: Löpfe, Lasse

## Index

Synopsis .....	4
Key words .....	4
Relevant Links .....	4
Table Index .....	5
Figure Index .....	6
Glossary .....	8
1. Introduction .....	12
1.1 Motivation.....	12
1.2 Problem Formulation.....	12
1.3 General Goals.....	13
1.4 Specific Goals .....	14
1.5 Project Scope .....	15
1.6 Target .....	15
2. State of the Art .....	16
2.1 Physics.....	16
2.2 Particles.....	19
2.3 Engines .....	20
2.4 Conclusions .....	22
3. Project Management .....	23
3.1 GANTT .....	24
3.2. SWOT.....	25
3.3 Risks and Contingency Plans .....	26
3.4 Initial Cost Analysis .....	27
4. Methodology .....	29
4.1 Tracing Procedures and Tools.....	29
4.1.1 HacknPlan.....	30
4.1.2 GitHub Repository and Versioning.....	31
4.2 Validation Tools .....	31
5. Project Development.....	32
5.1 Integrating Physics .....	32

5.1.1 Engine Repository Setup .....	32
5.1.2 Adding Physics to the Engine's Modules.....	33
5.1.3 Particle Manager .....	36
5.1.4 Particles, Emitters & Components .....	37
5.1.5 Progress Iteration .....	40
5.2 Collisions and Resolutions .....	42
5.2.1 Collisions.....	42
5.2.2 Impulse Resolution.....	45
5.2.3 Progress Iteration .....	47
5.3 Utility.....	51
5.3.1 Boundaries .....	51
5.3.2 Editor .....	55
5.3.3 Particle Emitter Workspace.....	58
5.3.4 Component Serialization .....	61
5.3.5 Progress Iteration .....	63
5.4 Optimizing .....	64
5.4.1 Testing Grounds .....	64
5.4.2 Broad phasing.....	69
5.4.3 Data-Oriented Design.....	71
5.4.4 Profiling Analysis .....	73
5.4.5 Progress Iteration .....	76
5.5 Publishing.....	78
5.5.1 Release .....	78
5.5.2 Wiki.....	81
5.5.3 Website .....	81
6. Conclusions .....	83
7. Future Projects .....	84
8. Bibliography.....	85
8.1 Citations .....	85
8.2 Complementary Research Sources .....	85

## Synopsis

From researching to designing, implementing, and publishing the results, this paper describes the process of adding a 3d particle physics pipeline into Redeye Engine: a C++ game engine software developed by Julià Mauri and me, who is author to the complementary project of developing its rendering pipeline.

## Key words

RedEye, C++, particles, physics, simulation, rendering, 3D, video, game, engine, open source.

## Relevant Links

- *Engine Website:* <https://www.redeye-engine.es/>
- *GitHub Repository:* <https://github.com/juliamauri/RedEye-Engine>

# Table Index

Table 1: Gantt Task Breakdown.....	24
Table 2: SWOT Diagram.....	25
Table 3: Project Risks & Solutions .....	27
Table 4: Initial costs analysis .....	27
Table 5: Software used.....	29
Table 6: First Particle Physics Test - Debug .....	68
Table 7: First Particle Physics Test - Release .....	69
Table 8: DOD Particle Physics Test - Release.....	73
Table 9: Comparative Particle Count profiling results .....	74

## Figure Index

Figure 1: Atari's Pong Screenshot.....	16
Figure 2: Unity's Particle System Emitter Component (left) .....	21
Figure 3: Unity's Emitter Playback Controls (right) .....	21
Figure 4: Gantt timeline.....	24
Figure 5: HacknPlan task examples .....	30
Figure 6: Hacknplan Board Screenshot .....	30
Figure 7: Release v4.0 Screenshot from Github's repository.....	33
Figure 8: Visual Studio's Class Diagram – particle system classes.....	33
Figure 9: Code Snippet – Physics Module Class .....	35
Figure 10: Code Snippet – Application's Main Loop's last calls .....	35
Figure 11: Code Snippet – Particle Manager Class .....	36
Figure 12: Code Snippet – Particle Struct.....	38
Figure 13: Code Snippet – Particle Emitter Struct.....	38
Figure 14: Code Snippet – Particle Update function .....	39
Figure 15: Engine's first particle setup screenshots .....	40
Figure 16: Initial Gantt v1 – Base layer systems.....	41
Figure 17: Adapted Gantt v2 – Base layer systems .....	41
Figure 18: Code Snippet – Particle Inter-Collisions .....	43
Figure 19: Code Snippet – Collision Resolution.....	46
Figure 20: Adapted Gantt v2 – Base layer systems .....	47
Figure 21: Final Gantt v3 – Base layer systems .....	47
Figure 22: Initial Gantt v1 – Profiling, Optimizing & Publishing .....	48
Figure 23: Adapted Gantt v2 – Utility, Optimizing & Publishing .....	49
Figure 24: Plane Boundary Collision Screenshot.....	52
Figure 25: Sphere Boundary Collision Screenshot .....	53
Figure 26: AABB Boundary Collision Screenshot.....	54
Figure 27: Particle Emitter Properties Screenshot .....	55
Figure 28: Circular Emitter Simulation Screenshot .....	57
Figure 29: Particle Emitter Physics Parameters Screenshot.....	57

Figure 30: Particle Emitter Workspace Screenshot - Spawning .....	58
Figure 31: Particle Emitter Workspace Screenshot - Status.....	59
Figure 32: Particle Emitter Workspace Screenshot - Physics .....	60
Figure 33: Code Snippet – RE_ParticleEmitter serializable variables.....	61
Figure 34: Code Snippet – RE_ParticleEmission Binary Serialization extract.....	62
Figure 35: Code Snippet – RE_ParticleEmission Json Deserialization extract.....	62
Figure 36: Adapted Gantt v3 – Utility, Optimizing & Publishing .....	63
Figure 37: Code Snippet – Profiling Operation.....	65
Figure 38: Code Snippet – Physics' Engine Par Update .....	66
Figure 39: Code Snippet – Physics' Fixed Update (left) .....	67
Figure 40: Code Snippet – Physics' Fixed Time Step Update (right).....	67
Figure 41: Simulation's AABB screenshot.....	70
Figure 42: Application Configuration Window Screenshot - Physics .....	71
Figure 43: Code Snippet – RE_Math's Minimum Integer function .....	72
Figure 44: Code Snippet – Particle Manager's Draw Simulation Function.....	72
Figure 45: Code Snippet – Particle spawning's memory allocation .....	75
Figure 46: Adapted Gantt v3 – Optimizing & Publishing .....	77
Figure 47: Release v5.0 from Github's repository .....	78
Figure 48: Release Assets .....	79
Figure 49: Release's RedEye.Engine.v5.0.Particle.Profiling.zip contents .....	79
Figure 50: Release's RedEye.Engine.v5.0.zip contents.....	80
Figure 51: GitHub Wiki Index list .....	81
Figure 52: Engine website's Home page .....	82

## Glossary

**CPU:** Central Processing Unit. Computer hardware component in charge of running the main thread and any other threads it may support based on the number of cores.

**Thread** (of execution): smallest sequence of code instructions. CPUs have a single main thread and some other cores that can run their own thread of execution in parallel to the main thread; this process is called multi-threading.

**RAM:** Random Access Memory. Computer hardware component that stores data for quick access to the CPU compared to a Hard Disk Drive.

**GPU:** Graphics Processing Unit. Computer hardware component specialized in rendering procedures due to its high amount of processing cores and therefore huge amount of parallel processing threads. These units also contain a separate memory allocation unit to make up for long time intervals it takes to accessing the RAM from the GPU.

**Rendering:** the process of producing a 2D image based on a given input.

**User Interface:** application space dedicated for user interaction.

**Real time:** software that processes input fast enough to answer with feedback instantly. Many videogame assets for example react to the player's actions while some others are pre-processed to free the game from the processing load; these assets are thereby referred to as "**baked**".

**Particle:** A minute portion of matter. In a 3d environment, we refer to a particle as just a point with coordinates in space (x, y, z).

**Physics engine:** software aiming to approximate given simulations involving collision detection and may support other complex physical systems such as fluids and rigid or soft body dynamics.

**Game engine:** software framework aimed at producing videogames by supporting a variety of different systems.

**Game object:** Entity used in game engines to organize scene layouts in a hierarchy of elements. These game objects can store **components** such as transform, geometry, animations, etc. This system is referred to as **ECS** (Entity-Component System).

**Static:** label given to an entity that lacks changes over time. Segregating entities on behalf of the required processing cost allows for a faster iteration on a given group of entities. Entities who are prone to changes are therefore labeled **dynamic**.

**Id:** unique number that identifies an entity. In Redeye, ids are 64-bit random numbers with one in a billion chance to be repeated.

**C++:** object-oriented programming language derived from the original C language which on the other hand is data oriented.



**Cross-platform:** software developed to be able to run on different devices such as computers, phones, or consoles.

**Library:** code that can be used in your own coding solution offering ready-to-use functionality.

**OpenGL:** Open Graphics Library. Application programming interface (API) for rendering 2D and 3D vector graphics. Normally used to interact with the GPU for a cross-language and cross-platform hardware-accelerated rendering.

**Lag:** a computer's slower than expected feedback delay. The average console runs at 30 fps and monitors usually refresh at 60 Hz. Any frame that reduces the aimed framerate will cause a lagged response.

**VSYNC:** stands for Vertical Synchronization. A graphics technology used to align the monitor's framerate to the program's window framerate. If the application runs at 240 fps, but the monitor goes at 60Hz, it will result in 3 out of 4 frames rendered not being shown through the monitor. VSYNC solves this problem for high performance apps.

**Debugging:** searching and removing potential error from code. Code can be compiled in Debug or Release mode. Pausing debug execution will allow access to current processing line directives and local variable values. Release will not be able to display the thread data when pausing execution.

**Release:** compiling in release mode will build an executable file without debugging data and therefore run much faster. It refers to the publishing of ready to use files for a programming project.

**Profiling:** the act of measuring performance and timings of different procedures.

**Readme:** file included together with the release's executable file. It is a text file that contains all necessary information about the program: description, utility, patch updates, controls, user interface options, code authors, license, or any other important information code authors wish to share with a user before using the software.

**To-do:** pending task.

**Serialize:** references the act of transforming data into a structure able to be read and rebuilt. Basically, means saving data to later load or deserialize it.

**Parsing:** the analysis of a string of data. Describes the process of analyzing the information into different identifiable parts and the relationship between them.

**Meta file:** binary file with its containing information optimized to be read faster.

**Json:** JavaScript Object Notation defines an easy to read for humans' data format compared to the illegibility of a binary meta file; although it remains relatively simple for computers to generate and parse.

**Vector (Euclidean):** a geometric object containing magnitude and direction.

**Vector (container):** data structure that allocates its contents in a contiguous manner. Improves data iteration at the cost of having to reallocate its contents when adding past the amount of memory reserved.

**List:** data structure that allocates its contents in different places in memory. Improved data allocation at the cost of slower iteration cycles.

**Map:** stores data with a keyed value following a specific order. This orderliness improves on the lists' iteration costs. Allowing for multiple registers of the same key value changes the container to a multimap.

**Hash map:** data structure that allocates its contents in a contiguous manner like a vector and maps the index of the values to a map key.

**Visual Studio:** Microsoft's integrated development environment used to code many different types of software.

**HacknPlan:** agile methodology inspired project management tool aimed at game development.

**GitHub:** provides version control using Git for hosted software and other own features.

**Google Drive:** Google's storage and synchronization service which integrates several editing tools such as office automation.

**Tableau:** interactive data visualization application focused on business intelligence.

**Discord:** instant messaging and digital distribution platform. Allows voice calls, video calls, text messaging, media, and files in private chats or as part of communities called servers.

**Wwise:** cross-platform sound engine developed by Audiokinetic aimed at videogames and other interactive media. This software is free to non-commercial developers and under license for commercial use. Used by Redeye engine to power audio.

**Dear ImGui:** free user interface library for C++. Fast, portable, renderer agnostic and self-contained. Used by Redeye to power its UI.

**AABB:** Axis-Aligned Bounding Box. Geometrical rectangular shape characterized by just needing the maximum and minimum values of its contained space as its faces are parallel to each axis.

**DOD:** Data-oriented design. Programming paradigm based on improving data allocation for increased RAM to CPU communication performance.

**OOP:** Object-oriented programming. Programming paradigm based on creating abstract classes with own functions and virtual functions to improve code legibility and be easier to modify.

**Dot product:** (or scalar product) is the total sum of the products of each vector entry. Also referred to as projection product as the result is the projection of one vector over the other.

**Big O notation:** mathematical expression that describes a function's limiting behavior. Sorting algorithms have different number of steps, and we can use Big O to represent the change in cycles depending on the amount of  $n$  entities to sort.

# 1. Introduction

## 1.1 Motivation

Redeye is an open source 3d game engine built from scratch by two students: Julià Mauri and me. We have worked for quite some time now developing our own engine together as a personal project and we both thought this opportunity was a great pathway to add new features whilst challenging ourselves on a stricter, more formal, and academic manner.

Since the first engine's releases, we have been working on adding support for OpenGL shaders, adding deferred lighting to the rendering pipeline, applying broad phasing paradigms with Dynamic AABB Trees, and implementing a hash table workflow for the Entity-Component System. The engine has grown ever more complex and adding new features now requires compatibility checks throughout all other systems. Orderliness has now become a requisite to our workflow, and we must adapt a stricter methodology to ease feature production.

Starting this project, the current state of the engine is apparently stable as the main bugs have been solved but it is not bulletproof by any means. We had some ideas around what we could add such as raytracing technology which recently has had some attention drawn, but splitting the job was too much of a nightmare. We settled on particles and split the workload between its rendering process and the physics simulation. It suits both of us; for he is a graphic rendering enthusiast, had a major role in setting up shaders and even went through the trouble of parsing the shaders to be able to extract general uniforms whilst adding all the surrounding support for the editor. On the other hand, I am more prone to math and oversaw the matrix transformation system for the scenes' geometry and the camera. So now I oversee the part of procuring a 3d physics simulation pipeline. I had the opportunity of using similar software's before and am very much willing to figure out just how complex do they really get. It is going to help us both improve our skills, our engine, our workflow and hopefully help any enthusiast developer willing to squeeze their brains with our own approach to particle simulation.

## 1.2 Problem Formulation

The main problem forgoes that our engine lacks particles whatsoever. We basically want to add new features to our engine and nowadays, developers just browse for libraries that solve their needs. If someone has already solved a problem, why not use their answer directly? Why go through the trouble of making it our own way? Well, turns out

that the best solution for our problem is what best suits our users' needs. These libraries will come full to the brim, with tons of extra features aimed to scope the broadest number of possible users. But if your task involves a teamwork with a tight schedule, libraries will be most useful in freeing workload compared to having to develop your own solution and not having it available for the rest of the team until further into production.

There are many physics software's available online and each one offers its own approach to processing these simulations. The most polished systems available are not open at all. Mainstream game engines offer high customization for particle simulations but force you to use their framework to achieve the results in your project. High-end companies like Havok offer premium cross-platform solutions for videogame physics and other systems such as cloth and AI. So, when facing how to manage particle physics without spending a dime we must decide between open-sourced libraries or developing our own framework.

From previous experiences with libraries used in Redeye we are confident on our skill to procure particles running in a few days but learning and applying the library's full potential is going to take forever. Making our own approach will allow us a tighter grasp on its systems and their limits. Because we are the authors of our engine's code, we are in a unique position to mess with all systems in whatever way we deem necessary; it eases having to constantly lookup the engine's entrails too (but sometimes we still forget).

Settling on managing ourselves from scratch we know that the same problem has been solved several different ways. Thereby the task involves adapting our own solution for simulating particles based on the different number of approaches already openly available. Taking on the physics side of the main problem we can focus on collision detection algorithms, the different ways to approach them, their limitations and how to overall optimize the system.

## 1.3 General Goals

The main goal consists in simulating 3d particle physics using our Redeye engine which at the start of the project lacks any physics or particles whatsoever. The engine does however compromise an abundance of specialized systems for file management, the entity component system, interface editor, and hardware accelerated rendering. All of them interact with game assets in different ways. Adding new types of assets such as particles and their emitters will require adapting these systems to enclose the new content. The new pipeline must be compatible across the other engine's features thus being intrinsically part of each task.

I will focus on procuring all the calculations required to run a simple real-time simulation, continue building further functionality, improve their performance and finally publish the results. Rendering the simulation is not part of my goal; but, as I am working side by side with Julià, I can already foresee having tons of overlapping tasks worked into the same pipeline. This brings me to the next challenge of improving project management, teamwork, and research skills. I aim to upgrade the methodology used for a more efficient and clear production process. Together with improving already honed skills, I am keen on applying and testing acquired knowledge throughout the academic experience in engine development and its adjacent mechanisms.

## 1.4 Specific Goals

To achieve the set goal, I must create the physics module and the emitter component and get them up and running. I must make sure there are no hidden bugs to secure its effectiveness for reaching the goal consistently by adding error handling layers to the module and improving debugging and user experience. Once the system is secure, we step into profiling. Having accurate time measures for each process will allow us to visually analyze the performance of the different types of simulations and the effect of any optimizations we are able to output. Specific goals are thereby compromised as:

1. Adding physics module with particle emitter component support.
2. Adding customizable fields for emission properties.
3. Integrating simulations into Redeye's circumvolving procedures.
4. Testing, and debugging all new and previous systems.
5. Improving particle iteration cycles and measuring performance changes.
6. Publishing a release and its complementary documentation.

Although as a programmer I love optimizing, I must just contemplate the different ways of improving efficiency and leave them as secondary objectives for a chance to apply them only if we can spare time before proceeding to publish a complete release with all updated adjacent resources. As amateur programmers, we are expecting complications along the way and having a broad spectrum of different optimizations allows us to react to the engines development and choose the most suited ones accordingly based on remaining time versus implementation complexity.

## 1.5 Project Scope

We already have a website set up, but it has been long since it was last updated. The target outcome is a complete Redeye engine release on GitHub with its complementary published documentation. The solution provided must include a readme file and have the necessary tools to run a simulation and edit its values through the engine's user interface. There are many ways to create interface, but we must stick to just being practical and having something both of us developers feel comfortable managing.

Particle implementation can range from simple to overly complex systems. Because we are limited in time, it is key to assess our limits beforehand to cull any features that seem interesting yet irrelevant to our goals. Changing redeye internal systems is allowed but not encouraged. The amount of testing required to assure code safety all around the engine would delay us too much and having less code to review is less stressful to our tight schedule. Julià will have to change the rendering system inevitably, but I intent to not participate in any rendering system modifications. We also do not want the new pipeline to be prepared as a library. We want Redeye to be able to run its own particles.

## 1.6 Target

Our main target is then aimed at developers in need of similar software or students interested in learning about particles, physics, simulation, optimization, and engine development. Although we, as the engine's, developers are going to be using it mostly, other programmers could integrate our approach into their own projects as the engine is open source and available for everyone.

Having our code featured in another project will only benefit us with exposure (and maybe feedback) whilst improving or adding to their system. On an indirect level, if a game ends up published with our approach, the player will then benefit from our work. Although the primary targets revolve within the physics simulation sector, the project tries to reach readers with and without prior knowledge for a broader audience. Advanced concepts are always grueling to grasp and will be complimented with ancillary annexes.

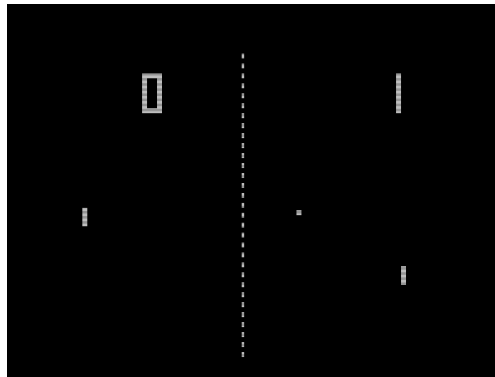
Because scheduling a programming project is based on our on arbitrary estimations of what we might be able to achieve, the scope is limited to the working released solution. Given the opportunity to improve out outcome, these are the brainstormed optimizations that could improve the system once the primary goals have been achieved: broad phasing, data-oriented design, multi-threading, and compute shaders.

## 2. State of the Art

The main goal is to build a 3d particle physics simulation into the existent Redeye's workflow. We can split its complexity by separating it into simpler subjects: physics, particles, and engines. The progress in these fields has exploded in the recent decades thanks to the rise in popularity of videogames: an industry bigger than movies and music together.

### 2.1 Physics

Physics refers to the way objects move and interact with each other. Physics in videogames spans rigid bodies, soft bodies, collision detection and fluid dynamics. The first hint of physics simulation in a videogame was none other than Atari's 1972 Pong: one of the earliest videogames. The paddles detected collision with the "ball" and changed its trajectory calculating the incidence angle. The ball's speed was not part of the collision output but instead increased constantly to create a tougher challenge over time. It lacked realistic physics for two reasons: gameplay and performance.



*Figure 1: Atari's Pong Screenshot*

So, are they still considered physics? Yes. Game physics do not have to follow realistic outputs. But what exactly conveys a physics system? Treb Connell breaks down game



physics into 4 key aspects during his “Intro to Game Physics” lecture<sup>1</sup>: instantiation, ticking, collision detection and collision resolution. To simulate a physical entity, we must first define this entity and then have its attributes updated at a high enough frequency to output a smooth result. Once entities change over time, they may collide between other entities and these intersections are to be identified to then calculate the consequence of this intersections. It feels quite straight forward but there are many ways to approach all the simulation’s stages.

Instantiating and ticking is quite essential: declare a structure and update it each frame. The easy way of setting up a simulation involves not caring at all about performance and just iterating all the data every frame: checking if each entity collides with any other present entities. On finding an intersection, we can choose either an impulse or constraint resolution. Impulse makes object reject each other as to avoid collision. The problem arrives when several entities are stuck above each other, and they try to resolve by continuously adding force to the entities until they free themselves. This occasionally ends up launching superposed entities with excessive force as the collisions stack each frame. Constraint resolution on the other hand tries to tackle this problem by locking these types of collisions and slowly resolving them as to not get the impulse method explosion. This then may cause objects that end up stuck inside the game’s floor to slowly get pushed upwards in a fixed position until the constraint can be removed, and it is safe to free the stuck entities.

Objects can normally get stuck together if there are too many in an extremely close space. They could even move fast enough to travel through a wall in the time it take collisions to be detected. Because we check collisions every frame, after entities move to their new calculated positions, they may have skipped a collision. An approach to this problem is splitting the interval into smaller intervals and iterating multiple times the simulation each frame. Checking segment collisions instead of point collisions also increases computing time but both solve the problem of fast objects going through stuff.

Another aspect that takes a toll in performance is that adding a colliding entity will force collision detection to run yet another cycle to check if this new entity is intersecting with anything else in the simulation. Therefore, the simulations will increase the required processing calculations exponentially for each entity simulated.

We should also pay attention to how space transformations are applied for physic bodies. The usual way to represent rotations is with the easy-to-use Euler angle system, where each axis has an assigned degree of rotation. Its great for visualizing and editing transforms. Inside the engine’s entity-component system, Euler causes calculations to tarry too much due to its hierarchical structure. Rotations expressed as quaternions

---

<sup>1</sup> Connell, T. (2021, March 10). *Intro to Game Physics*. Lecture presented at DigiPen Game Engine Architecture Club. <https://youtu.be/wPKzwSxyhTI>

alleviate this toll at the cost of unintelligible 2x2 matrix rotations. They also prevent a transformation anomaly called Gimbal lock where axis end up parallel losing a degree of freedom.

Even with quaternions, matrix space transformations will cause accuracy loss after consecutive iterations. As a last resort, we can even apply Runge Kutta<sup>2</sup> methods to increase accuracy. They compromise different variances of mathematical procedures used to approximate differential equations such as the ones our particles will follow. We cannot fully preview how much accuracy our particle system will lack. We believe they will prove unnecessary although some physics simulator utilize them, and we hereby hold judgement on whether we will need them.

Now that we have gone through all the extra calculations added to further control the system's imperfections, we must cover paradigms that may ease the required processing power. If the simulation starts to lag it will affect player experience and therefore, we cannot allow for the simulations to run wild and uncontrolled. Physics systems usually can cap the available time allowed to iterate each frame or directly just iterate once every 2 or a couple of frames. With games now running at 60fps on a standard computer, skipping frames in a simulation might not even produce visible changes or have object like before moving fast enough to go through other entities. When low on performance, a couple of slight miscalculations are preferable to freezing the players' screen.

A paradigm already applied in Redeye in its camera's frustum culling consists of iterating through a broader aggrupation of entries to quickly discard them faster from the main iteration. This is referred to as broad phasing and Redeye's Dynamic AABB Tree is used for this very same purpose as we can filter geometry outside the camera's field of view and only send to render visible geometry. It works best to have different trees for static, dynamic, and kinematic objects. User marked entities with the static tag imply they are not going to move and therefore will always be at the same location and there is no need to check if it has moved or collided with something new.

---

<sup>2</sup> Serrano, H. (2016, May 04). Visualizing the runge-kutta method. Retrieved March 10, 2021, from <https://www.haroldserrano.com/blog/visualizing-the-runge-kutta-method>

## 2.2 Particles

Particles refer to points in space. Computers can handle a lot of particles with a small amount of memory space for they constitute a minute unit of information. Particle visual effects use a dataset of simulated points over which to render sprites mainly, although they could as well output geometry. Sprite particles orient with the player's camera as to perceive the emission as a three-dimensional; this technique is called billboarding. Faking the 3d effect with particle orientation will offer improved performance compared to more complex geometry as only two triangles create the rendering surface.

The quickest solution in C++ for setting up a particle physics simulation is to create a base particle class and add all necessary methods inside the same class to calculate its collisions. It is easy to write ready to run in C++, but this quality of life comes at a great cost in memory. Not precisely cost in higher space but allocating object-oriented data will by default be spread throughout the RAM. If we then want to iterate through disperse memory fragments, the CPU's is going to process a particle extremely fast and have then to wait for the RAM to retrieve the next data chunk. The CPU enters this "wait state" that may not have much effect iterating on small amounts of data but having a huge pool of particles will benefit most by allocating data continuously.

Other situations that may stall the CPU are called branches. These branches occur when the compiler splits and organizes code in different memory location to optimize use of conditional contexts. If statements and calling a virtual function will most likely result in branching code slowing performance. Stoyan Nikolov<sup>3</sup>, creator of Hummingbird (the fastest HTML rendering engine) applied data-oriented design to C++ code to improve performance. Simple repeated sprite animations run x6.4 faster compared to Google's Chromium by applying data-oriented design in performance-critical areas removing virtual functions, allocating memory continuously and avoiding code branching.

Sean Middleditch<sup>4</sup> goes into detail in his Data-Oriented Design lecture explaining how branchings appear and how data-oriented design is a double edge sword too as it does improve performance at the cost of having less flexible code. Standard C++ code shines for being easy to modify and adapt while data-oriented classes to only hold data and have their functionality separated. Data-oriented C++ code eases multi-threading setup.

---

<sup>3</sup> Nikolov, S. (2021, March 10). *OOP Is Dead, Long Live Data-oriented Design*. Lecture presented at CppCon. <https://youtu.be/yy8jQgmhbAU>

<sup>4</sup> Middleditch, S. (2021, March 10). *Data-Oriented Design*. Lecture presented at DigiPen Game Engine Architecture Club. <https://youtu.be/16ZF9XqkfRY>

Contiguous iteration data is perfect for creating several threads and having each one tick a given range of indexes.

## 2.3 Engines

Real-time 3d particles offer a high amount of feedback improving player experience. That is why mainstream videogame engines include particle physics simulation editor built with as many features as they could fit. Compared to the scope of a pair of indie developers, they can produce massive tools to create graphically engaging professional results. Some engines are cross-platform and even allow compiling to different architectures. Each engine has its own licensing prices and offer a variety of different plans for the different types of developing entities: from big businesses to one-maned studios. Covering as many developers' needs as possible and reducing the amount of labor required into programming back-end systems, engines may convey an Asset Library or Store showcasing own and user-built available assets and add-ons. But when settling for an engine, we must consider that it also forces your product to use the engine's base.

Physics and particles' state of the art already offers numerous approaches to building our own 3d particle physics simulation. Let us take Unity for example. It offers a Particle System Emitter Component with different modules that modify the emission. Having selected a particle emitter in the current loaded scene opens playback controls to preview changes in real-time. It has a module for collisions, lights, and many over-lifetime changes. The particle count limit is undefined as it varies depending on the architecture it runs. We do however know that they cover a range from a thousand to a couple hundred thousand. This soft cap is due to them running through the CPU and can be surpassed to millions when using hardware acceleration and sending the calculations to run in the graphics card.

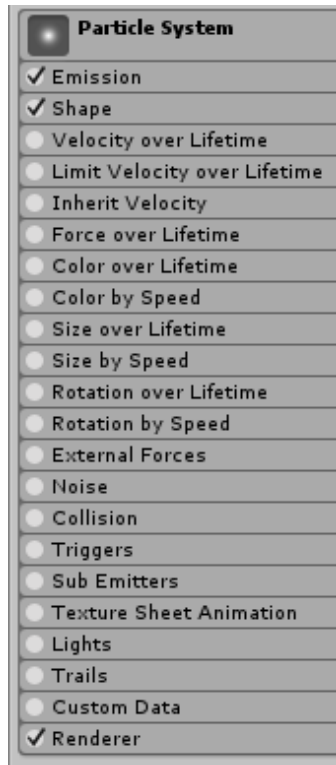


Figure 2: Unity's Particle System Emitter Component (left)

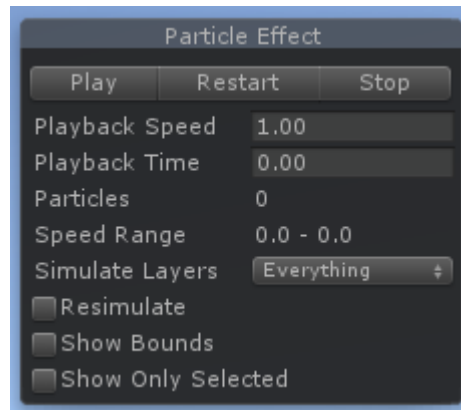


Figure 3: Unity's Emitter Playback Controls (right)

Unity also counts with a Visual Effect Graph Editor that allows VFX artist to create their nodal network for particles processed by the GPU using compute shaders versus the native CPU processed emitter component. Making the GPU process the simulation will greatly increase total available simulated particles using all the available threads. It frees the CPU's processing load but may cause threads to idle until the simulation's output is returned. RAM to CPU communications is much faster than RAM to GPU. Redeye's current rendering process is mainly occupied sending uniforms to the GPU.

Unity has been very vocal about the data oriented approach of their ECS. Mike Acton's "A Data Oriented Approach to Using Component Systems" GDC lecture<sup>5</sup> also stresses the importance of data-oriented design's increased performance. Redeye uses dynamic hash tables to store game objects and transforms. It does offer higher component iteration performance, but we stopped using memory pointers as the hash tables reorganize their content and memory locations for entities and component may vary from one frame to another.

---

<sup>5</sup> Acton, M. (2021, March 10). *A Data Oriented Approach to Using Component Systems*. Lecture presented at Unity at GDC, San Francisco. <https://youtu.be/p65Yt20pw0g>

## 2.4 Conclusions

Physics, particles, and engines' state of the art is quite extensive and thousands of different approaches available can solve and further improve the main goal. Optimizing seems to be a recurrent approach when dealing with data-intense procedures like particles, physics, and rendering systems. So clearly when dealing with all three at a time, we can expect lag as a foreseeable issue.

Physics' 4 stages are simple procedures that can be largely improved through the right paradigms due to the systems' exponentiality. Having playback controls and complementary editor support will be deemed essential to test and iterate the simulations. After having the system running, there are different options to advance judging on performance and efficacy. Accuracy deficiency and such other problems have a multitude of approaches. They are case-sensitive and therefore to be applied depending on how the system ends up being set.

Improved performance increases the number of simultaneous particles to be simulated in real-time. We are adding particle physics as a new feature to our engine and thus basing our approach on other engines' designs seems fit. Aiming for a result such as complete as Unity's optimized systems seems out of our reach. To emulate Unity's modular native component being just 2 developers we must narrow down our scope to an achievable goal with space for improvement. The paradigms available within our scope scarcely convey broad phasing, changing the C++ code to data-oriented design, multi-threading, and GPU compute shaders. Measuring the highest supported number of particles and comparing the performance profiles we can objectively measure the system's improvement before and after optimizing.

Setting up a physics module and its component editor values in C++ for a viable simulation can be achieved in the project's span as the problem of creating particle simulations has already been tackled by many developers. Yet each approach defers from one another as their complexity makes them arduous to categorize. We will inevitably build our own approach as it must comply with Redeye and base it on the other documented approaches.

## 3. Project Management

The next enumeration lists and swiftly describes all predicted tasks required together with our lesser priority optimization options.

### 1. Base layer systems.

#### 1.1. Physics Module

- 1.1.1. **Initialization/Clear:** create a RedEye module that can initialize a set of particles in different customizable ways.
- 1.1.2. **Update:** allow the coordinates to change over time kinematically.
- 1.1.3. **Detection:** iteratively find collisions of overlapping coordinates.
- 1.1.4. **Resolution:** apply impulse resolution to particle collisions.

#### 1.2. Emitter Component

- 1.2.1. **Playback buttons:** simulation controls.
- 1.2.2. **Editor values:** customizable input parameters.

### 2. Profile System

- 2.1. Setup function profiling and search for unexpected results.
- 2.2. Analyze performance values.

### 3. Optimize (if there is enough time).

- 3.1. **Broad phasing:** adding a layer of collision detection to reduce checking redundant collisions.
- 3.2. **DOD:** translate C++ structures into data-oriented design entities to ease branching and CPU wait state.
- 3.3. **Multi-threading:** add support for multiple processing threads for CPUs with several cores. Organizing the system into data-oriented design entities prepares the processing of a single chunk of data into threads.
- 3.4. **Compute shaders:** Julià, who will work on rendering may end up digging into compute shaders allowing the simulation to be calculated on the graphics card using its many more threads. No other optimization will allow for faster simulations, but it is by far the most complex addition possible.

### 4. Publish Release

- 4.1. Update read.me, wiki, and website.

## 3.1 GANTT

To organize our tasks, we can split the workload into 4 different phases: building the base system, profiling, optimizing, and publishing. We can allocate 6 weeks for the base, a week to profile, 4 weeks to optimize, and 2 weeks for publishing. These time frames enclose our goals into achievable sectors. HacknPlan will further break down the workload unto smaller unit tasks but having a Gantt chart simplifies task estimates over time.

TASK	START DATE	DUE DATE	DAYS
<b>1 Base layer systems</b>			<b>46</b>
1.2 Initialization/Clear	15/03/21	25/03/21	11
1.3 Kinematic Update	26/03/21	28/03/21	3
1.4 Collision Detection	29/03/21	04/04/21	6
1.5 Impulse Resolution	05/04/21	12/04/21	8
1.6 Playback Controls	13/04/21	16/04/21	4
1.7 Editor Values	17/04/21	30/04/21	14
<b>2 Profile System</b>			<b>7</b>
2.1 Setup Profiling	01/05/21	02/05/21	2
2.2 Analyze Graphs	03/05/21	07/05/21	5
<b>3 Optimize</b>			<b>27</b>
3.1 Broad phasing	08/05/21	14/05/21	7
3.2 DOD	15/05/21	21/05/21	7
3.3 Multi-threading	22/05/21	28/05/21	7
3.4 Compute shaders	29/05/21	04/06/21	6
<b>4 Publish</b>			<b>16</b>
4.1 <a href="#">Read.me</a>	05/06/21	05/06/21	1
4.2 Wiki	06/06/21	12/06/21	7
4.3 Website	13/06/21	19/06/21	7
4.4 Release	20/06/21	20/06/21	1

Table 1: Gantt Task Breakdown

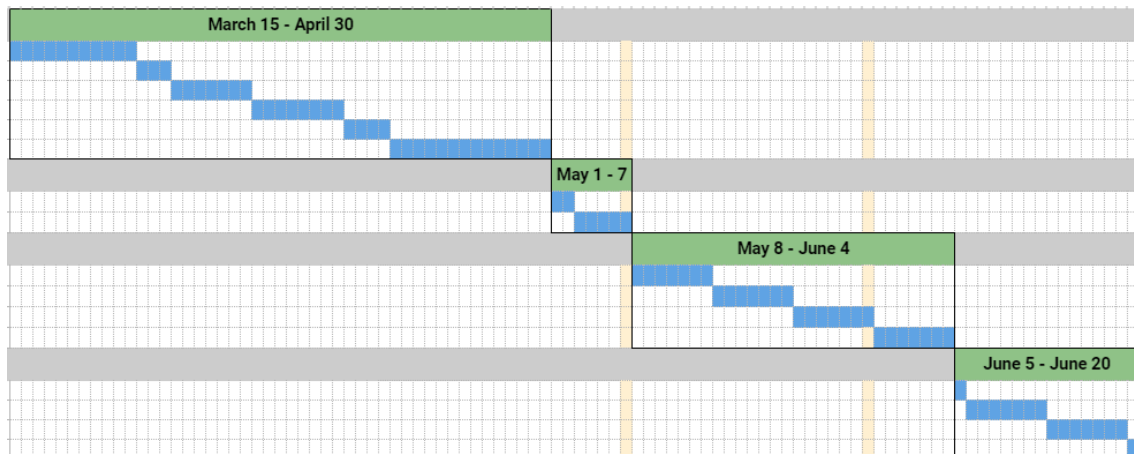


Figure 4: Gantt timeline



## 3.2. SWOT

	Pros	Cons
Internal	<ul style="list-style-type: none"> <li>- We have been working on Redeye for more than 3 years and have improved significantly over the years our teamworking skills. We both feel confident in our personal and group skills.</li> <li>- Working on our own engine gives us a unique advantage as we have complete control and understanding over all systems.</li> <li>- Our only resource is time. Redeye engine requires no financial inversion apart from a single license and spending actual money is no viable solution. Lacking budget could be considered a con; but, because we believe we are able to manage without, we take it as a pro.</li> </ul>	<ul style="list-style-type: none"> <li>- Being used to a certain workflow and methodology, change will bring chaos until we have the changes figured out.</li> <li>- Redeye now counts with many complex systems that have been altered innumerable times. Its unique workflow and entity management limit new modules' integration. Any given obstacles with prior modules are best resolved locally on the new module.</li> </ul>
External	<ul style="list-style-type: none"> <li>- This project is a great opportunity to further expand knowledge on physics software development and improve our engine. Particles are and have been key in video game development. They offer the highest visual feedback to players and range from simple effects to full-on simulations and explosions.</li> <li>- The problem of simulating particles has been solved in thousands of different ways. Every approach is fit to its own target. This may be overwhelming and could be a threat, but there are thousands of resources available online which makes researching smoother, compensating the massive amount of similar software.</li> </ul>	<ul style="list-style-type: none"> <li>- Some developers have started to monetize their development by producing informative media content. This could be a missed opportunity for we could share or stream our process to show our work methodology and gain some funding. For now, it is best to focus on completing the goals first.</li> </ul>

Table 2: SWOT Diagram

### 3.3 Risks and Contingency Plans

Software development is filled with suffering for when debugging we basically act as a detective in a case where we ourselves are also the murderer. Previous bugs encountered while developing Redeye ended in sleep-less nights looking for the hidden errors. Bugs are my biggest fear for they are unpredictable and sometimes a solution may trigger other bugs in previously tested systems. The best way to prepare for them is by defining how to face them beforehand. The possible risks and corresponding procedures are ordered from least to most critical:

Risk	Solution
<b>Task taking more time than estimated</b>	<p>When organizing tasks, we must add extra time frames to be able to include unexpected inconveniences into the schedule. Abusing this solution can lead to no margin for error and we must then decide which tasks can be avoided and left for a future improvement.</p> <p>We both include in our scope the possibility to have enough time to improve our systems. The initial schedule contains optimization tasks whose low priority sets them as de default to cancel in the event of lacking time.</p>
<b>Function triggers crash</b>	<p>Having an error localized increases the chances of it being solved. Knowing what goes wrong we then open a GitHub Issues report, explain the problem, set its priority, and try to add any visual aid.</p>
<b>System compatibility issues</b>	<p>When joining systems, they may not be compatible on how they operate in the workflow. Normally when merging branches this error may occur. Both parties may have different approaches to the fix and must now restructure the schedule to include the new corresponding tasks of making it work. Both hands on deck is the best strategy even during the merging process; just in case.</p>

<b>Unknown crash trigger</b>	These are just nasty. We do not even know what went wrong. Production must stop immediately. These big-time surprise errors require intense testing and debugging from both for any of our systems could be the cause. Follow standard bug reporting procedures and prioritize above all other tasks its resolution.
------------------------------	--

Table 3: Project Risks & Solutions

### 3.4 Initial Cost Analysis

If I were to organize ourselves into a business, we would only have costs for no revenue is expected yet planned. Requisites include the large amount of different software (mainly free), equipment, services, and my own hourly rate. The next table shows a breakdown of this project's estimated costs.

Costs							
	Name	Rates	Price	Amortization Years	Monthly Work Hours	Monthly costs	Quarter costs
<b>Direct</b>						<b>€1,041.42</b>	<b>€3,124.27</b>
Software	GitHub	Monthly	€0.00			€0.00	€0.00
	HacknPlan	Monthly	€0.00			€0.00	€0.00
	Visual Studio	Monthly	€0.00			€0.00	€0.00
	Tableau	Monthly	€60.00			€60.00	€180.00
	Google Drive's	Monthly	€0.00			€0.00	€0.00
	Discord	Monthly	€0.00			€0.00	€0.00
Equipment	Laptop	Amortization	€2,152.36	10		€17.94	€53.81
	Chair	Amortization	€150.00	5		€2.50	€7.50
	Desk	Amortization	€74.50	10		€0.62	€1.86
	Mouse	Amortization	€21.99	5		€0.37	€1.10
Employees	Ruben	Hourly	€8.00		120	€960.00	€2,880.00
<b>Indirect</b>						<b>€70.00</b>	<b>€210.00</b>
Services	Water	Monthly	€15.00			€15.00	€45.00
	Electricity	Monthly	€25.00			€25.00	€75.00
	Internet	Monthly	€30.00			€30.00	€90.00
<b>Total</b>						<b>€1,111.42</b>	<b>€3,334.27</b>

Table 4: Initial costs analysis

The duration of the project's development is around 3 months, a financial quarter. The personal remuneration was estimated to 4 working hours a day engorging up to 86% of the total costs of the project. The remaining 450€ cover the only paid license, services, and equipment amortization. Merging costs with Julià's would double the costs in everything except for the internet connection which is the only cost we could share. Manpower is our most expensive and therefore valuable resource.

Completing tasks before planned will reduce workload and save us money. Targeting essential activities only to produce a minimum viable product can also potentially save even more time and therefore money. Though, more tasks may appear consequently if code quality diminishes; yet again, another double-edged sword.

## 4. Methodology

As a two-people project with no time to spare we will follow the hybrid model of mixing waterfall approach with agile. The plan is set for making features one after the other in a waterfall order based on priority, but new features must be tested and analyzed after their submission. We are a small enough group to be able to react to changes and adapt our strategy. Depending on what we conclude after each added feature, schedule could be prone to changes to target more efficient results. Although we will be working together most of the time, we must reserve a couple hours a week to discuss our opinions on the project's development. These weekly meetings will mark sprint durations to ease task breakdowns.

### 4.1 Tracing Procedures and Tools

Software	Usage
RedEye	Engine
Visual Studio	Code Editor
HacknPlan	Sprint Task Scheduling
GitHub	Version Control & Issues
Google Drive's office automation	Documentation
Tableau	Profiling Analysis
Discord	Communication

Table 5: Software used.

During Redeye production, the methodology used has been improving from just using Trello with a couple of massive tasks at the beginning to now using HacknPlan to organize tasks and milestones and measuring the progress with visual aids. Redeye recently got added profiling procedures to either use Optick or output the profiled data into a json text file that we can later graph using Tableau. We use different software's to carry out all the different tasks that involve adding new features to the engine in the most orderly manner. All these resources are free to use and carry no necessary economic cost (except for Tableau).

### 4.1.1 HacknPlan

We used to work with Trello but found HacknPlan a much more complete tool for organizing tasks, measuring the project's metrics, and sharing individual progress in a group environment. The more in depth we define our tasks, the better feedback our colleagues will reckon. We will continue to use HacknPlan to keep monitoring each other and coordinating tasks.

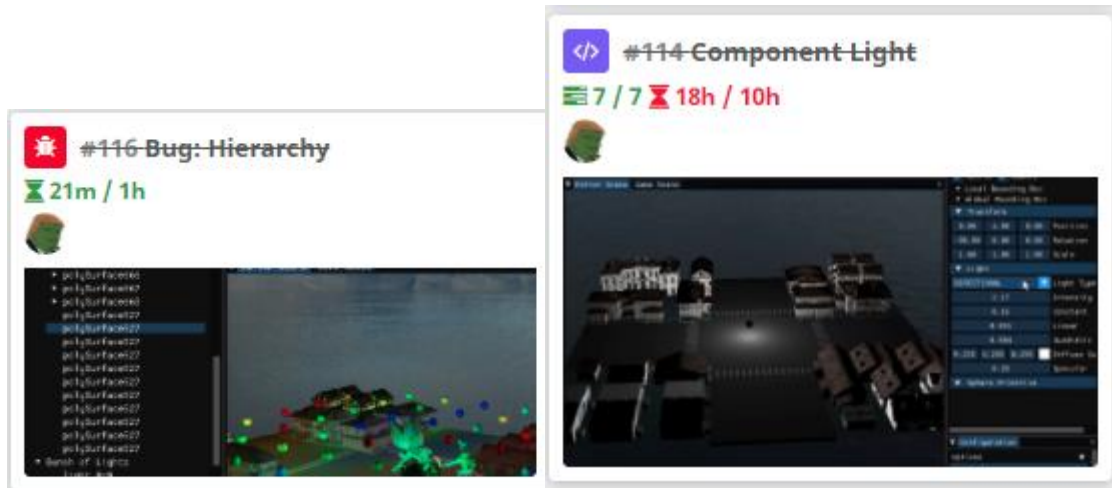


Figure 5: HacknPlan task examples

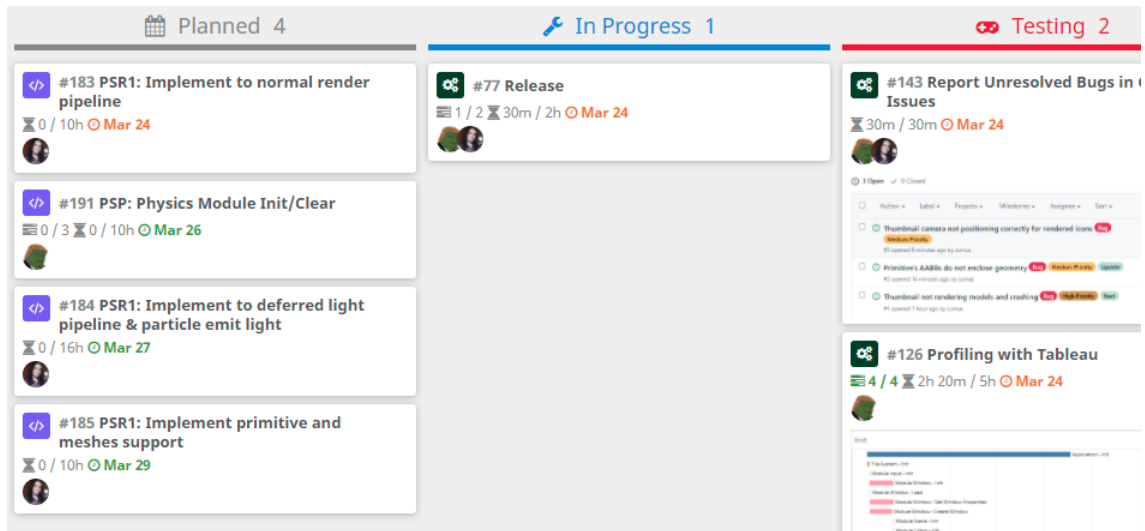


Figure 6: Hacknplan Board Screenshot

### 4.1.2 GitHub Repository and Versioning

Redeye's first commit was in February 2018; the repository is now 3 years old. The project started using only the master branch and has had use of more branches recently. Our branch policy has been ever-changing, but we must now aim for good work practices and fully use branches to ease merging changes.

Releases follow a "major.minor" version numbering system. Every time the engine reaches a milestone, its major version increases. Otherwise, any other release will only increase its minor version.

Bugs were reported creating new tasks in HacknPlan defining the error. For a more open approach to Redeye development, we will now change to reporting bugs using GitHub Issues section. Error reporting will now be categorized by priority and available to anyone viewing the repository apart from our standard procedure of creating an assigned task on HacknPlan.

## 4.2 Validation Tools

To validate internal code, tasks are not deemed completed until they have been reviewed and tested by a peer. Tasks therefore must contain accurate functionality descriptions. Not understanding the task impedes testing it and therefore cannot be approved. This QA workflow has been most useful in a two-person project. It forces both members to understand each other's work.

The physics simulation content does not count with subjective qualifiers. Code style could be critiqued, but we are only going to check objective results: does it work? How many particles does the simulation have? How much time does it take to process? We complete our mission if we are happy with the performance, and it does not crash.

Externally we must share Redeye with fellow developers in search of feedback or error reports; hence, the reason behind adding GitHub Issues to our repertoire to facilitate testers reporting bugs. To ensure software runs on different architectures we must also run the code on as many devices possible. Different machines will output different results. We can profile the differences and visually present them with Tableau portraying the different tested architectures' performance.

## 5. Project Development

We start the project by building primitive versions of the different parts our systems will convey. Once the foundations work in the crudest way possible we progress onto developing the layers into full-out features. The crudeness can later be tackled once we have all the features by changing from a generalist approach to a specialized and optimized one. Redeye is an open-source engine and adding functionality involves providing the complementary documentation for other developers to understand and navigate the code's entrails. The project therefore features its publishing process too.

### 5.1 Integrating Physics

Starting a new project usually has a preproduction period to develop the underlying technologies required to start production. All our technology was up to date and the engine was already there, but it still required some polishing to make way for the new systems. Once everything was ready, adding particles was smoother than envisioned. The first particles shown by the engine are crude, yet functional. The engine did reveal inconveniences, but none are critical to achieving our first steps.

#### 5.1.1 Engine Repository Setup

Previously we both had been developing the engine without a concrete development plan. Since the latest uploaded release from almost a year ago, we added primitives support, an audio module, deferred shading, an overly complex hash table ECS, water & light components, internal profiling, and fixed some code all around the place. We just kept applying into our own engine the new technologies we had been learning about during the degree.

Our first task was to clean the code and finish some ToDo's as to procure a reliable initial infrastructure that we can measure and compare performance after the scheduled changes. The new Readme's version notes are the longest jet after publishing the latest v4.0 release.



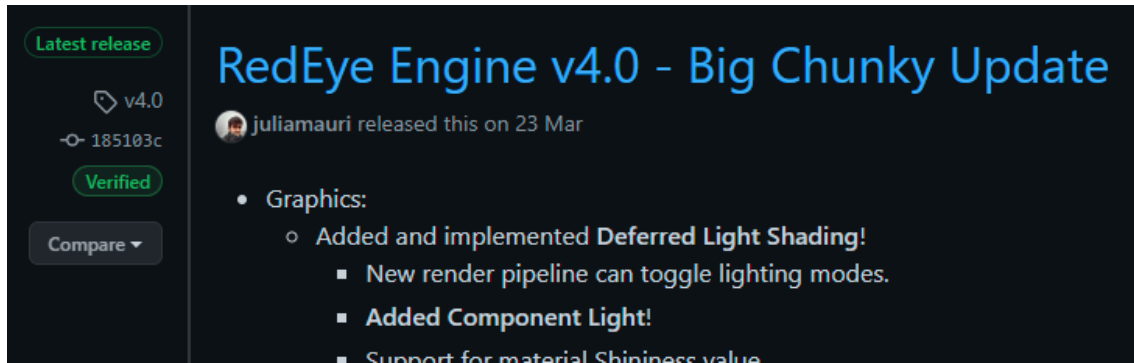


Figure 7: Release v4.0 Screenshot from Github's repository

### 5.1.2 Adding Physics to the Engine's Modules

Now that we were completely ready to start adding everything we envisioned, we started adding the empty template classes. We started rushing the process because setting the particles into the engine's loop had us both locked into the task until we could part ways and start segregating code from each other. If we were to divide the task, merging the changes together would have been horrible for either one of us. Therefore, we took turns as we watched each other's screens to understand the structure we were creating. We agreed on having these 5 main classes to start with:

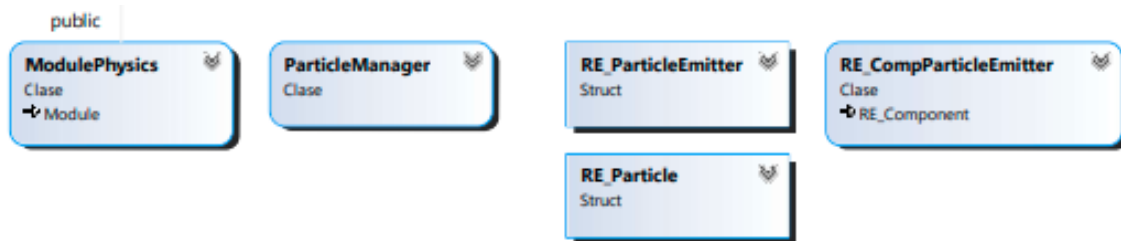


Figure 8: Visual Studio's Class Diagram – particle system classes

The engine's main loop calls to update a main class called App that contains modules and utility classes. The utility classes are for example the file system and resource manager systems that do not require to be updated each frame and therefore have no need to derive from the base Module class. They do however all derive from the Event Listener class (App as well) to be able to receive events. The modules are therefore updated each frame and all control a different aspect of the engine.

The next list shows the modules including the empty physics module. They are iterated in this order:

1. **Input:** stores any user, window, and internal events to be processed at the start of the update.
2. **Window:** stores window's properties and further calls listening modules' that change with the window such as the rendering framebuffer size.
3. **Scene:** stores the gameobject hierarchy and its broad phasing Dynamic AABB Tree that aids at culling the scene for rendering and mouse-picking. The scene module handles updating transformations and other gameobject components through the hash tables and it is best for physics to be updated just after unsimulated changes in scene.
4. **Physics:** our new module that will store all current particle simulations. The focus of my development. The physics module in the future must be able to handle other physics structures such as colliders and rigid bodies, but for now we just need it to contain a Particle Manager. Instead of just adding a Particle Module, we preferred having this extra layer because we want particles to interact with other physics entities in the future.
5. **Editor:** iterates dozens of editor windows to allow the user to edit and control the scene. Also stores some interactions as commands that can be undone and redone using Ctrl + Z and Ctrl + Y shortcuts. This functionality is limited to some parts of the engine, and not a priority to our particle system.
6. **Renderer:** handles OpenGL calls and has different workflows to choose from through the editor. Julià's tasks will mostly follow into improving the renderer with functionalities for the particle system. He is responsible for rendering the output particle simulations and therefore will call from the physics module the simulations that require rendering. It is improbable that I end up editing the renderer as I already have enough with the modules' operations. If the particle output's format changes, we must first agree on given changes as to not impede each other's workflow.
7. **Audio:** uses Wwise to handle music and sound effects playback. The module is very compact and will be worked furthermore on a later stage of the engine's development in the future. For now, the engine can run with Wwise's audio banks to play some demo tracks included inside the data.zip file that stores all engine assets and configurations.

In the next code snippet, we can see the module's structure. It contains its constructor and destructor, the overridden virtual functions from Module, and the particle manager with some utility. The particle manager is set as a private parameter as an extra layer of security to only be handled by the module itself. This module structure is consistent with the rest of modules.

```
class ModulePhysics : public Module
{
public:
    ModulePhysics();
    ~ModulePhysics();

    bool Init() override;
    bool Start() override;
    void Update() override;
    void Cleanup() override;

    RE_ParticleEmitter* AddEmitter();
    void RemoveEmitter(RE_ParticleEmitter* emitter);

    unsigned int GetParticleCount(unsigned int emitter_id) const;
    eastl::list<RE_Particle*>* GetParticles(unsigned int emitter_id) const;

private:
    ParticleManager particles;
};
```

Figure 9: Code Snippet – Physics Module Class

To mitigate huge spikes in the engine, we had the time manager monitor and calculate any milliseconds left after updating the application's main loop. After each frame we display everything the renderer outputs onto our window. But, if the engine's framerate is higher than the monitor's refresh rate, the user will not see it. The way the engine and the monitor synchronize is through VSYNC (vertical synchronization). RedEye already allows for activating or deactivating VSYNC. This graphics technology solves the engine not having to run extra frames and we can exploit this extra time and invest it into other processing blocks such as reading any changes in our assets' zip file or Wwise's audio banks.

```
unsigned int extra_ms = time->FrameExtraMS();
if (extra_ms > 0) extra_ms = fs->ReadAssetChanges(extra_ms);
if (extra_ms > 0) extra_ms = audio->ReadBanksChanges(extra_ms);
if (extra_ms > 0) time->Delay(extra_ms);
```

Figure 10: Code Snippet – Application's Main Loop's last calls

Because of how RedEye works, the render module will not be directly accessing the simulations to draw them. The render calls the scene module to get the rendering IDs (64-bit random numbers). The scene module stores the Entity Component System hierarchy: all the scene's gameobjects and their components using hash tables with their IDs. On getting the call from the render, scene then iterates the hierarchy using a dynamic AABB tree structure to cull meshes and other rendering entities that require no rendering in a broad phasing manner.

This workflow complicates things a little, but because we are the authors, we already know the procedure of adding new components to the hash tables. Also, using hash tables in a DOD style did offer higher performance mainly on scene transformations. We had a nasty lag in v3.2 whenever something moved and now it runs smoothly with barely any noticeable lag.

The first obstacle to encounter was that until the simulations don't adapt the gameobjects' bounding boxes, they wouldn't render unless their origin position showed on screen. It is a slight inconvenient that will be dragged until time to setup particle collisions. There we can iterate and enclose the particles inside a bounding box for the gameobject to signal if the simulation is seen by the camera and thereby flag for rendering. We can further upgrade the AABB by extending their use with broad phasing paradigms.

### 5.1.3 Particle Manager

The particle manager holds all emitters and their particle simulations. To really stress the system and the importance of DOD, the simulations are stored using lists with pointers. After all the research on data-oriented C++ I can be sure to state that it will not run the fastest, but we can check how optimizing data structures really affects our code performance once we are comfortable with the systems' features. As I oversee physics, this is the part of code that I am tasked to ready for Julià's rendering.

```
class ParticleManager
{
public:
    ParticleManager();
    ~ParticleManager();

    eastl::list<eastl::pair<RE_ParticleEmitter*, eastl::list<RE_Particle*>*>*> simulations;
};
```

Figure 11: Code Snippet – Particle Manager Class

The manager is basically a list of particle emitters, and each particle emitter has its own particle list. Having lists will ease the amount of process power required to instantiate an emitter's simulation because we do not require allocating at the start. If we change the structure to vectors and have all the data required allocated beforehand on a continuous space in memory, we free the allocation process from adding new particles between frames on random locations each frame. This will come at a cost in performance when setting up each emitter but run more smoothly between frames.

Once physics are iterated and can be split into processing different chunks of data, we can apply the same extra milliseconds paradigm to the physics module and reduce the vector's space allocation spike. Unity in contrast allows for a special update function mainly for scripting physics behaviors called Fixed Update were even if the game varies between running at 60 and 200 fps, this update will be called in a consistent frequency. This value is stored inside Unity's the physics system and by default is set to 0.02 seconds (50 calls per second) and can be changed though the projects' settings. We could combine both approaches if the system really starts lacking performance. Capping the physics system's update interval will not be too difficult as the physics module already separates the scene's update from physics.

#### 5.1.4 Particles, Emitters & Components

Emitters and their particles are stored inside the particle manager, but to link the gameobjects to the simulations we need a component derived class that will include the simulation into the scene's hierarchy and draw out its editor interface to control the parameters in real time. So, for the scene to return to the renderer its current simulations for drawing, the scene's gameobjects must contain the reference to their particle emissions through an emitter component, same as Unity but with our custom features. Once adapted the scene's hash tables to include emitter components, we can now add a button to the editor for creating a gameobject with a default attached emitter component. Now the user can create emitters and edit their parameters, but because they are not the definitive parameters, scene serialization does not save the emitter data although they should before having finished the particle system. The file system infrastructure and the components' base class have all the functionality ready to be called but we prefer not to invest time into saving the early stages of the emitter component. However, it is a task to be tackled together as we both will be adding more attributes with concurrent features. It is not a huge workload but changing serializations will make previous saved scenes unable to load.

```
struct RE_Particle
{
    bool Update(float dt);

    math::vec position = math::vec::zero;
    math::vec speed = math::vec::zero;

    float lifetime = 0.0f;
    float max_lifetime = 15.0f;
};
```

Figure 12: Code Snippet – Particle Struct

```
struct RE_ParticleEmitter
{
    // Playback Controls
    enum PlaybackState { STOP = 0, PLAY, PAUSE } state = STOP;
    bool loop = true;

    // Simulation parameters
    float spaw_frequency = 3.f;
    float lifetime = 1.5f;
    float spawn_offset = 0.f;
    float speed_multiplier = 2.f;
};
```

Figure 13: Code Snippet – Particle Emitter Struct

For now, we only want points, and get them moving if possible. To achieve our first outline we add position, speed and lifetime to each particle and have the ranges for each parameter set from its emitter properties. Before having to apply custom features, we can first try using random parameters for initial particle values and get some different behaviors between them; no collision, no resolution, just randomly moving somewhere.

```
bool RE_Particle::Update(const float dt)
{
    bool ret = false;

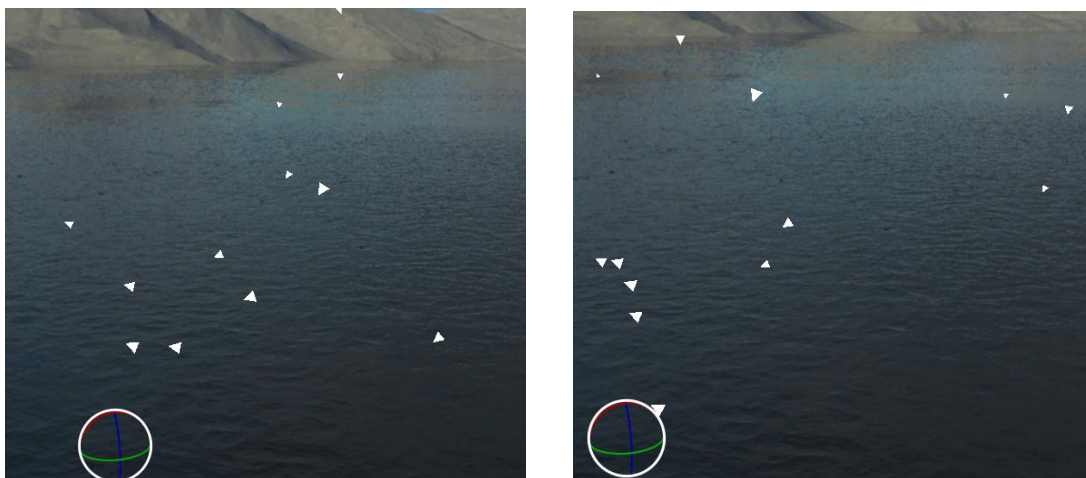
    lifetime += dt;
    if (lifetime > max_lifetime)
    {
        position += velocity * dt;
        ret = true;
    }

    return ret;
}
```

Figure 14: Code Snippet – Particle Update function

The above snippet shows how particles change position and return whether their lifetime is over. Setting the particles velocity to a random vector will emit particles traveling in different directions at a constant speed. Setting random starting positions and maximum lifetime values gives particles individuality. Their particles' behavior remains unvarying just outputting a kinematic movement. Data oriented design dictates for functions to be used outside the data structures' fields but for now, we must work in object-oriented formalities for the practicality of easily modifying our classes. At this stage of the project, optimizing the procedure preemptively could have repercussions limiting the code's scope.

### 5.1.5 Progress Iteration



*Figure 15: Engine's first particle setup screenshots*

Thanks to Julià's procuring a simple triangle VAO to be rendered into the current pipeline we can see our particles showing and moving. Customization is as limited as possible, but the systems fit into the engine with visible results without crashing. Because the implementation was still crude, it was convenient to add simple playback controls to the emitters. Once this setup was complete, our work could be perfectly split between physics and rendering. We now had visual feedback on our progress too, which aids massively when checking and testing instead of reading straight code.

Upon arriving at this point in the project, we now have a rough base over which to add the missing features and customizability. The opening goal of simulating particles inside RedEye is almost complete. It is missing the other two steps for a complete physics system: collision detection and its resolution. The physics module is ready and handles all initializations and clears and updates the simulations. The emitter component is setup with playback and rendering controls but without serialization as it will be added once the component's structure is final. So compared to the initial Gantt, I had to include the engine's setup and split editor values where the second part deals with the added functionalities.

The initialization/clear task included the engine's setup. We thought it was going to be a quicker task and did not consider it as an independent task. But, thanks to spending the needed time into a clean release, we reduced the amount of time we were going to have to spend adapting the particles to the engine later. Luckily, although the tasks have moved around, it seems the work is up to schedule, and the Base Layer Systems bracket will maintain its concluding data.



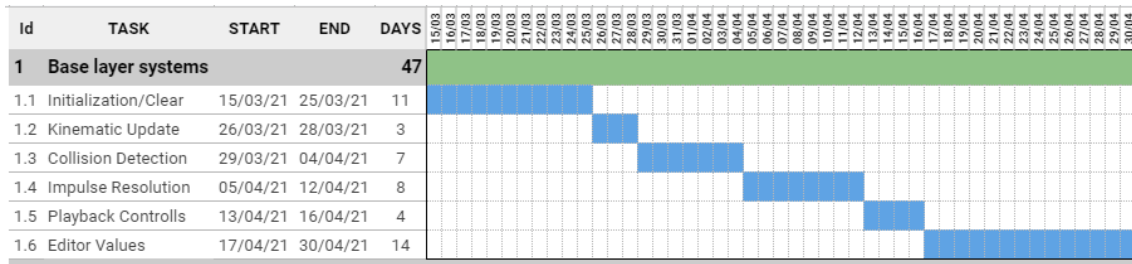


Figure 16: Initial Gantt v1 – Base layer systems

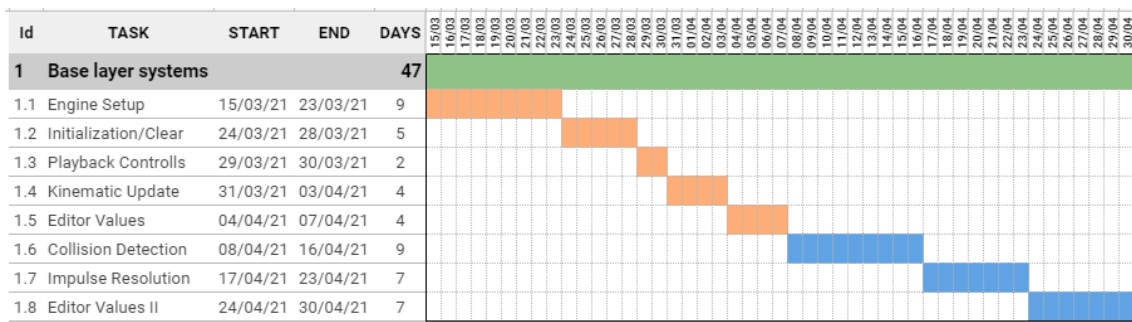


Figure 17: Adapted Gantt v2 – Base layer systems

Recorded bugs during this period refer to the crashes in the thumbnail and the scene's systems. They do not directly impede the particles' system, but having the engine sometimes crash the first time it runs on a new device is troubling. We must tackle this error, but for now, let us leave it alone to fix it later. There are also several inconveniences such as the scene culling the particles if the origin does not show inside the camera and the new emitter component missing its serialization. All these future tasks are estimated and registered to take into consideration once the base layer is finished.

## 5.2 Collisions and Resolutions

The next bracket finishes the base layer systems. Particles finally get physical properties and behaviors. We are having a clearer picture of the desired result and the correct way to achieve it. It proved the need to add visual feedback to the simulation together with more features and changes to the planned tasks.

### 5.2.1 Collisions

Before being able to apply a force to a particle, we must first iterate all particles to find if any intersect. In the most simplified manner, two different points should only collide once their 3d positions are identical. 4-byte floats give enough decimal range for volume-less particles to barely collide. But that is just for collision between point particles, the other collisions we focus on are sphere and box colliders.

The most straight forward approach is checking each particle against each other. Because we do not check two objects twice, the complexity of the system turns into a triangular number sequence instead of straightly exponential. Its complexity can be expressed using Big O notation as:

$$O\left(\frac{n \times (n + 1)}{2}\right)$$

The broad phasing paradigms previously mentioned can reduce the amount narrow collisions to be checked each loop, but improving performance is scheduled for a later stage of the project. Comparing both algorithms will output a clear comparison between the different ways to approach the problem.

To check if a particle is colliding, we calculate the difference between the particles' positions. We use that vector and apply a dot product with itself, which gives us the unrooted length. If this distance is smaller than the squared combined radius, then particles do collide. Square rooting distances exerts a performance toll and this way we skip having to process roots if there was no collision.

Once we assert the collision and two particles are overlapping in space, we then proceed to process the rooted value and calculate the minimum translation distance (MTD) to resolve the intersection. This distance can be extracted using the actual distance

between them and taking advantage from the previously calculated position difference vector.

For  $(a_x, a_y, a_z)$  and  $r_A$  being the position vector and radius for particle A and  $(b_x, b_y, b_z)$  and  $r_B$  the position and radius for particle B:

$$dist_{A,B} = |AB| = \sqrt{A \cdot B} = \sqrt{(a_x \times b_x) + (a_y \times b_y) + (a_z \times b_z)}$$

$$MTD = (B - A) \times \left( \frac{r_A + r_B + dist_{A,B}}{dist_{A,B}} \right)$$

Once we have the MTD, we just need to move the particles away from each other based on the mass ratio between particles. We can conclude the resolution with this translation. For  $m_A$  and  $m_B$  being the respective masses of the previous A and B particles, the change in position ends up being:

$$MTD_A = MTD \times \frac{m_A^{-1}}{m_A^{-1} + m_B^{-1}}$$

$$MTD_B = MTD \times \frac{m_B^{-1}}{m_A^{-1} + m_B^{-1}}$$

```
// Check particle collision
const math::vec collision_dir = p1.position - p2.position;
const float dist2 = collision_dir.Dot(collision_dir);
if (dist2 <= combined_radius * combined_radius)
{
    // Get mtd: Minimum Translation Distance
    const float dist = math::Sqrt(dist2);
    const math::vec mtd = collision_dir * (combined_radius - dist) / dist;

    // Resolve Intersection
    const float p1_inv_mass = 1.f / p1.mass;
    const float p2_inv_mass = 1.f / p2.mass;
    p1.position += mtd * (p1_inv_mass / (p1_inv_mass + p2_inv_mass));
    p2.position -= mtd * (p2_inv_mass / (p1_inv_mass + p2_inv_mass));
}
```

Figure 18: Code Snippet – Particle Inter-Collisions

To keep the performance cost low, we removed particle's individual data and dealt with them as if they were all identical. The problem arose when adding light to each particle from Julià side of the project. The amount of light data the GPU can handle is based on its hardware properties. Having the particles contain this data limited the number of particles being able to render lights. So, to simplify work, the emission contains all the particle data that will be later adapted for the user to choose the configuration. Therefore, we should aim for particles to have varying number of parameters depending on the type of simulation, just like similar software's using tables to view their simulation data.

Other improvements upon the system can be extracted from looking at its subtle lack of perfect accuracy derived flaws:

- Dead Particles Colliding: The iteration first checks if the particle is dead before starting to look for collisions, but the particles it measures against does not check if it is also alive until it is his turn to iterate. The delta time is usually too small to cause anything too big of a deal.
- Missed Collisions during Spikes: If the delta time were to be too big to cause particles to miss a collision because the engine spikes on a given moment, the problem is not the particle system, it is the other module that caused the spike. But because a computer has many programs running at the same time, it could as well be externally caused. Previously it is mentioned the different ways to split the operations, but they also convey inherent derived flaws plus add to the amount of workload.
  - Divide delta time and iterate multiple times the system.
  - Unity-like fixed update, but this one is best for the opposite case where we have too many frames.
  - Enclosing both before and after positions into a primitive shape (line, sphere or even AABB) to tackle any possible intersections in its trail. This method is overkill, but we do take it into consideration.
- Curved vs Straight Movements: the changes in particle position are measured by adding its direction directly over its position scaled by the delta time. If we were to draw its change in space, it would be a straight line, but the actual movement is a curve. Runge Kutta methods are used by more accuracy-centered simulations to approximate the value of the intervals in this curved movement when changing from kinematics to dynamics.

If the engine's focus was not videogame production, these situations may be interesting to solve, but accuracy is not a priority as many simulations are to be run in parallel. Performance is a bigger funnel in our situation because of the engine's utility. The current structure iterates collisions from simple emissions and triggers when the particles intersect. When reaching the profiling stage, we will stress test it to see if the structure holds its purpose or if it should be upgraded.

### 5.2.2 Impulse Resolution

Once particles collide between themselves using different collider shapes, we have the trigger to act over the imminent intersection of space. Now to act over this junction, each particle must apply force to each other and end up moving elsewhere. Therefore, we change the direction parameters' use to now hold the force each particle carries. Force is measured in newtons ( $\text{kg/s}^2$ ), which means that particles need mass to extract the force ratio between them. Because the particles are all identical for simplification purposes, they all have the same mass stored inside the emitter. A dampening value for softening the collision rebound strength of softer solids was also added.

To resolve the change in speed derived from particle collisions we now evaluate over their speeds who receive a positive or negative impulse. The dampening value sets the particles' surface restitution to soften or increase the applied impulse.

Using the same particles, A and B, from the previous equations, we let  $V_A$  and  $V_B$  be their respective speed vectors and  $rest_A$  and  $rest_B$  their restitution values:

$$N_{A-B} = \text{Normalized}(A - B)$$
$$\text{Impulse} = - \left( \frac{(rest_A + rest_B) \times ((V_A - V_B) \cdot N_{A-B}) \times N_{A-B}}{m_A^{-1} + m_B^{-1}} \right)$$
$$V_{A \text{ final}} = V_A + \frac{\text{Impulse}}{m_A} \quad \text{and} \quad V_{B \text{ final}} = V_B - \frac{\text{Impulse}}{m_B}$$

```
// Resolve Collision
const math::vec col_normal = collision_dir.Normalized();
const math::vec col_speed = (p1.velocity - p2.velocity);
const float dot = col_speed.Dot(col_normal);

// Check if particles not already moving away
if (dot < 0.0f)
{
    // Resolve applying impulse
    const math::vec impulse = col_normal * (-(p1.rest + p2.rest) * dot) / (p1_inv_mass + p2_inv_mass);
    p1.velocity += impulse * p1_inv_mass;
    p2.velocity -= impulse * p2_inv_mass;
}
```

Figure 19: Code Snippet – Collision Resolution

Calculating forces from two collided entities may cause a third entity to apply the same amount of force in the opposite direction, negating the previous impulse and therefore getting them stuck inside each other. Adding constraint support for particles to free themselves from complicated spaces is a solution in many videogames, but unapplied to particles. As they are smaller in size and bigger in quantity, constraints can easily scale in required performance. We cannot check how much of a problem stuck particles really are in our case because the particles are boundless and floating in space.

To really visualize this process and be finally able to judge more effectively, I must add the option to bound the simulation's space and render some debug geometry. Getting them to collide with other entities apart from between themselves is part of the original plan, but at this point of the project, the changes are perceivable in the particle behaviors when playing the simulation. We are missing being able to view more information about them. Rendering is Julià's responsibility, but I must partake in rendering the debug geometry and be able to portray the added behaviors.

The last aspect to take into consideration from collisions is the rotation problem. Point particles have no direction, but 2D planes do. When drawing a sprite, two triangles in the shape of a square are positioned at a point in space to be rendered from the camera's perspective, but it could face anywhere. The editor allows for the user to set up where the particles face: billboard (face the camera), face the direction of its gameobject's transform or a user inputted custom direction. For drawing sprites, this is a perfect standard solution. But if these particles were to have volume and collide, they would rotate depending on their shape and center of gravity. Because the emitter stores a VAO to render, it can be changed from two triangles to a full mesh. Plus, because we have pre-calculated AABB's for imported meshes, we can already apply broad phasing and cull iterating triangles if there is no intersection with its box. Colliding meshes always face the same direction though and would require rigidbody physics setup to achieve 3d rotations.

### 5.2.3 Progress Iteration

Now that the first stage of the project is complete and the particles use a physics based impulse simulation, the goal is technically complete, but it lacks so much feedback and user options. The direction the project must now take is towards polishing the results into an organized-looking and efficient tool. Having this first stage locked into the estimated time frame successfully concludes the integration of physics' base layer systems. Check the Gantt's final form in the leading figures:

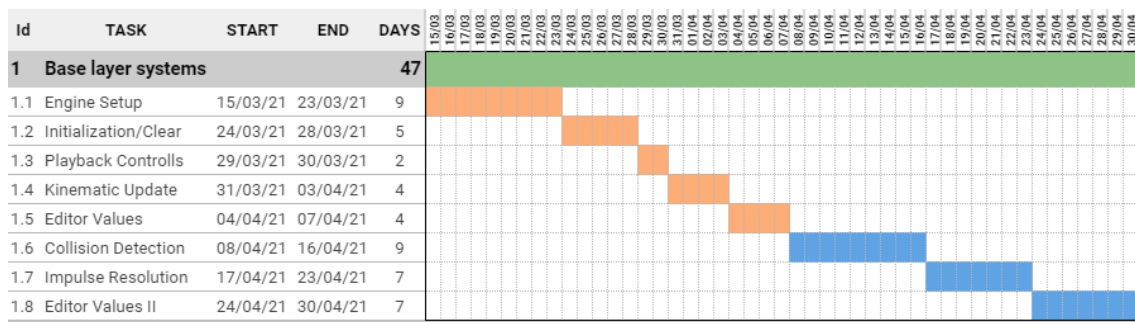


Figure 20: Adapted Gantt v2 – Base layer systems

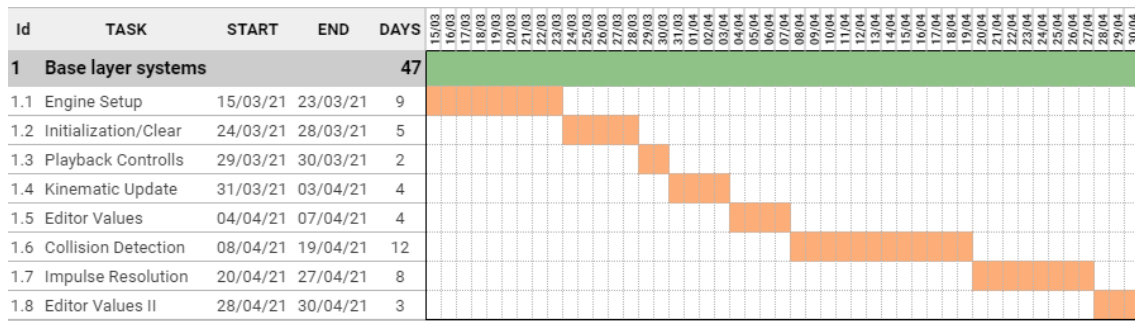


Figure 21: Final Gantt v3 – Base layer systems

Until this point, the methodology used has been key in keeping each other updated on what tasks and documents were being worked on. We even felt as if we knew too much about each other's coding problems and inconveniences. Understanding each other's point of view helped ease making decisions about the development of the project. Testing each other's tasks also helped us identify errors.

The negative side of working too close on the same project is that we were editing the same documents all the time. We were to have no dependencies because our codes

operate at different places, but in practice we partook in each other's fields and team worked through them. The particle emitter was always changing and made it inconvenient to synchronize our work. Because rendering goes after physics, I was always worried that something I changed could impede Julià's rendering (which happened often) and therefore decided to hold my changes locally depending on the engine's state.

Some contemplated risks arose. Majorly we both had tasks starting to complicate too much and we ended up taking longer on our tasks. Our contingencies allow for the schedule to alter and adapt to these unavoidable problems. The two weeks reserved for multi-threading and compute shading are hereby dismissed. Although unanticipated in the initial planning, 3d rotations and editor specific tools such as mouse-picking are also out of our range and direction. With the freed time, we had to prioritize what could most benefit the project to develop. The duration of the project did not change and remains with the same initially estimated costs. Check out the differences between Ganttts:

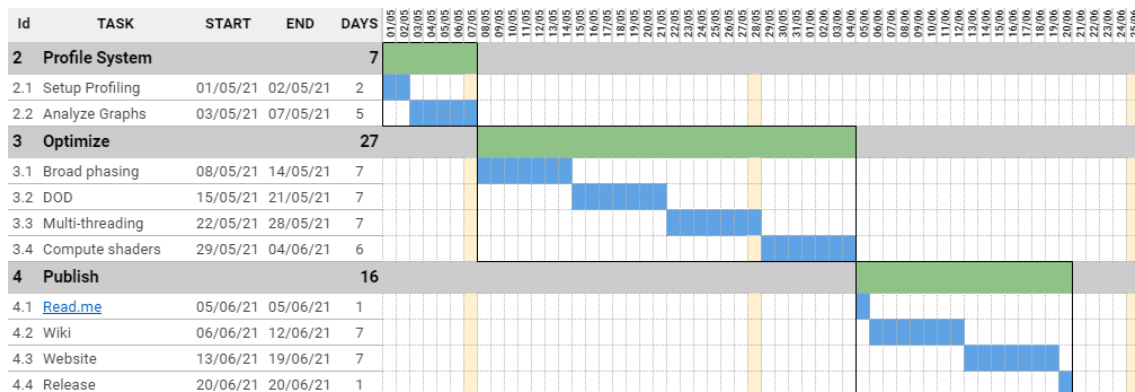


Figure 22: Initial Gantt v1 – Profiling, Optimizing & Publishing



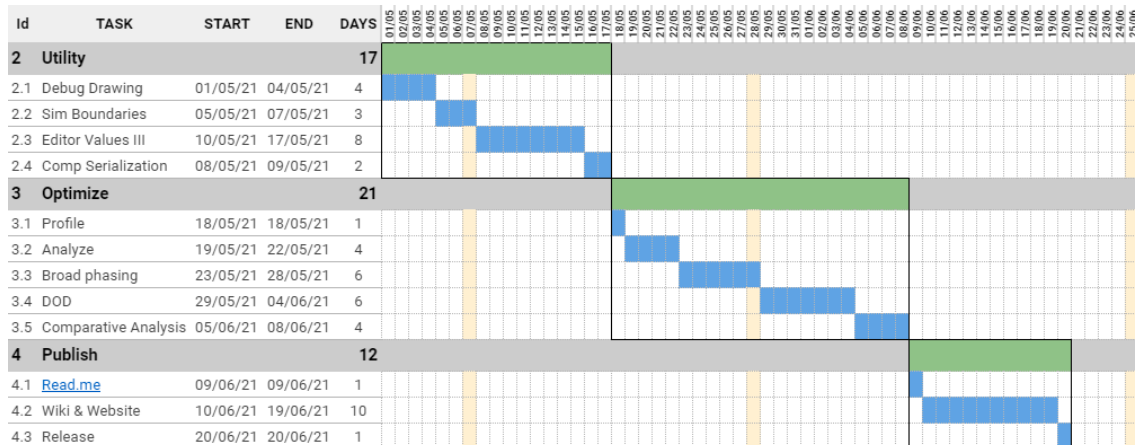


Figure 23: Adapted Gantt v2 – Utility, Optimizing & Publishing

Optimizing and profiling now belong to the same bracket and have about the same amount of time reserved. This procedure offers a better way to portray measuring before and after profiling data. If profiling is only done before adding new paradigms, we are losing the chance to compare and further analyze the systems performance. We reduced publishing workload by 4 days (-25%) because we still maintain the 5-day margin and we prefer to invest in better code than in better documentation.

With a freed schedule we decided to use this time to add a Utility bracket that includes new tasks that have been appearing through the development:

- **Debug draw:** playing simulations and changing editor values apply at real time. It does return feedback, but its intricacies are hidden. To visualize the physical world simulation, colliders should be able to be seen. Checking for errors will be simplified if it shows itself instead of just the end-result.
- **Boundaries:** adding other objects for particles to collide with will further complement the visual feedback of the system. Enclosing simulations will improve broad phasing and can be positioned to trick the user into feeling the particles collide with the environment.
- **Editor Values III:** the task of editor values encapsulates the different interface interactions for the user to customize the emitter. I tried adding these features whilst building the base layers, but they ended up impeding the workflow and were best to leave for a after completing the base.

1. Local vs global emission: Instantiating particles in the gameobject's local space or using its transform to position them globally.

2. Inherit force: track the gameobject's changes in motion to instantiate particles with a derived impulse.
  3. Intervals: Add option for values to be capped or randomized between intervals for when instantiating the particles.
  4. Start delay.
  5. Continuous vs burst emission: offer quick preset configurations for the user to quickly produce the desired spawning effect.
- **Component Serialization**: Once all the data is settled, serializing them will just be going attribute by attribute and storing it in a Redeye scene file. The problem of making serialization beforehand was that we have been changing non-stop the emitter's variables and would have had to change their serialization process on every step.

## 5.3 Utility

This section of the development arose from the substantial number of customizable features we felt were missing to pass as a complete engine asset. The broad spectrum of properties will force us to create a designated working space editor window. All these attributes are handled and referenced through Redeye's file and resource systems.

First step here is drawing the real simulation. Redeye uses other debug drawing methods and we easy to copy and adapt. We also adapted methods for collision detections and resolutions on boundaries. Colliders now will bounce off walls. Once utility concludes, straight to finally optimizing code.

### 5.3.1 Boundaries

First task was drawing the particles volume to aid at visualizing the particles colliding between themselves. Once debug drawing these primitive geometries, we could now step into adding boundaries. These boundaries contain geometry over which the particles will have to check collision if active. We can also cut the process short by forgetting to resolve intersections and just kill the particle. If not, collision must be resolved, and impulse must be applied. We skip impulse only if particles are already moving away or parallel to the surface of the boundary.

Bounds enclose the simulation, and their aim on the long run is to falsify the simulation's surrounding. Three different shapes for boundaries can be selected through the editor.

## Plane

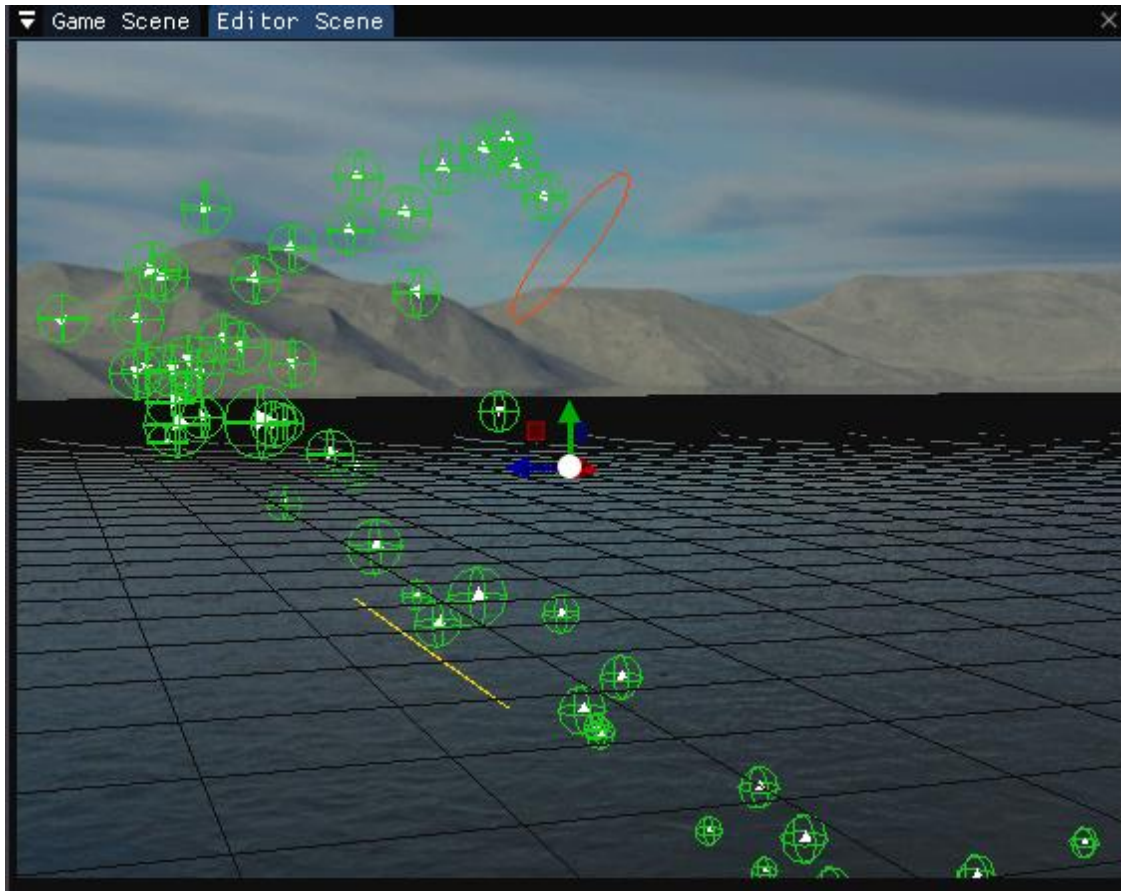


Figure 24: Plane Boundary Collision Screenshot

Planes split 3d space in 2 sides: positive and negative. With a normalized direction vector ( $a$ ,  $b$ ,  $c$ ) and the distance ( $d$ ) from origin we can represent any plane:

$$ax + by + cz = d$$

The complicated part arrives when procuring the user with the option to rotate the plane. We obviously must convert to Euler angles, but 3D vectors can only be evaluated to polar coordinates as roll in a direction vector is redundant. Yaw and pitch are therefore able to be edited though the component's editor, but at straight angles, it gets Gimbal locked and changes rotation order. We could spend some time into adapting the editor's gizmos to comfortably transform geometries inside the emitter. But given that we estimate it to be a huge workload, we settled on just converting the values shown from radians to degrees and allowing the user access to edit them.

## Sphere

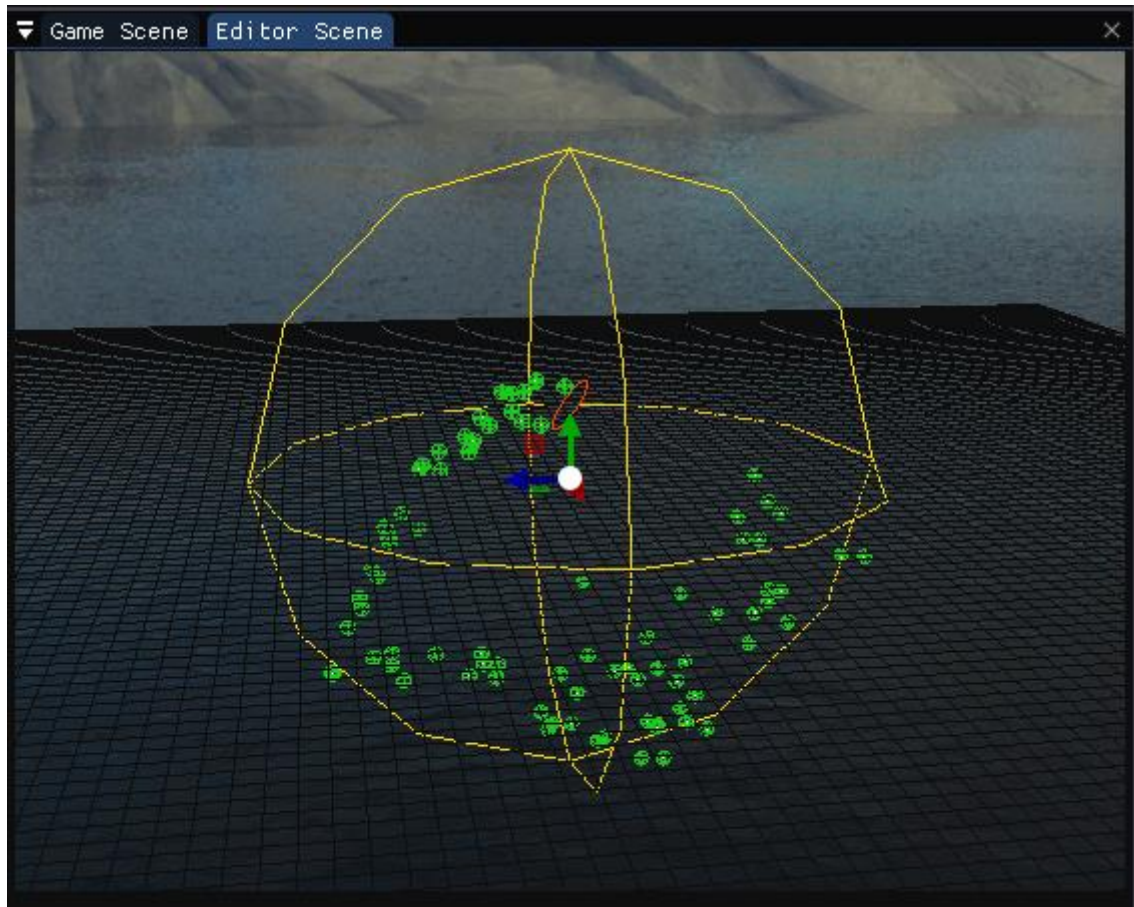


Figure 25: Sphere Boundary Collision Screenshot

Spheres' surfaces are at a constant distance ( $r$ ) from its origin ( $a, b, c$ ).

$$(x - a)^2 + (y - a)^2 + (z - c)^2 = r^2$$

Checking the distance between the particle and the sphere's surface is straight forward as the distance from the sphere's origin can be measured by subtracting it from the particles position. We must then convert the vector into a scalar, and it now gets tricky as square roots are not performance friendly. Therefore, before extracting the actual distance between the surface and the particle, we compare with the un-rooted values.



### Axis-aligned bounding box (AABB)

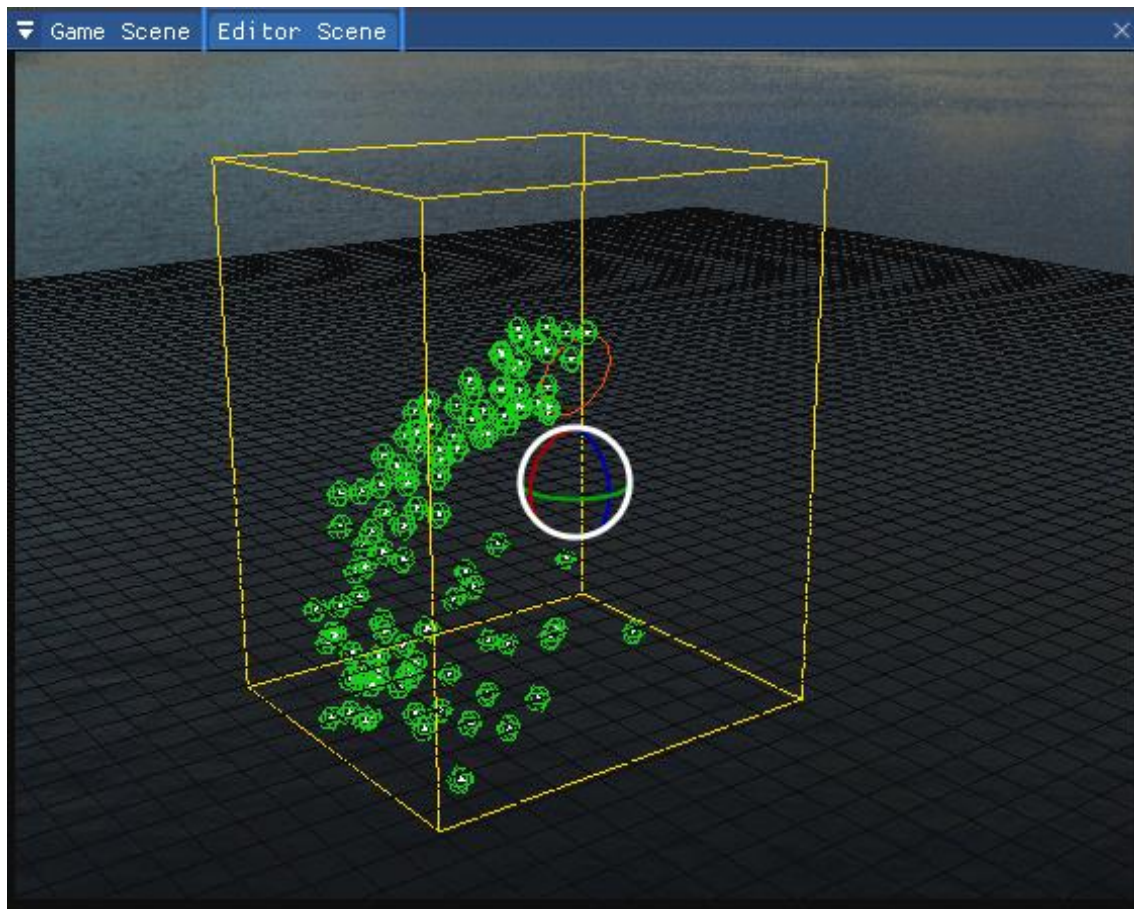


Figure 26: AABB Boundary Collision Screenshot

AABBs are composed of 6 planes, but because they all face in the cardinal directions, we only need the lowest and highest position values. Therefore, it is not 6 times more costly than a single plane. Having simulations bound with an AABB allows for a direct integration into Redeye's broad phasing data partitions.

### 5.3.2 Editor

After iterating many times over the particle emitter structure, it is full to the brim with values pertaining to instantiation, physics, rendering and lighting. To organize all of them, we try to emulate Unity's approach with a module-like interface adding booleans to filter procedures and simplify the properties tab. Therefore, we added an option to open an emitter workspace to better manage and organize the interface. The next screenshot shows the properties the user can edit.



Figure 27: Particle Emitter Properties Screenshot

At the start we have playback controls with buttons and some labels showing key simulation indicators:

- **Particle count:** the current number of active particles in the simulation
- **Max distance:** the furthest distance from game object's position. It is stored squared and rooted only for editor's purposes. Having it squared reduced having to perform a square root each frame per particle.
- **Max speed:** the highest speed magnitude from particles. Also stored squared.

Then follow instantiation parameters which decide particle values at spawn. Before we only had single values for each parameter. Now some values such as particle maximum lifetime can be changed to an interval defined within the emitter. For initial speed, it expands from a vector to a vector and its different dimension combinations for its intervals.

- **Time multiplier:** scales the speed of simulation playback.
- **Spawn frequency:** defines number of new particles per second.
- **Lifetime:** defines the amount of time particles will live. It can be set to a random value between a given interval.
- **Speed:** Sets the initial speed a particle will hold when spawned. It can be set to a random value between a given interval.
- **Emitter Shape:** defines the area over which new particles will appear. This gets complicated to handle; because, if many particles are spawned with physics within a reduced amount of space, they will overlap, and some spawned particles will have an unnatural instantaneous resolution. So be warned of its mishaps when limiting particle space. Bounding the space to even smaller margin for particles will saturate processing time with constant collisions. To attack this issue, we setup 6 different types of emitter shapes: Point, Circle, Ring, Sphere, Hollow sphere and AABB.



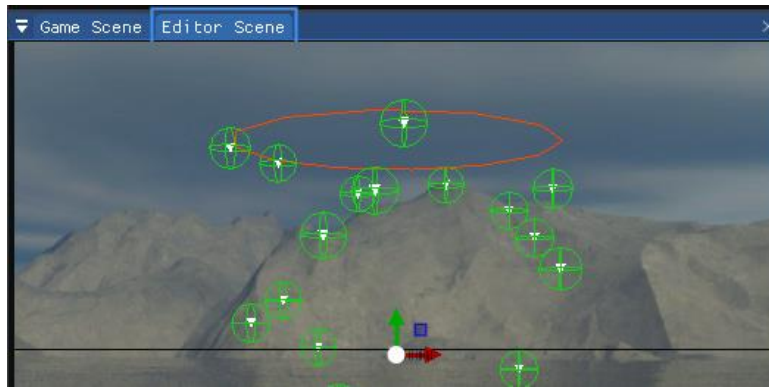


Figure 28: Circular Emitter Simulation Screenshot

The physics part of the emitter holds all the data necessary for particles to collide mass, collider radius and collider restitution. Each attribute can be set as a fixed value or a random value within an interval.

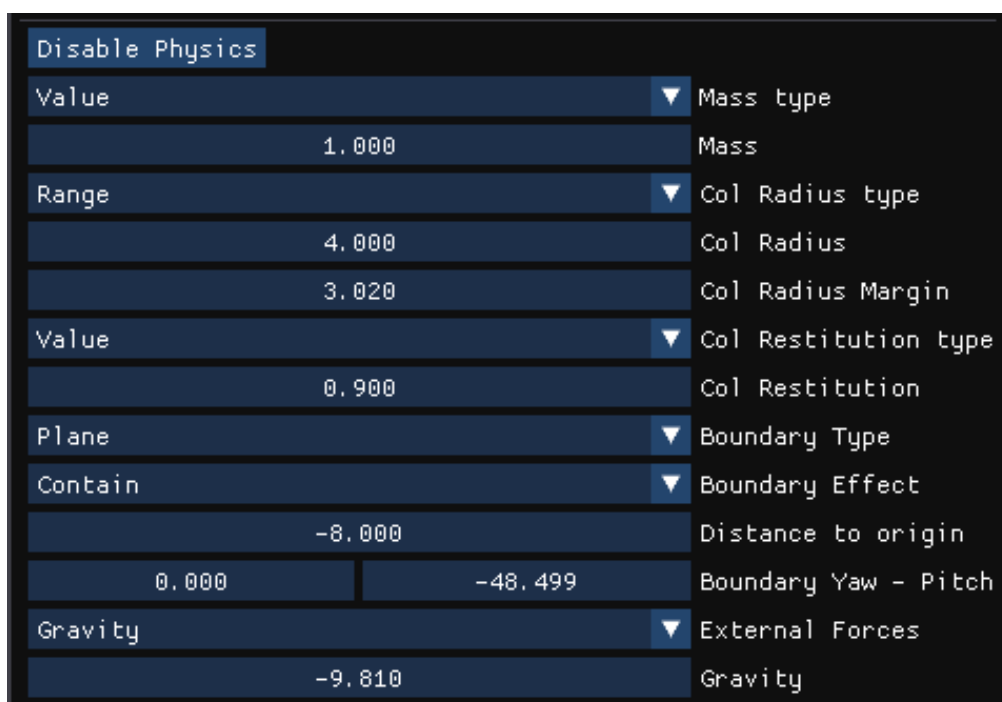


Figure 29: Particle Emitter Physics Parameters Screenshot

These values are stored per particle for when any are set to an interval. Physics also contains constant external forces to be applied to the particles and the previously disclosed boundaries. External forces contemplate having a wind vector and a gravity magnitude for the user to use just one of them or mix them both.

### 5.3.3 Particle Emitter Workspace

A higher amount of user customization made the properties panel that shows the emitter component over crumbled. Testing the simulations made us scroll too much for comfortable navigation. We also knew that more editor values were on their way. To address the usability issue, we moved the fields to a designated editor window. Julià procured the base heading organization setup and even prepared it with an independent viewport for visualizing isolated simulations. I will focus on the physics and instantiation parameters section as the rendering options now also include curve editing for variable attributes such as particle color and opacity. The editor window approach lets us stack several headings on a single area reducing the amount of mouse wheel scrolling necessary to access all fields. I ended dividing the properties into 3 prominent sectors.

#### Spawning



Figure 30: Particle Emitter Workspace Screenshot - Spawning

The Spawning tab enables modifying the way the particles appear. The previous figure shows the highest number of editable fields available because all options have been set to show the most extensive fields. Different configurations limit the user options and redundant fields will hide to reduce the amount of editor space used. Setting the Workspace window gave the opportunity to sneak in new added features:

- **Intervals:** emission can be set to enable spawning at intermittent intervals or operate at different on and off intervals.
- **Spawning Method:** the default emitter is set to spawn a constant stream of particles and now can be changed to bursts. It requires defining the number of particles per burst and the time between bursts. Particularly useful for explosion effects where particles start flying in different directions at the same time.
- **Timing:** digging into the engines complementary systems, particles now react to the scene's playback controls and have the option to start on play.
- **Parent Dependencies:** parent position and speed are shown in the status tab and used when changing the emission from local to global space instantiation or if particles inherit the gameobjects speed. Because of the dynamically handled hash mapped ECS, we cannot store the gameobjects pointer for easy access and so the component handles the bridge between by sending the data to the emitter.

### Status

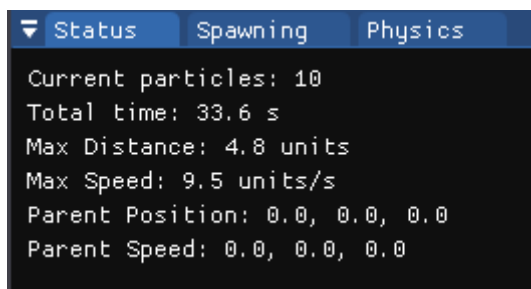


Figure 31: Particle Emitter Workspace Screenshot - Status

Here no values can be edited by the user; they are read-only. We added the total time and parent's position and speed. These new values are stored by the emitter and updated the emitter's simulation playback. The emitter component acts as a bridge sending parent information to the actual emitter.

## Physics

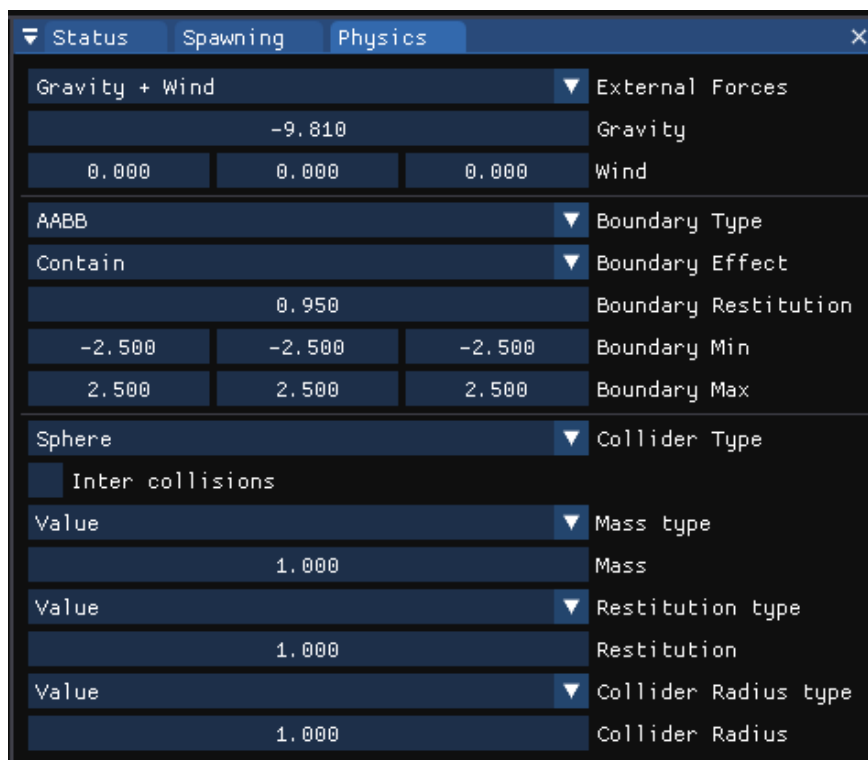


Figure 32: Particle Emitter Workspace Screenshot - Physics

The Physics tab enables modifying the way the particles interact during the simulation with the boundaries and each other:

- **External Forces:** adds constant forces to each particle as to simulate gravity and/or wind.
- **Boundary:** boundaries have 3 different shapes available and editable through the ImGui editor fields. The need to resolve collisions can now be removed if we kill the particles when reaching the boundaries.
- **Collider:** Boundaries have no effect on collider-less particles. To check collisions, they must contain mass and restitution properties; and a radius, given they it is set to spheres.

### 5.3.4 Component Serialization

Now that the emitter contains all the necessary properties and reacts to scene events, we must now adapt the engine to save emitters into the already serializable scenes. Not all emitter attributes are to be serialized. The emission id, playback state, the particles container and the status' read-only values. The values that are going to need saving are the spawning and physics properties. Saving basic data types like floats, ints and bools are straight forward, but the rest of attributes may save different amount of data from different configurations.

```
// Playback
bool loop = true;
float max_time = 5.f;
float start_delay = 0.0f;
float time_multiplier = 1.f;
bool start_on_play = true;

// Spawning
unsigned int max_particles = 5000000u;
RE_EmissionInterval spawn_interval = {};
RE_EmissionSpawn spawn_mode = {};

// Instantiation
RE_EmissionSingleValue initial_lifetime = {};
RE_EmissionShape initial_pos = {};
RE_EmissionVector initial_speed = {};

// GO & Space
bool local_space = true;
bool inherit_speed = false;

// Physics
RE_EmissionExternalForces external_acc = {};
RE_EmissionBoundary boundary = {};
RE_EmissionCollider collider = {};
```

Figure 33: Code Snippet – `RE_ParticleEmitter` serializable variables

The scene serializes every time we hit the play button to be able to return the entities and their components the same values they contained before running the scene. Serializing a Redeye resource constitutes methods for saving in both json and binary mode. Binary mode is preferable to a json structure as it allocates bit by bit the values and will read much faster than parsing a string of text. These files will be saved in the `/Assets/` directory.

```
void RE_ParticleEmission::BinarySerialize() const
{
    RE_FileBuffer libraryFile(GetLibraryPath(), RE_FS->GetZipPath());

    uint bufferSize = GetBinarySize() + 1;
    char* buffer = new char[bufferSize];
    char* cursor = buffer;

    size_t size = sizeof(bool);
    memcpy(cursor, &loop, size);
    cursor += size;
```

Figure 34: Code Snippet – RE\_ParticleEmission Binary Serialization extract

Through the json composition, an MD5 encryption algorithm encodes the string into a unique abbreviated string used to name the meta file for the resource. These uniquely named meta files help us identify if two resources are identical and are stored in the /Library/ directory. These mentioned directories are stored inside the compressed data.zip file. All the engine's data files are stored there, and the only other directory included is /Settings/ containing configurable data about the engine's configuration.

This is key to keeping track of the number of entities in the scene referencing the same resource as to control not having duplicate instances of the same data. The reference counting paradigm is a huge optimization as it allows for many instances of the same entity without a duplicating the amount of data the resource allocates. We even went through the trouble of separating the emitter resource into its rendering and emission parts. The pipelines for rendering and simulating have been build parallel to each other and have a clear separation between the values' usage.

```
void RE_ParticleEmission::JsonDeserialize(bool generateLibraryPath)
{
    Config emission(GetAssetPath(), RE_FS->GetZipPath());
    if (emission.Load())
    {
        RE_Json* node = emission.GetRootNode("Emission");

        loop = node->PullBool("Loop", loop);
        max_time = node->PullFloat("Max time", max_time);
        start_delay = node->PullFloat("Start Delay", start_delay);
        time_multiplier = node->PullFloat("Time Multiplier", time_multiplier);

        spawn_interval.JsonDeserialize(node->PullJsonObject("Interval"));
```

Figure 35: Code Snippet – RE\_ParticleEmission Json Deserialization extract

### 5.3.5 Progress Iteration

Second stage of the project completed. Particle's emitters are full of customizable parameters. Particle colliders, emitter shapes and boundaries render debug geometry for a proper visual feedback of the procedures The editor's workspace window allows for more comfortable customization and the emitter's use Redeye systems to serialize and reference count emissions.

It took 4 more days than expected and will unfortunately dispense with the physics' parameter dependencies. Publishing timeframe has yet again been cut down to absorb the excess workload. Check the Gantt's updated form in the leading figures:

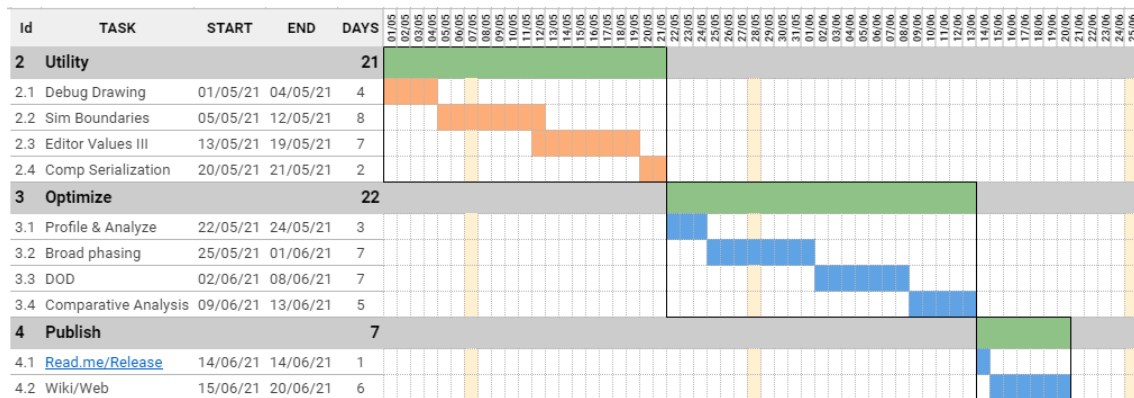


Figure 36: Adapted Gantt v3 – Utility, Optimizing & Publishing

Now that all necessary systems are set, we can start to measure our performance and analyze any optimizations we can squeeze. Once graphed the engine's raw timings, we will adapt the boundaries to compliment broad phasing calculations. We have 3 full weeks dedicated for optimization, 1 week for repository publishing and still save 5 just in case margin days.

## 5.4 Optimizing

Optimizing a system encloses any type of change improving efficiency. To properly document and present the effect of these changes, we must first build a control state. Once we have the base results over to which compare, we can then extract the impact they have.

As to the broad optimization options available, we settled on applying broad phasing and data-oriented design paradigms. Throughout development we have been building many different ideas as to how to apply the paradigms. We intentionally applied object-oriented principles so that we could later upgrade them during the optimization stage. If we started building it as fast as we could from the start, we would miss the chance to apply and measure it.

### 5.4.1 Testing Grounds

There is no use to optimizing if we cannot prove that our changes are in fact an optimization to the system's performance. Therefore, before we can begin to improve the system, we must be able to measure the system's efficiency before and after our changes. For Redeye we previously used to profile its functions using Optick for efficient performance analysis and optimization; but, after learning to manage big-data visualization using Tableau, we decided to make our own approach with Redeye's internal profiling system that extracts function call timings into a json file. It is much simpler compared to Optick, but because it is ours, we can modify it comfortably any way we see fit.



```
struct ProfilingOperation
{
    RE_ProfiledFunc function;
    RE_ProfiledClass context;
    unsigned long long start; // ticks
    unsigned long long duration; // ticks
    unsigned long frame;

#if defined(PARTICLE_PHYSICS_TEST) || defined(PARTICLE_RENDER_TEST)

    unsigned int p_count = 0u;

#ifdef PARTICLE_PHYSICS_TEST

    unsigned int p_col_internal = 0u;
    unsigned int p_col_boundary = 0u;

#elif defined(PARTICLE_RENDER_TEST)
```

Figure 37: Code Snippet – Profiling Operation

In the previous snippet, we can see the use of C++ macros. These allow us to configure code before compiling to not include profiling methods in a clean published release. The start value of each register uses high precision 8-byte long timers to measure accurately the duration of each operation. Registers vary in the number of fields depending on the type of profiling test we wish to conduct. Particle performance tests are divided into physics and rendering. Because rendering involves sending data to the GPU, the process takes longer than just accessing the systems RAM. Physics' testing therefore can be stressed more than its rendering processes and it makes sense to split them into different procedures.

### *Simulation Configurations*

The challenge with profiling physics comes when establishing the different setups to analyze. Depending on the spawning properties, simulations will have completely different results. To narrow down the number of setups and end up with the most relevant, particles have no lifetime limit, rendering deactivated and a circular spawn shape. We decided to split the simulations into categories by their:

1. **Particle inter-collisions:** can be enabled or disabled.
2. **Particle collider shape:** can be set as point or sphere.
3. **Boundaries:** have 3 different shapes: plane, sphere and AABB.

This leaves just 12 simulations to test over. The test's procedure was then procured to add a gameobject to the scene with a particle emitter component and have it run until the time it took to update the particles was higher than a thirtieth of a second. Having a simulation take the amount of time equal to a whole frame at 30 fps will take a toll on performance and render it useless. After setting on the condition to pass to the next simulation, we had to decide on the other properties for the collider (mass, restitution, and radius for spheres) and the boundaries.

The only inconvenient was that disabling inter-collisions allowed for much higher pool of particles before the time limit and took much longer. To alleviate the amount of time it took to profile the simulations, boundaries, spawn shape size and spawn frequency are increased for inter-collision-less ones. Later another set of configurations was set to separate between release and debug compilations as release took more than 10 times more per simulation. We register particle count and the number of collisions with the boundaries and between each particle to have a measure on the number of operations the updating goes through.

### *Physics' Module Update Configuration*

To reduce even more the testing time, the physics module was upgraded to have three different types of update call configurations: engine par, fixed update, and fixed time step. They can be changed in-engine through the module configuration tab.

```
default:
{
    update_count++;
    particles.Update(global_dt);
    break;
}
```

Figure 38: Code Snippet – Physics' Engine Par Update

Engine par uses the engines raw delta time to calculate each frame the simulations' state. Fixed update calls over a given period. If delta time increases past the fixed update, we add the current delta time with the previous accumulated offsets. Fixed update reduces the average amount of calls per frame but given a lagged framework will lose much accuracy. The third option of fixed timed steps now only updates a frame every given interval and if delta time where to double it, update would be called twice that frame. Having a fixed interval with unified delta time will improve result consistency.

```
case ModulePhysics::FIXED_UPDATE:
{
    dt_offset += global_dt;

    float final_dt = 0.f;
    while (dt_offset >= fixed_dt)
    {
        dt_offset -= fixed_dt;
        final_dt += fixed_dt;
    }

    if (final_dt > 0.f)
    {
        update_count++;
        particles.Update(final_dt);
    }

    break;
}
```

```
case ModulePhysics::FIXED_TIME_STEP:
{
    dt_offset += global_dt;

    while (dt_offset >= fixed_dt)
    {
        dt_offset -= fixed_dt;
        update_count++;
        particles.Update(fixed_dt);
    }

    break;
}
```

Figure 39: Code Snippet – Physics' Fixed Update (left)

Figure 40: Code Snippet – Physics' Fixed Time Step Update (right)

## First profiling

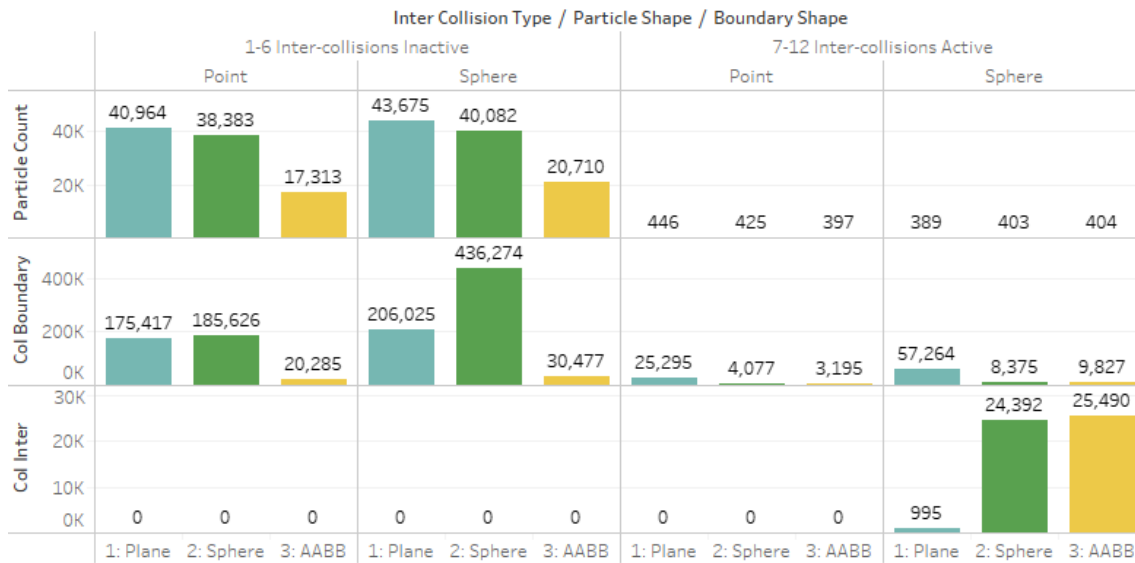


Table 6: First Particle Physics Test - Debug

The first iteration had terrible numbers as expected when compiling in debug mode. The graph features the number of boundary and internal collisions together with the number of particles it reached before taking too long to update.

The first observation we make is that removing inter-collisions will allow for a higher capacity of particles in the simulations. Point collider inter-collisions are nonexistent and just removing that iteration yields between 4 times more particles for AABB and 100 times for plane and sphere boundaries.

They all had the same gravity component; and, when impacting on a flat surface, bounced at the same spot countless times without changing direction. For active inter-collisions, only sphere particles will bounce off each other modifying the x and z position parameters. With the plane boundary, particles would bounce away from their origin as they collided and then end up too spread out to collide with each other again. Sphere and AABB boundaries contained the other 2 dimensions enclosing the simulation in less space and therefore causing almost on par the highest number of inter-collisions.

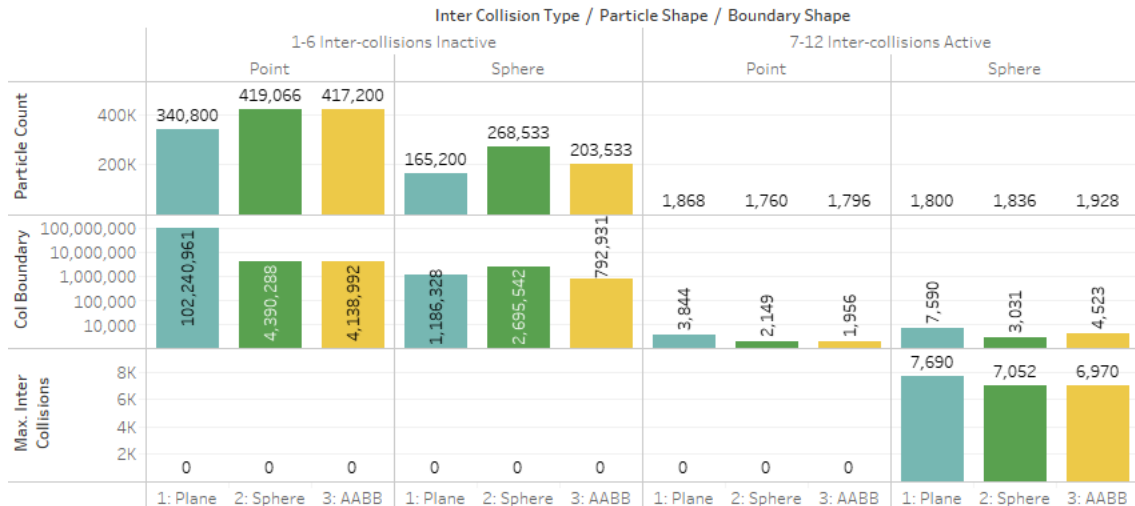


Table 7: First Particle Physics Test - Release

We expected performance to increase drastically in release mode, and it did not disappoint. We now have the maximum number of particles our system runs at a thirtieth of a second: 419K particles. This value compared to the previous 43K by sphere inter-collision-less particles over a plane is 10 times greater. Turns out that point particles in a sphere yields the highest outside our debugging environment.

## 5.4.2 Broad phasing

The engine's scene uses a dynamic axis-aligned bounding box tree to organize gameobjects in space through their bounding box. The new emitter component should update the gameobject's box. Simply changing the box values does not directly update its position in the dynamic tree for the gameobject has 2 AABBs: local and global. The event system is used to signal the scene that a gameobject's box has been altered and therefore must recalculate its position in the dynamic tree from the also recalculated global AABB. Module signaling uses events which are called during the input's Pre-Update function and in the application's remaining time between frames.

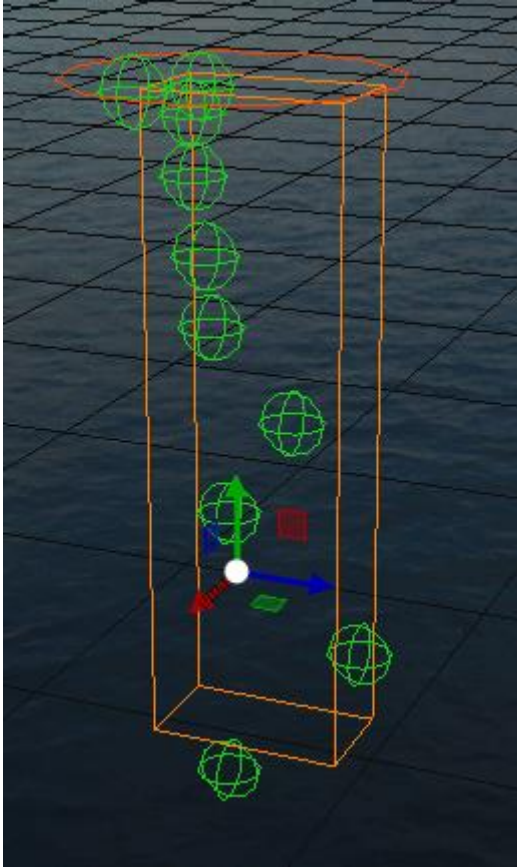


Figure 41: Simulation's AABB screenshot

For applying broad phasing paradigms, I had in mind segregating the particles inside the simulation to cull rendering and reduce the amount of collision checkups. But it proved to be more complicated than expected and lacking time to spare we had to reduce broad phasing to iterating with the objects in the scene. This optimization reduces the rendering time for out-of-sight particles effectively by 100%. This only improves the rendering pipeline as now our update cycle has extra calculations to enclose its particles.

Simulations without boundaries or with plane boundaries have the highest degree of particle freedom and end up spreading endlessly. Sphere boundaries on the other hand are easy to enclose into an AABB and its boundaries can just be copied without having to depend on the maximum distance or iterate through each particle.

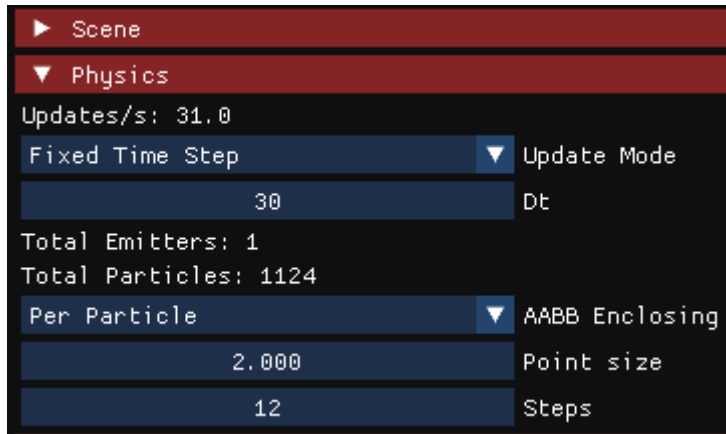


Figure 42: Application Configuration Window Screenshot - Physics

To extract the simulation's bounding box, we have 2 options available through the physics header in the editor's configuration window. The first one is iterating through the particles and saving the maximum and minimum values of each of the particles' positions. This adds extra stress on performance for each particle is checked every update. The other method is remarkably similar. It only uses the already calculated maximum distance used for rendering calculations to build an all-encompassing AABB. This AABB is marginally less accurate but only checks once per simulation. The option to change how the AABB is measured can be found in the physics section of the application's configuration window.

### 5.4.3 Data-Oriented Design

DOD is a programming paradigm with no definitive procedure of implementation. It takes into consideration the hardware's mechanical limits for memory management. We will concentrate on scoping the engine looking for possible CPU wait state triggers. Allocating data as contiguously as possible improves procedure performance. Because data is sent in chunks from RAM to CPU, these chunks may contain the next data structure to be processed. If it does coincide, we effectively free the CPU from having to requests data from another section in memory and stall until it is returned.

Code branching is an algorithm used by compilers to split code into different sections with the intention of optimizing code performance. Functions are stored in memory and must be loaded into the CPU to run. The same way we allocate data contiguously, we can also apply continuous memory usage to functions. Virtual functions and conditional "if" statements may trigger compiler branching.

Let us look at Redeye's math functions to find a perfect application of DOD. The `MinI` function returns the minimum value between 2 integers and causes no branching. First, the function is inline, which will make the compiler include its contents into calling functions. Next, we calculate the index of the correct value through a conditional expression. C's Boolean values can be interpreted as 1 for true and 0 for false; effectively choosing the value without the use of an "if" statement and assuring no code branching for these lines of code.

```
inline const int MinI(const int a, const int b)
{
    const int res[2] = { a, b };
    return res[b < a];
}
```

Figure 43: Code Snippet – RE\_Math's Minimum Integer function

After adding particle update to the emitter and setting emitter functions to inline declarations, we removed as many if statements with the previous method as we could. We then changed particle storage from a list of pointers to a contiguous vector without pointers. We used this opportunity to include using constant references for particle iterations that do not modify their attributes. This way, we do not need to copy the whole particle's data into another location.

```
for (const auto &p : simulation->particle_pool)
{
    // Calculate Particle Transform
    const math::float3 particleGlobalpos = simulation->local_space ? g...
    math::float3 front, right, up;
    switch (simulation->particleDir)
    {
        case RE_ParticleEmitter::PS_FromPS:
```

Figure 44: Code Snippet – Particle Manager's Draw Simulation Function



5.4.4 Profiling Analysis

Having applied the researched paradigms, and being confident in its results, we proceeded to replicate the profiling testing sessions and measured how much of a difference we really made. On first sight, the table reveals having an identical distribution as the previous graphs.

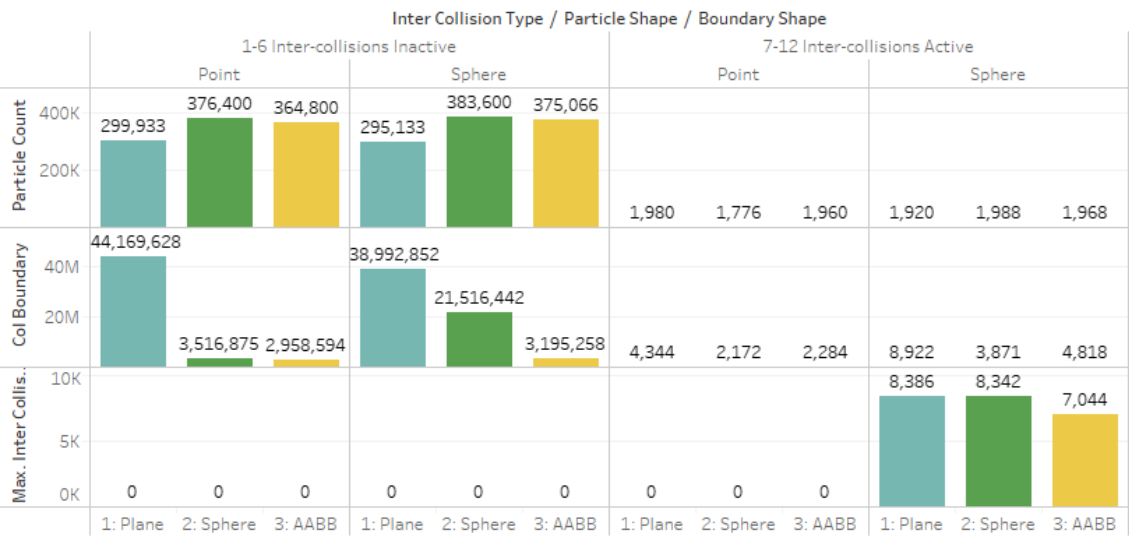


Table 8: DOD Particle Physics Test - Release

Now that we have a relatively large amount of simulation data, we can compare results and output them visually using our particle physics' Tableau solution which had be storing all the different simulation results. Looking up and evaluating the charts we can extract the difference of supported particles per configuration. The next table features this comparison throughout all 24 profiled simulations. It may seem overwhelming at first, but it is the same data from the previous charts organized in a compact way so it could fit inside the paper's margins. The rows classify simulations by their:

- **Compilation mode:** release or debug.
- **Collider shape:** point or sphere.
- **Boundary shape:** plane, sphere or AABB.
- **Code iteration:** default or DOD.

Code iteration is set to last to have the different particle counts side by side in default and DOD mode inside the same panel. Inter-collision split particle counts into columns where axis intervals can be independent from each other. Simulations with active inter-

collisions range a particle count from 400 up to 2K and having their own range aids the graphs' visual feedback compared to having them all run the same column. Finally, the Difference field values refers to the change in particle count from with to without DOD and showing green when positive and red otherwise.

			Inter Collision Type									
			1-6 Inter-collisions Inactive				7-12 Inter-collisions Active					
Mode / Particle Shape	Debug	Point	1: Plane	Default	<div><div></div></div>			<div><div></div></div>				
			DOD	<div><div></div></div> 47,693		<div><div></div></div> 6,729	<div><div></div></div> 527	<div><div></div></div> 81				
			2: Sphere	Default	<div><div></div></div>		<div><div></div></div>					
		DOD	<div><div></div></div> 42,499		<div><div></div></div> 4,116	<div><div></div></div> 566	<div><div></div></div> 141					
		3: AABB	Default	<div><div></div></div>		<div><div></div></div>						
		DOD	<div><div></div></div> 21,364		<div><div></div></div> 4,051	<div><div></div></div> 514	<div><div></div></div> 117					
	Sphere	1: Plane	Default	<div><div></div></div>		<div><div></div></div>						
		DOD	<div><div></div></div> 43,675	<div><div></div></div> -11,368	<div><div></div></div> 550	<div><div></div></div> 161						
		2: Sphere	Default	<div><div></div></div>		<div><div></div></div>						
		DOD	<div><div></div></div> 40,082	<div><div></div></div> -948	<div><div></div></div> 461	<div><div></div></div> 58						
		3: AABB	Default	<div><div></div></div>		<div><div></div></div>						
		DOD	<div><div></div></div> 20,710	<div><div></div></div> -1,731	<div><div></div></div> 529	<div><div></div></div> 125						
Release	Point	1: Plane	Default	<div><div></div></div> 340,800		<div><div></div></div>						
		DOD	<div><div></div></div>	<div><div></div></div> -40,867	<div><div></div></div> 1,980	<div><div></div></div> 112						
		2: Sphere	Default	<div><div></div></div> 419,066		<div><div></div></div>						
	DOD	<div><div></div></div>	<div><div></div></div> -42,666	<div><div></div></div> 1,776	<div><div></div></div> 16							
	3: AABB	Default	<div><div></div></div> 417,200		<div><div></div></div>							
	DOD	<div><div></div></div>	<div><div></div></div> -52,400	<div><div></div></div> 1,960	<div><div></div></div> 164							
	Sphere	1: Plane	Default	<div><div></div></div>		<div><div></div></div>						
		DOD	<div><div></div></div> 295,133	<div><div></div></div> 129,933	<div><div></div></div> 1,920	<div><div></div></div> 120						
		2: Sphere	Default	<div><div></div></div>		<div><div></div></div>						
DOD		<div><div></div></div> 383,600	<div><div></div></div> 115,067	<div><div></div></div> 1,988	<div><div></div></div> 152							
3: AABB		Default	<div><div></div></div>		<div><div></div></div>							
DOD		<div><div></div></div> 375,066	<div><div></div></div> 171,533	<div><div></div></div> 1,968	<div><div></div></div> 40							
			0K 500K	0K 200K	1K 2K 3K	0 100 200						
			Max. Particle Co..	Difference	Max. Particle Co..	Difference						

Table 9: Comparative Particle Count profiling results

The first observation we can extract is that DOD has improved all active inter-collision simulations. The differences are approximately a hundred more particles with DOD. Looking at the left side, we see that it has not been the case with the rest of simulations where the differences were more substantial.

Sphere colliders in debug mode worsened while the point colliders improved particle count. These settings have the completely opposite effect when running in release.

These results first left me in awe and had me doubting about any errors in the testing procedures. After re-taking them numerous times to account for inherent randomness in particle spawning, their consistency in not reaching higher than non-DOD particles assured that the tests were indeed a valid reflection of the system's performance.

Point particles in debug mode increased by 4K and the plane got up by 6K, but the same particles in release took the highest toll. We lost our high score values by 40K reducing the 400K limit. Sphere inter-collision-less colliders had the reverse effect improving in release instead of debug mode. Here they rose by 115K and by 171K for AABB boundaries whilst declining by 11K for the plane boundary and around 1K for the other.

So, if applying the DOD paradigm were to improve our performance, how was it that it ended being counterproductive for some simulations? We did complicate the allocation procedures for particle instantiation. With list containers we have no capacity limits and adding new particles was as easy as pushing the particle's location into the list's last linked node. With vectors, their capacity range had to be updated if overburdened. Because resizing the container was very costly, we divided the necessary allocations to just 10 for each simulation based on their pre-defined maximum particle count every time they were about to exceed this capacity.

```
const unsigned int to_add = RE_Math::MinUI(
    spawn_mode.CountNewParticles(local_dt),
    max_particles - particle_count);

particle_count += to_add;
if (particle_pool.capacity() < particle_count)
{
    const unsigned int allocation_step = max_particles / 10u;
    unsigned int desired_capacity = allocation_step;

    while (particle_count < desired_capacity)
        desired_capacity += allocation_step;

    particle_pool.reserve(desired_capacity);
}
```

Figure 45: Code Snippet – Particle spawning's memory allocation

All simulations had the same capped maximum particles at half a million because the previous profiling results had never reached this limit before. This sets resizing procedures to proc every 100K particles. If the tests were to reach this limit, we would have had to change it and retake the tests. We ended up not having to make this change as the results had reduced the maximum particles supported for our highest scoring simulations.

Because the simulations' spawning frequency for inter-collision-less particles was set to 980Hz, the physics module's fixed timed step forces around 33 new particles every update call. For the container resizing procedure to cause enough lag for the update to last more than 33 milliseconds, simulation results would have ended at particle counts near 100K multiples. We do take into consideration this stress in performance, but it does not account to any results as each had numbers far from the multiples.

The only other factor that worsened performance was the added broad-phasing procedures for particle envelopment. The physics module was configured with the otherwise high precision AABB enclosure to stress extra calculations per particle. Even so, only inter-collision-less point collider configurations worsened for the release mode.

We are slightly bothered that our 400K maximum particles record was lost, but we are convinced that DOD does improve engine performance. The testing grounds set for particle physics only had this sole gameobject in the middle of the scene with a single huge simulation. The scene's composition is not a realistic portrayal of the environments for videogame scenes. Having only the simulation to compute, nothing other than particles were being allocated in memory. Our list containers therefore had memory allocated in a continuous way. That is the reason why lists had improved performance during these closed environments. If we were to conduct simulation testing in a videogame-like scene with objects moving, disappearing, and appearing constantly, list containers would end up with their data completely spread throughout the memory and DOD would shine through.

Simulations run fast enough in our own opinion. 383K is a remarkably high number for maximum particles. If we ever want to increase our cap, we still have different options to improve results. Particles' structure stores all values they could need; but, depending on the simulation's properties, they may end up not being used. They then only occupy memory wastefully. We could adapt the particle's memory to the simulation and increase the yield. Having specialized particle structures, we could also extend this feature to the update procedure and extract different functions instead of the current all-enclosing one we have right now. If the function does not have to check which procedures to go through, we reduce its processing load.

#### 5.4.5 Progress Iteration

The optimizing stage took a week longer than expected. Specifically, the profiling task where we had to build the testing grounds. All the accumulated previous issues and bugs started to show when forcing the engine's limits. Tests run the engine with the simulations and automatically change from one simulation to another once they take

too long to update. If an error triggered, we had to debug the error and go back to retaking the previous simulations because we had changed the code.

Deciding on the emitter's settings for testing ended being more tedious than implementing the procedures to run and output a file per each simulation. Some tests took too long, while others finished almost instantly, and we had to tweak each configuration through trial and error. We knew that we had to be incredibly careful with how these tests where to run as they had to be unchanged for the subsequent tests to be comparable.

Having applied some types of optimizations, the particle system is now ready for deployment. Other optimizations discussed like multi-threading and compute shaders could improve our yield; but unfortunately, this projects' scope had to be cut down to our available time space. Looking at the semi-final version of the Gantt, we see we have less than a week left to proceed with publishing a new release.

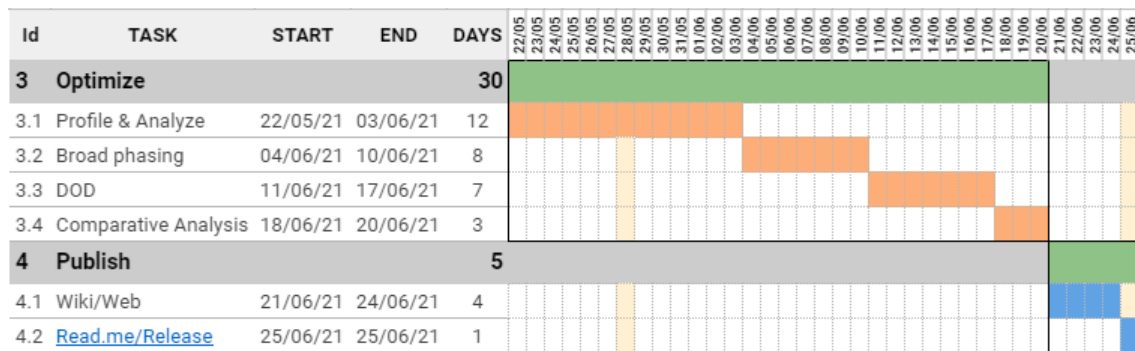


Figure 46: Adapted Gantt v3 – Optimizing & Publishing.

## 5.5 Publishing

Publishing code is part of developing it. Redeye engine is open-sourced meaning that its contents are online available for anyone who might be interested. The engine has a total of 144 code files with our own structures and procedures apart from the code from all the different libraries we use. If we were to plainly upload the code, it would be incredibly challenging for anyone understand how it works. To ease this process, publishing redeye compromises of a GitHub release, updated wiki and the engine's own website which also runs through the GitHub's repository.

### 5.5.1 Release

GitHub releases are perfect way to showcase the engine and portray our milestones. With a clean code and finished implementation of the particle system, we can publish the current state of the engine. Together with each release we list the changes the engine has foregone.

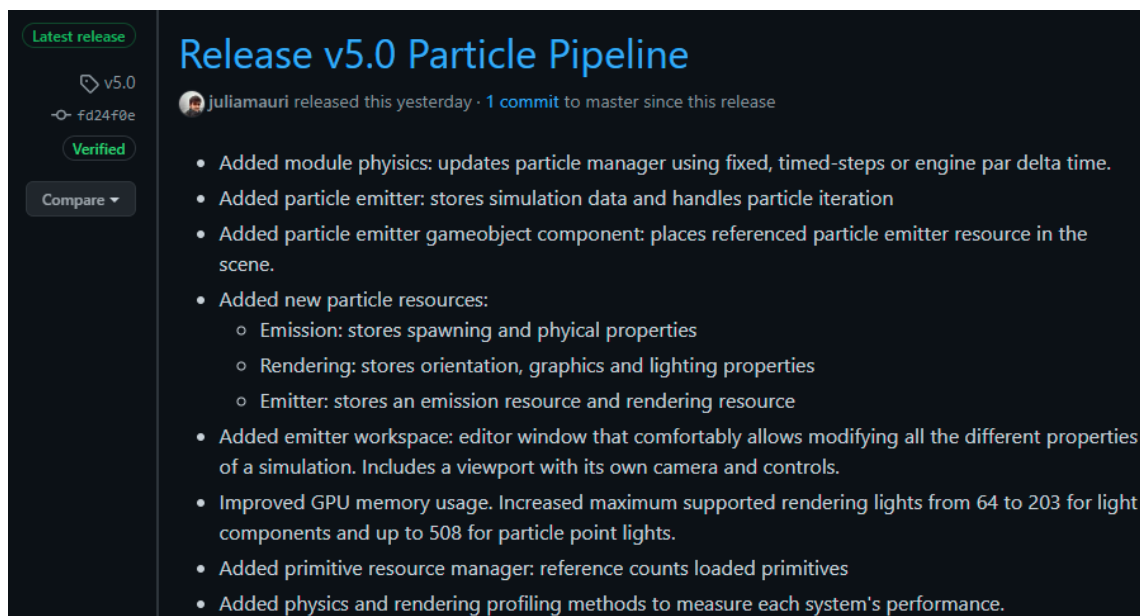


Figure 47: Release v5.0 from Github's repository

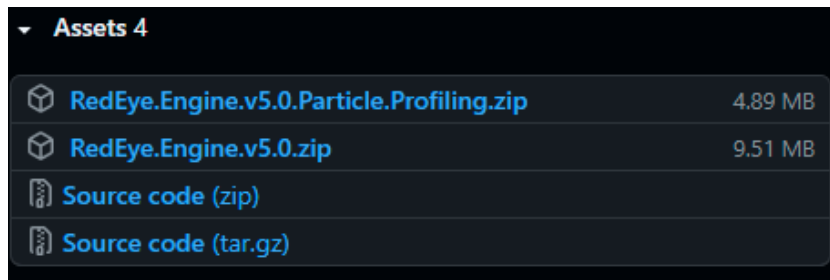


Figure 48: Release Assets

After the latest changes, we find the available downloadable resources packed into 4 different assets. Two of them are different compressed folders containing all the repositories files automatically offered by GitHub. The other two are compressed files procured by us.

**RedEye.Engine.v5.0.Particle.Profiling.zip** contains:

- Particle Physics and Particle Rendering folders containing all the earlier profiled simulation results.
- The 3 different Tableau solutions used to analyze the profiling results.
- Both versions of redeye's executables with the profiling procedures. Users can replicate our testing environment and check whether their hardware is able to yield more particles.
- An empty file used to guide the user to move the executables to their correct environment in the other release asset as we do not include engine files here and the executables will not be able to run.

Name	Type	Size
Particle Physics	File folder	
Particle Rendering	File folder	
Engine Profiling.twb	Libro de trabajo d...	159 KB
Particle Physics Profiling.twb	Libro de trabajo d...	159 KB
Particle Rendering Profiling.twb	Libro de trabajo d...	115 KB
PLEASE MOVE EXECUTABLES TO ENGINE RELEASE	File	0 KB
RedEye Particle Physics Profiling.exe	Application	3,050 KB
RedEye Particle Render Profiling.exe	Application	3,050 KB

Figure 49: Release's RedEye.Engine.v5.0.Particle.Profiling.zip contents

The other asset we include with our release is **RedEye.Engine.v5.0.zip** and it contains:

- Redeye's **executable** file that runs the engine without profiling procedures.
- **Dynamic Linked Library** (DLL) files from the libraries used. If they were to not be dynamically linked, compiling Redeye engine would take several minutes every time we made changes in code.
- **Imgui.ini** file storing the editor's window configurations. If a user where to move editor windows around, their changes will stay the next time they run the engine.
- **GeneratedSoundBanks** folder stores audios for the audio module which we had to clear for we do not own the rights for the samples we were using.
- **README.md** file containing all the necessary information needed to use the engine. It also serves as the repository's welcome page contents. It features:
  - A brief synopsis.
  - Links to our profiles, our tutors' profiles, university, and engine license.
  - Latest and previous version notes.
  - Available user actions and editor window descriptions.
  - Brief notes on of some of Redeye's internal systems
  - List of libraries used.















Name	Type	Compressed size
 GeneratedSoundBanks	File folder	
 assimp-vc142-mt.dll	Application extension	1,676 KB
 data.zip	Archivo WinRAR ZIP	5,161 KB
 DevIL.dll	Application extension	651 KB
 glew32.dll	Application extension	110 KB
 ILU.dll	Application extension	54 KB
 ILUT.dll	Application extension	14 KB
 imgui.ini	Configuration settings	2 KB
 OptickCore.dll	Application extension	62 KB
 physfs.dll	Application extension	58 KB
 README.md	MD File	8 KB
 RedEye.exe	Application	1,454 KB
 SDL2.dll	Application extension	403 KB
 zip.dll	Application extension	85 KB

Figure 50: Release's RedEye.Engine.v5.0.zip contents



### 5.5.2 Wiki

Code documentation may adopt many forms. If we downloaded a library to use in our project, we would expect a full-on dictionary of every function with details about their parameters and intricacies. Redeye is not a library luckily, but we still want to give our users as much information as possible to ease understanding how Redeye is meant to be used. Our wiki starts with a welcome page through which we can navigate to different pages and explain the engine through its different systems. We do re-use content inside the readme file such as the release version notes, the libraries, us authors and application systems. It is in the applications explanation that we dig deeper into how exactly each one operates.

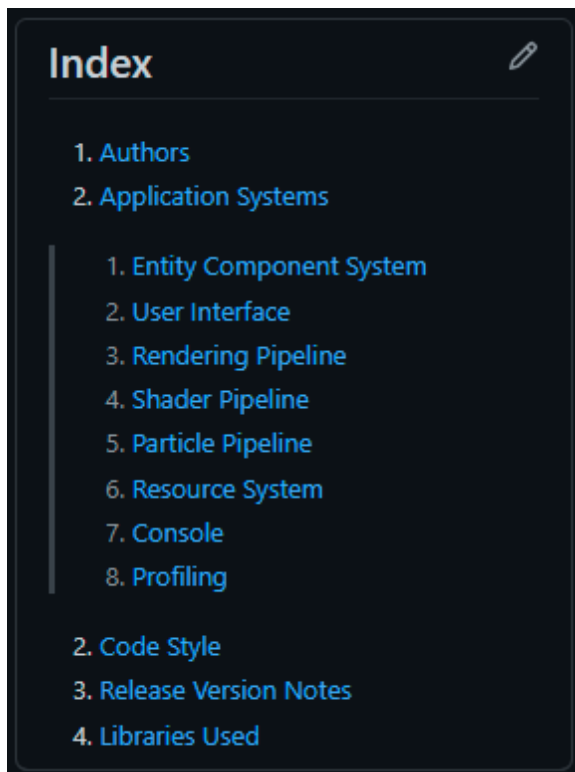


Figure 51: GitHub Wiki Index list

### 5.5.3 Website

Having a website for the engine makes it feel like a more polished product. Using some of the wiki's contents, we can export them to the engine's website. Here the way we present the contents can be edited to fit our own style. Web design is not our strong

suit, but we were able to make some scripts that make the site more unique and customized. We use the engine's repository directly and use GitHub services to show the html files from the repository onto our domain. The sites contents include:

- Homepage featuring relevant engine information.
- The Team page referencing its authors.
- Engine Guts with specific system explanations.

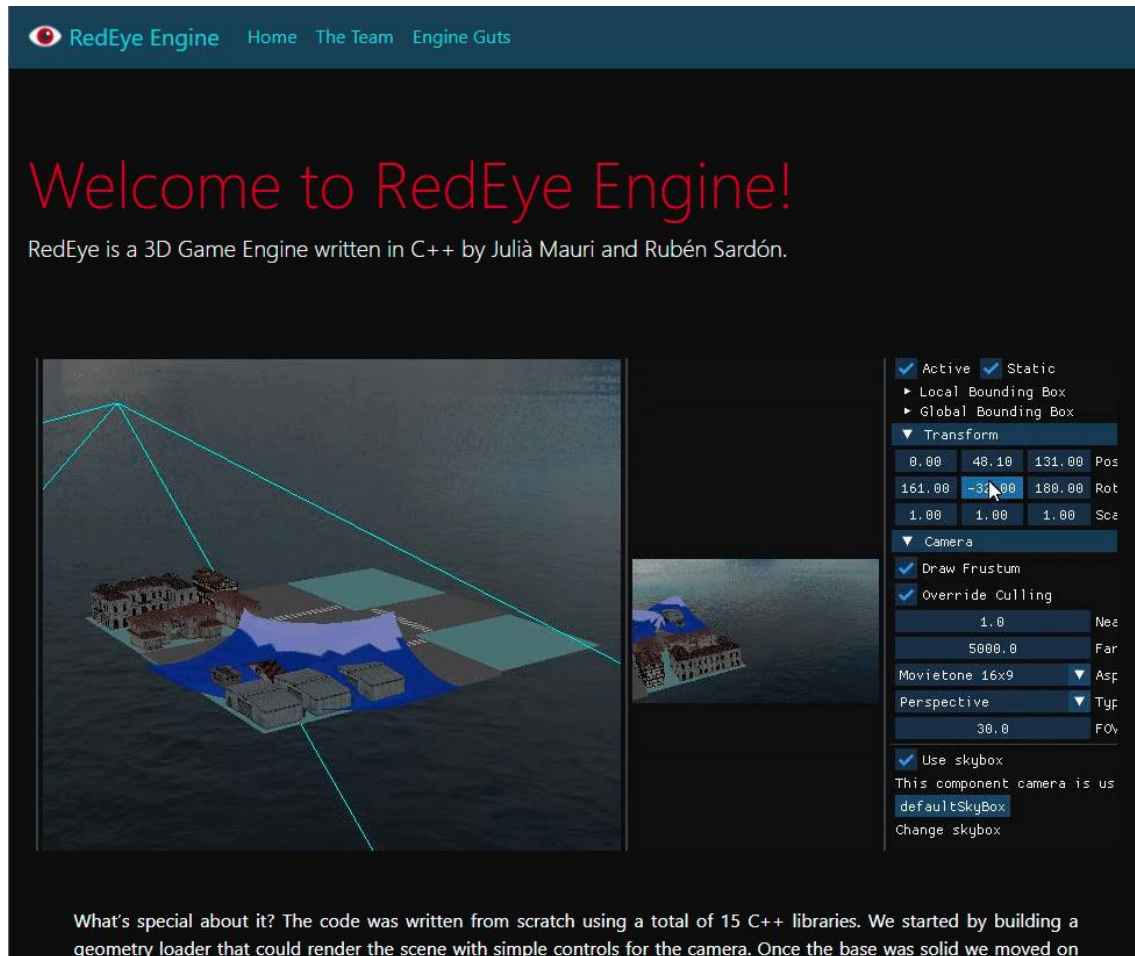


Figure 52: Redeye's website home page

## 6. Conclusions

Once the engine's release and resources had been published, we finally completed our ordeal. In just 3 months, we were able to improve Redeye with a polished particle pipeline. Adding particle physics deemed more challenging than expected, but we ended up with an excellent tool to edit and run particle simulations. Particle emitters have numerous customizable fields that change the emission's behaviors. Physics, spawning, and rendering attributes can be changed through the workspace editor window and the physics module counts with delta time configurations, particle enclosing methods and their debug rendering properties.

These components run through the engines systems to allow reference counting, resource serialization and scene's broad-phasing camera culling. Particle iterations have been tested, profiled, and optimized for improved particle yield and its release is published with a complementary updated wiki and website. Everyone can access our open-sourced engine's code and apply any part of it they wish into their own projects.

Through this project's development, we have improved our programming and profiling analysis skills. The engine is looking better with every new feature we add. There are no wrong answers on how to implement particles, but our pipeline clearly feels like an adequate way. Implementing broad-phasing and DOD paradigms improved most simulation performances shown through the profiling analysis which presents a generic overview of the system, but it all depends on its final usage.

We even tested the differences between debug and release modes. End-users might not care for debug because it only affects us developers. Still, having knowledge about debug and release performance variance gives us a better idea of how the compilation modes will vary for future added features that may as well handle high amounts of data.

Adaptability has been key during this teamwork exercise. Being two programmers changing the same pipeline, we stepped in each other's lines of code all the time. No matter how much we tried to impede the situation, we kept bumping in. We ended working mostly side-by-side and specially during the first and last stages of the project.

Starting from a crude particle implementation, we were able to carry out a complete workspace dedicated to editing all the different customizable fields. We managed to reach all objectives and half the propounded optimization paradigms from our initial planification. We did however encounter with tasks taking more time than expected and ended up with our actual costs deferring from our initial statement. From the 120 hours we expected the particles to require, we ended up almost doubling this time to 220 hours averaging 2.5 hours a day during the whole 3 months. This time increase sets our total costs to a whopping 5734 € from the initial budget of 3334 €.

## 7. Future Projects

We intend to continue working on our engine and keep adding features we deem interesting. Our highest priorities are completing the particle pipeline with the systems we were not able to implement:

- Add attribute dependencies for particles.
- Implement gizmo support for particle spawner geometry.
- Add other types of shapes for colliders, emission areas, and boundaries.
- Add particle angular rotation support.
- Support adaptive memory size for particle containers.
- Apply broad phasing structures inside simulations.
- Apply multi-threading.
- Add compute shader support for hardware accelerated physics calcs.

The physics module was designed with the intention to later add rigidbody support. The collider structure used by particles can be refactored to include its use outside of particles. Having the same collider structure will help the physics module run as a whole and make particles interact with the scene's colliders.

Apart from the particle systems improvements, we also envisioned different projects before deciding to make particles. Their nuance got our attention, but splitting the workload was significantly easier with the particle pipeline. Raytracing technologies for example have recently experienced a boom in popularity with graphic card distributors boasting about how their cards are aimed at handling raytracing software. Following the rendering trend, we also considered what we refer to as “multi-rendering”. It consists of changing out rendering module into a more general structure and adding support for more rendering APIs such as DirectX or Vulkan. The last future project we have on hold is the scripting system. There are many different approaches, and we believe the most fun to develop is Visual Scripting using nodes to handle component behaviors.

## 8. Bibliography

### 8.1 Citations

- Acton, M. (2021, March 10). *A Data Oriented Approach to Using Component Systems*. Lecture presented at Unity at GDC, San Francisco. <https://youtu.be/p65Yt20pw0g>
- Connell, T. (2021, March 10). *Intro to Game Physics*. Lecture presented at DigiPen Game Engine Architecture Club. <https://youtu.be/wPKzwSxyhTI>
- Middleditch, S. (2021, March 10). *Data-Oriented Design*. Lecture presented at DigiPen Game Engine Architecture Club. <https://youtu.be/16ZF9XqkfRY>
- Nikolov, S. (2021, March 10). *OOP Is Dead, Long Live Data-oriented Design*. Lecture presented at CppCon. <https://youtu.be/yy8jQgmhbAU>
- Serrano, H. (2016, May 04). Visualizing the runge-kutta method. Retrieved March 10, 2021, from <https://www.haroldserrano.com/blog/visualizing-the-runge-kutta-method>

### 8.2 Complementary Research Sources

- Baumel, E. (2021, March 5). *Understanding data-oriented design for entity component systems*. Lecture presented at Unity at GDC 2019. [https://youtu.be/0\\_Byw9UMn9g](https://youtu.be/0_Byw9UMn9g)
- Middleditch, S. (2021, March 5). *A Brief Introduction to OpenGL*. Lecture presented at DigiPen Game Engine Architecture Club. <https://youtu.be/IXxc9yNBpuo>
- Connell, T. (2021, March 5). *Code Architecture*. Lecture presented at DigiPen Game Engine Architecture Club. <https://youtu.be/SsDhk9i-q-w>
- The Chernobyl (Director). (2021, January 13). *3D physics! // hazel engine dev log* [Video file]. Retrieved March 5, 2021, from <https://youtu.be/oqTAHwwSA9I>
- Melegari, B. (2014, November). *Physics Simulation and Video Games*. Retrieved March 5, 2021, from [http://ffden-2.phys.uaf.edu/webproj/211\\_fall\\_2014/Bryce\\_Melegari/intro.html](http://ffden-2.phys.uaf.edu/webproj/211_fall_2014/Bryce_Melegari/intro.html)
- *Rendering Particles with Visual Effects Graph in Unity!* [Video file]. (2020, August 19). Retrieved March 6, 2021, from <https://youtu.be/pNYZZk5h3IM>
- *Video game explosions are actually a lie* [Video file]. (2019, October 7). Retrieved March 6, 2021, from <https://youtu.be/pNYZZk5h3IM>
- *Multi Layered Effects with Visual Effect Graph in Unity! (Tutorial)* [Video file]. (2020, August 7). Retrieved March 10, 2021, from <https://youtu.be/UybzSIUlc0>

- Caston, N. (Director). (2019, December 18). *Unity DOTS vs Handbuilt: Sample Project* [Video file]. Retrieved March 10, 2021, from <https://youtu.be/tInaI3pU19Y>
- Game Dev Guide (Director). (2020, December 31). *Getting Started with Compute Shaders in Unity* [Video file]. Retrieved March 10, 2021, from <https://youtu.be/BrZ4pWwkpto>
- Java - Encapsulation. (n.d.). Retrieved March 11, 2021, from [https://www.tutorialspoint.com/java/java\\_encapsulation.htm](https://www.tutorialspoint.com/java/java_encapsulation.htm)
- Goldberg, D. (2019, June 24). Data-Driven design: What it is and why it matters. Retrieved March 10, 2021, from <https://www.springboard.com/blog/data-driven-design/>
- Chou, A. (2014, September 15). Game physics: Broadphase – DYNAMIC aabb Tree: MING-LUN "Allen" CHOU: 周明倫. Retrieved March 10, 2021, from <http://allenchou.net/2014/02/game-physics-broadphase-dynamic-aabb-tree/>
- Unity Technologies. *Introduction to Particle Systems*. 12 Feb. 2021, <https://learn.unity.com/tutorial/introduction-to-particle-systems>
- Unity Documentation. (2021, April 30). `MonoBehaviour.FixedUpdate()`. Retrieved April 30, 2021, from <https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html>
- Smid, T. (n.d.). *Elastic and Inelastic Collision in Three Dimensions*. Plasma Physics. <https://www.plasmaphysics.org.uk/collision3d.htm>
- MDN. (2021, April 24). *3D collision detection*. MDN Web Docs. [https://developer.mozilla.org/en-US/docs/Games/Techniques/3D\\_collision\\_detection](https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_collision_detection)
- Baker, M. J. (2020). *Physics - Collision in 3 dimensions*. Euclidean Space. <https://www.euclideanspace.com/physics/dynamics/collision/threed/index.htm>
- Goodstein, D. L. (2020, August 21). *Motion of a particle in two or more dimensions*. Britannica. <https://www.britannica.com/science/mechanics/Motion-of-a-particle-in-two-or-more-dimensions>
- *Mechanics: Momentum and Collisions*. The Physics Classroom. (2021). <https://www.physicsclassroom.com/calcpad/momentum>
- Toft, A. (2012, April 12). *Implementing 3D collision resolution*. The Physics Classroom. <https://www.atoft.dev/posts/2020/04/12/implementing-3d-collision-resolution/>
- Hecker, C. (2007, May 14). *Rigid Body Dynamics*. Chris Hecker. [http://www.chrishecker.com/Rigid\\_Body\\_Dynamics](http://www.chrishecker.com/Rigid_Body_Dynamics)
- Rotenberg, S. (2018). *Collision Detection*. UC San Diego. [https://cseweb.ucsd.edu/classes/wi18/cse169-a/slides/CSE169\\_12.pdf](https://cseweb.ucsd.edu/classes/wi18/cse169-a/slides/CSE169_12.pdf)
- Gomez, M. (1999, October 18). *Simple Intersection Tests For Games*. Gamasutra. [https://www.gamasutra.com/view/feature/3383/simple\\_intersection\\_tests\\_for\\_games.php](https://www.gamasutra.com/view/feature/3383/simple_intersection_tests_for_games.php)