UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Centre de la Imatge i la Tecnologia Multimèdia

RedEye's 3D Particle Rendering

Final Degree Project

Videogame Design & Development Degree

Author: Mauri Costa, Julià

Plan: 2014

Director: Löpfe, Lasse

Index

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Centre de la Imatge i la Tecnologia Multimèdia

Synopsis

This paper describes the process of adding a 3D particle render pipeline into Redeye Engine: a C++ game engine software developed by Rubén Sardón and me, who is author to the complementary project of developing its physics pipeline. It counts with a particle editor, that uses circumvolving RedEye Engine systems, and improved graphic memory management.

Key words

3D, Open source, Simulation, Rendering, C++, Particles, RedEye, Engine, GPU memory, GPU technology, OpenGL, ImGui, Particle editor, Resources, Particle pipeline, Profiling, Particle render

Links

Web: https://www.redeye-engine.es/

Github Repo: https://github.com/juliamauri/RedEye-Engine

Latest Release: v3.3:

Table Index

Figure Index

Equation Index

Glossary

Videogame engine: the tool to create videogames.

Particle: a mesh to render when normally are small and that can be smoke, water, dust…

Particle system: the system that handles all the particles.

Graphic programming/engineering: the code that represents the display visualization.

Rende system: the system that draws the final picture.

Resource system: the system that handles all the resources that are using on the project. It can be vertex from objects, textures, entities component system…

Shader system: the system that interpret and compiles code dedicated to draw pictures.

Physics pipeline: the system that handles all the physics calculations.

Deferred lighting: a draw method which uses a light system.

Profiling: measure and present the duration between process.

Alpha: is the opacity, transparency from some object.

Textures (albedo / diffuse / specular): the image that defines how looks an object.

Hot-reloading: a process that watch some changes and import automatically.

Unity: a videogame engine.

GLSL: a graphic programming language.

C++: a low-level programming language.

Mesh: object that contains vertex for drawing.

Billboard: a method calculation that makes the object points to the camera.

Material: a structure where is variables that defines how looks some object.

GPU: graphics process unit, which calculates all the draws.

CPU: central processing unit, which calculates all the programs operations.

Multithreading: runs parallel code.

Compute shader: a technology like multithreading for GPU.

Scripting: a low-level code that interprets some functionalities.

Unreal Engine: a videogame engine.

Atlas: a single texture that contains tons of textures.

Interface Block: the name of a structure that contains variables on GLSL.

SSBO: shader storage buffer object, an interface block with a high capacity of memory allocation.

API: application programming interface, programs that are made for using at another programs.

1. Introduction

1.1. Motivation

After researching and studying, I discovered different game development paths and I keep learning new ways to improve engine systems. Because of this, I found out programming is my preference and I really enjoy coding.

At the 3rd grade of my career, at the videogame engine subject, I learned out how an engine works and managed to implement my own version. After the subject, with my teammate, Ruben Sardón, we have continued working on our engine, named ''RedEye'' Engine. I did almost all the graphics part, looking into the most specific graphic engineering intricacies, and other systems like the resource system.

We decided to write a TFG about how to implement a particle system in a videogame engine, testing it out with our engine. My work includes the render part because I focused my work on implement the resources and shader systems. Meanwhile, Ruben has been focusing on the physics side of the problem.

Furthermore, I am still working on my engine because I love to work on it and feature it as part of my portfolio.

1.2. Problem Formulation

At present, we are implementing a shader system inside our engine, so the users can create their custom shader and work with it in real-time, showing the results immediately when they compile it, also they can manage their scene, import their own resources, etc. Our challenge is to implement a particle system, in the case our user wanted to work with particles, thus improving the engine. The system will be able to display on monitor different simulations of particles.

We are going to focus on the graphic part. How can we draw all those elements? How many particles can we draw without altering the performance? One million particles? How to implement it on a deferred light system? Our engine has implemented a deferred light system, that can be used to make a particle emit light. The challenge with deferred lighting lays on its use of opacity. These alpha values depend on the depth buffer which RedEye's deferred rendering pipeline at the moment does not handle.

Nowadays, this problem is already solved by top tier videogame engines like Unity and Unreal Engine who use their own integrated implementation or by low level libraries like Thor C++ that can be integrated through developers' own approach. We are going to study all the current particle systems and do our own design using the best elements that we see fit together with user experience tests to implement a usable user interface to working with our resulting pipeline.

1.3. General Goals

The main goal of this project is to learn how to work in parallel with the other part, RedEye's 3D Particle Physics by Ruben Sardón, and piloting all the concepts learned at the grade, principally the programming and user experience subjects. There are four general goals getting more into the project:

1. Creating a **particle rendering pipeline** on RedEye Engine. We need to adapt our current pipeline to so it can draw particles.

2. Implement the **resource particle emitter** with corresponding engine usability. Try adding the shader support to the new pipeline. Which will allow the users to be able to code their custom shader for rendering their custom particles, also we will design a simple and useful UI to create particles and it will be also available as a shader script, to start working with a template.

3. **Profile the pipeline** to show our advancements with optimization, like the quantity of particles that the engine can draw at the same time.

4. Publishing release and work on GitHub wiki and the **engine's webpage** to showcase all its features.

1.4. Specific Goals

For the first goal, we start by adding the particle render into our pipeline rendering, that means we need to adapt it to a normal render and the deferred light render (without alpha). Next comes adapting the particle system to our material resource, the particles will be drawn with the proprieties that contain the material, such as diffuse and specular color, the possibility to use a texture, and the custom values that the user defined to their own shader (mentioned to next goals).

We then proceed to focus on creating some main tools for building particles because most users do not know how to code shade. For example, change the color during the time/distance/time of life, use a set of random colors, etc. We believe that during the development maybe some new tools could be added while implementing features, mentioned next up in the Project Scope where we explain some features that we would implement if we do not spend as much time doing the main goals.

Moreover, we will try adding support for working with particles directly into a shader, using the basic shader, which we will create for previous goals, as a template. Also, we will do profile tests during all the developments for show the advancements and then we will be doing several tests for show the optimizations implemented.

When all these functionalities are implemented, comes the resource implementation with their own management like deletion.

Finally, when we deploy final release, we are going to work on the webpage, updating all the content with media.

1.5. Project Scope

This project does not cover alpha on deferred light implementation due to the limited time frame. Another interesting feature reserve for future project is compute shader, a technology that we transfer all the information to the graphic memory, it means that the users can draw millions of particles on the scene, with less control. Furthermore, there is the possibility to use post visual effects (VFX) on particles and creation of particles using a visual scripting as a graph with nodes.

The main obstacle is communicating from the processor to the graphic card, there is a lack of optimization that costs time to read things back from the video card. It´s interesting to handle all that information without accessing it on the video card for keeping all the optimization.

The target of this product are the designers that want to create particles, with the option to do a customizable particle coding shader, to our engine. Moreover, the developers that want to implement their particle system, can follow this project.

2. State of the Art

Includes studies and research on the render part of different particles systems and optimizations.

2.1. Market Study

2.1.1. Unity

Unity version 2012.1 is an engine for developing games, which contains their user interface that the developers can create their games as they want.

There are two particle systems, the Built-in particle system and Visual effect graph. The first one is the basic and limited particle system, which works with their entity component system with script interaction. On the other hand, the Visual effect graph is a recent particle system, which uses graphs to create particles and works directly at GPU, means that it is possible to use frame buffers and draw millions of particles. (*Unity - Manual: Choosing Your Particle System Solution*, n.d.)

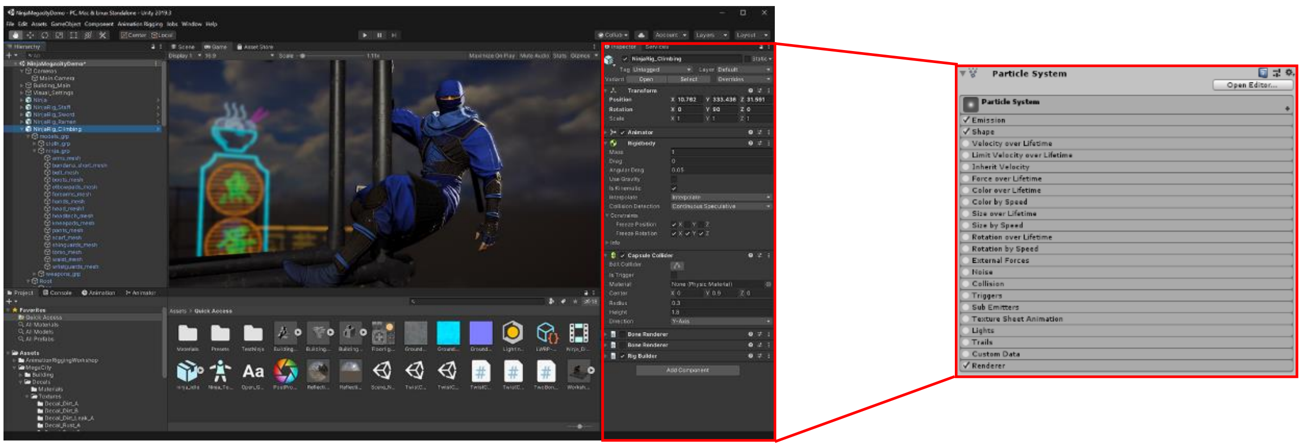
On the one hand, if we compare Unity with RedEye Engine, our render pipeline works directly with frame buffers, it means that our particle system can work with frame buffer from the early development. Also, our shader script is GLSL with support to material, hot-reloading, provide the content of some values if the user defines it on shader script and our material adapts to the shader if there are custom variables.

On the other hand, Unity uses its custom shader script language, which makes it more difficult to port GLSL script to their code. Although, we need to keep in mind that Unity supports almost all the platforms and RedEye Engine only Windows (we tried hardly to run it on Linux but currently the process is in pause).

The next sections we explain more accurate how their particles work.

2.1.1.1. Built-in Particle System

Our focus is on the inspector panel, which is located at right side of the editor and display the component values, in our case we explain the particle system module. Also, when a game object with a particle system is selected, on the editor appears a new little window for playback and watch the results immediately. (*Unity - Manual: Particle System*, n.d.)



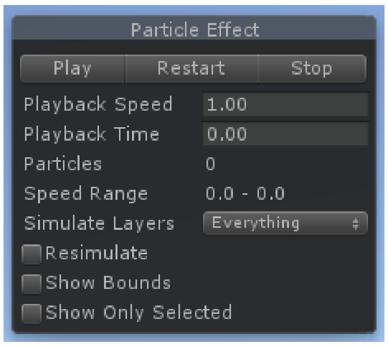
Particle System
Open Editor...

Particle System

- ✓ Emission
- ✓ Shape
- Velocity over Lifetime
- Limit Velocity over Lifetime
- Inherit Velocity
- Force over Lifetime
- Color over Lifetime
- Color by Speed
- Size over Lifetime
- Size by Speed
- Rotation over Lifetime
- Rotation by Speed
- External Forces
- Noise
- Collision
- Triggers
- Sub Emitters
- Texture Sheet Animation
- Lights
- Trails
- Custom Data
- ✓ Renderer

Figure 1: Left: Unity Edior user interface. RIght: Particle System component

On the particle main module (*Unity - Manual: Particle System Main Module*, n.d.) there is a variable Start Color (*Unity - Manual: Using the Built-in Particle System*, n.d.), it can be single color and gradient, which start and end color and works with the color over lifetime or by speed. Also, it can be a random of two choices or completely a random color if it is single.

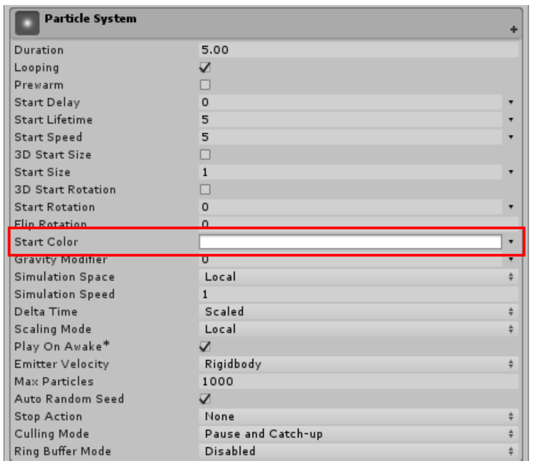


Particle Effect

Play | Restart | Stop

Playback Speed	1.00
Playback Time	0.00
Particles	0
Speed Range	0.0 - 0.0
Simulate Layers	Everything
☐ Resimulate	
☐ Show Bounds	
☐ Show Only Selected	

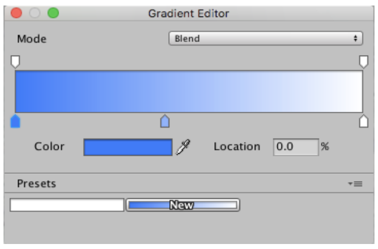
Figure 2: Particle playback controller



Particle System

Duration	5.00
Looping	✓
Prewarm	☐
Start Delay	0
Start Lifetime	5
Start Speed	5
3D Start Size	☐
Start Size	1
3D Start Rotation	☐
Start Rotation	0
Flip Rotation	0
Start Color	⬜
Gravity Modifier	0
Simulation Space	Local
Simulation Speed	1
Delta Time	Scaled
Scaling Mode	Local
Play On Awake*	✓
Emitter Velocity	Rigidbody
Max Particles	1000
Auto Random Seed	✓
Stop Action	None
Culling Mode	Pause and Catch-up
Ring Buffer Mode	Disabled

Figure 3: Particle System main module



Gradient Editor

Mode: Blend

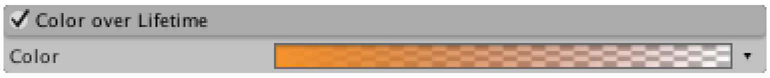
Color: 🟦 Location: 0.0 %

Presets

Figure 4: Gradient Editor

Seeking every particle module, we focus more into the three next modules:

- Color Over Lifetime changes the color by using a gradient over the lifetime particle. (*Unity - Manual: Color Over Lifetime Module*, n.d.)

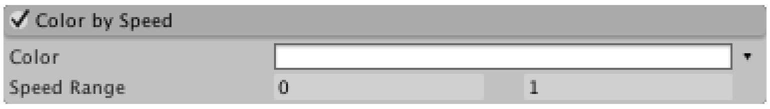


✓ Color over Lifetime

Color	

Figure 5: Color Over Lifetime module

- Color by Speed changes the color using a gradient and specifying the velocity range. (*Unity - Manual: Color By Speed Module*, n.d.)

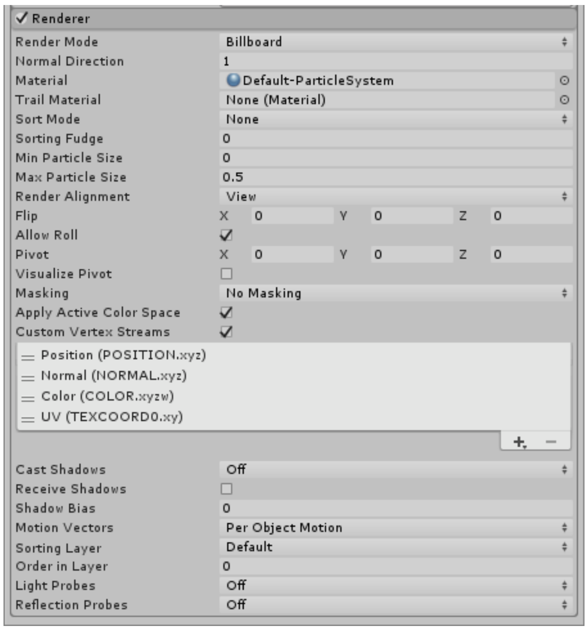


✓ Color by Speed

Color		
Speed Range	0	1

Figure 6: Color by Speed module

- Renderer module contains settings like how the particle rends, the most common useable are billboards, the material selected, the possibility to emit light or shadow and configure variables that can work with the shader. (*Unity - Manual: Renderer Module*, n.d.)



✓ Renderer

Render Mode	Billboard
Normal Direction	1
Material	Default-ParticleSystem
Trail Material	None (Material)
Sort Mode	None
Sorting Fudge	0
Min Particle Size	0
Max Particle Size	0.5
Render Alignment	View
Flip	X 0 Y 0 Z 0
Allow Roll	✓
Pivot	X 0 Y 0 Z 0
Visualize Pivot	☐
Masking	No Masking
Apply Active Color Space	✓
Custom Vertex Streams	✓

Position (POSITION.xyz)
Normal (NORMAL.xyz)
Color (COLOR.xyzw)
UV (TEXCOORD0.xy)

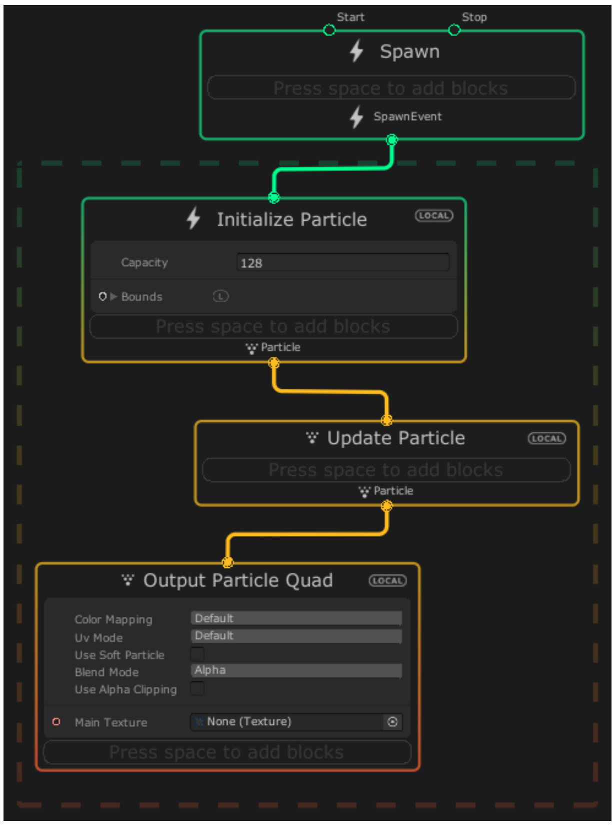
Cast Shadows	Off
Receive Shadows	☐
Shadow Bias	0
Motion Vectors	Per Object Motion
Sorting Layer	Default
Order in Layer	0
Light Probes	Off
Reflection Probes	Off

Figure 7: Renderer module

2.1.1.1. Visual Effect Graph

Visual Effect Graph is Unity package, which current version is 10.4.0, that provide a visual scripting for building particle systems (*Unity - Manual: Visual Effect Graph*, n.d.) (*Visual Effect Graph | Visual Effect Graph | 10.4.0*, n.d.). This system works directly with the memory of the GPU, means there is not a communication between CPU and GPU during the particle generation and render. Moreover, Unity's documentation specifies that with this system they can render millions of particles by compute shader and shader memory storage buffer, and maybe they work with that, that mentioned in project scope and we get more into it during the next sections.

Their main loop is based of contexts, each one belongs to different phases for generate a particle and it can contain blocks that give customization. There are four contexts: spawn, initialize particle, update particle and output particle. (*Visual Effect Graph Logic | Visual Effect Graph | 10.4.0*, n.d.)



Capacity 128

Color Mapping Default
Uv Mode Default
Use Soft Particle
Blend Mode Alpha
Use Alpha Clipping
Main Texture None (Texture)

Figure 8: Visual Effect Graph contexts

There are different types of output particle, from point to mesh, with general variables like soft particles, alpha and texture and combines with blocks for adding curves, noise filters, anti-aliasing… (*Shared Output Settings and Properties | Visual Effect Graph | 10.4.0. Particle Options Settings.*, n.d.)

2.1.2. Unreal Engine

Cascade Particle System is provided by Unreal Engine 4 (current version) and it is designed for produce a great quality of particles based on complete effects with optimizations for applying post process effects (VFX). (*Cascade Particle Systems | Unreal Engine Documentation*, n.d.)

To the best of my knowledge Unreal Engine has a complete render pipeline but there is not the possibility to write shader scripts. They only use a visual scripting for create custom materials, which it contains shader complexities. (*Essential Material Concepts | Unreal Engine Documentation*, n.d.)

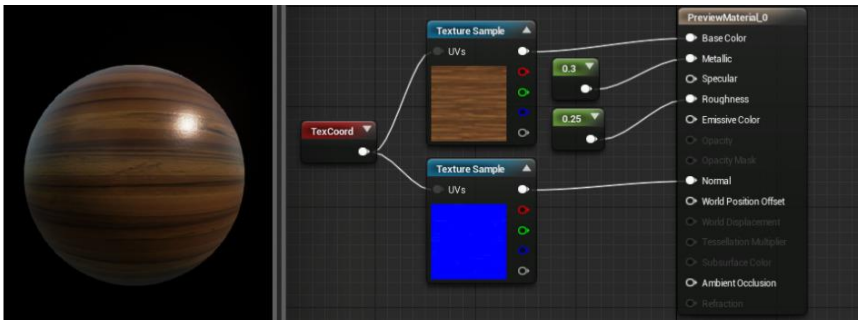
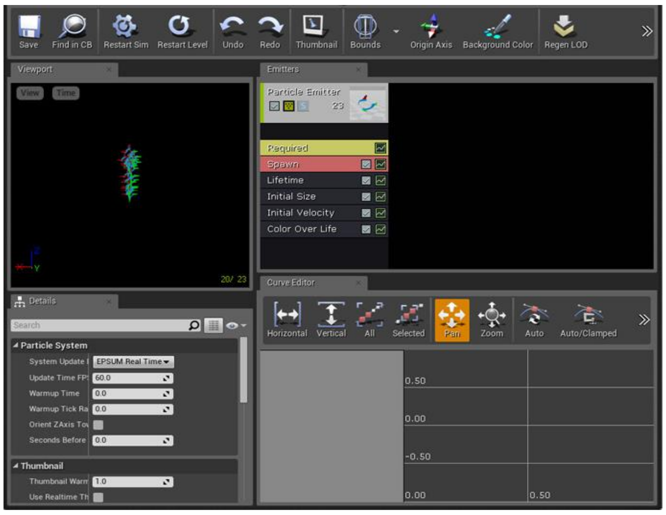


Figure 9: Unreal Engine Material editor

Unreal uses a different workspace separates from the scene for creating particles systems. There is a viewport, which displays the visual results, a cascade system for configure the emitters which it goes from breed to the final color output shape by modules, and two windows for editing values or edit curves. (*Particle System User Guide | Unreal Engine Documentation*, n.d.)



Particle System
System Update EPSUM Real Time
Update Time FPS 60.0
Warmup Time 0.0
Warmup Tick Ra 0.0
Orient ZAxis Tow
Seconds Before 0.0
Thumbnail
Thumbnail Warm 1.0
Use Realtime Th

Figure 10: Unreal Engine Particle System workspace

2.1.2.1. Modules

Below we are referring the next modules which belongs to rendering pipeline:

- Color Module defines how it is going to be drawn during its lifetime. The users can define a multiple set of colors with curves and blend alpha. (*Color Modules | Unreal Engine Documentation*, n.d.)

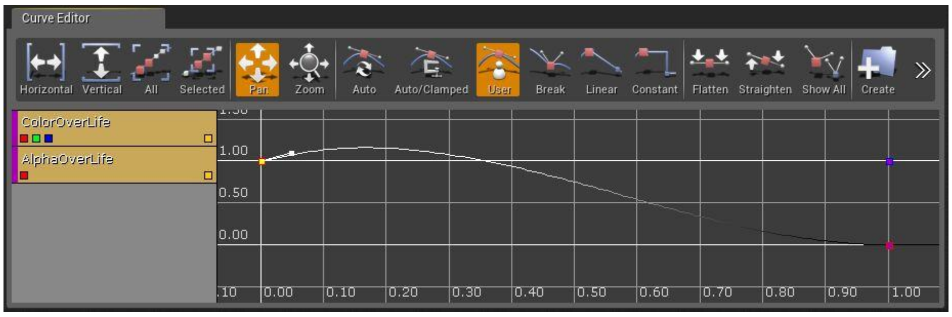


Figure 11: Unreal Engine Curve editor

- Particle Lighting extend the particles to emit light. (*Particle Lights | Unreal Engine Documentation*, n.d.)

Figure 12: Unreal Engine particle system emitting light.

- SubUV module uses the textures coordinates to draw only a section of a texture. It means that there is the possibility to create animations with particles or use a single texture as an atlas. (*SubUV Modules | Unreal Engine Documentation*, n.d.)

- Beam Module use noise filters for create a line as a particle. (*Beam Modules | Unreal Engine Documentation*, n.d.)

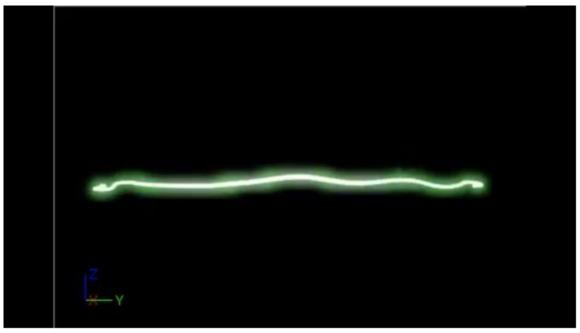


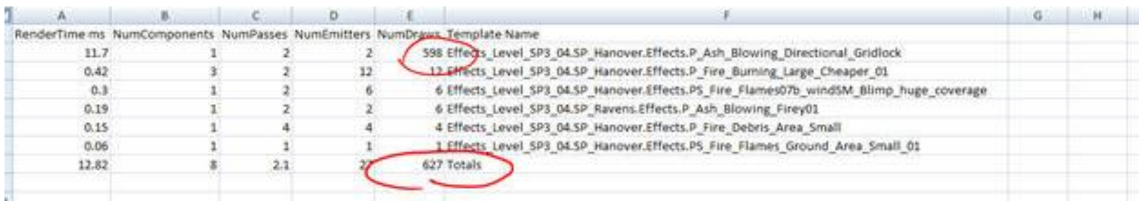
Figure 13: Unreal Engine beam particle

2.1.2.1. VFX

Unreal Engine provides different types of post processing effects which some of them improve the result of a particle system such as anti-aliasing, it means calculating more draws for gaining precision and some particles like points gain definition and bloom, which increase the effect of light with a glow effect. (*Post Process Effects | Unreal Engine Documentation*, n.d.)

We mentioned VFX because they write documentations that helps to optimize the use of particle system with post process effects. (*VFX Optimization Guide | Unreal Engine Documentation*, n.d.)

One noticeable thing is that they do profiling for each particle system which makes a possibility to implement it in our profiling system and displaying it to a graphic data visualization program.



RenderTime ms	NumComponents	NumPasses	NumEmitters	NumDraws	Template Name
11.7	1	2	2	598	Effects_Level_SP3_04.SP_Hanover.Effects.P_Ash_Blowing_Directional_Gridlock
0.42	3	2	12	12	Effects_Level_SP3_04.SP_Hanover.Effects.P_Fire_Burning_Large_Cheaper_01
0.3	1	2	6	6	Effects_Level_SP3_04.SP_Hanover.Effects.PS_Fire_Flames07b_windSM_Blimp_huge_coverage
0.19	1	2	2	6	Effects_Level_SP3_04.SP_Ravens.Effects.P_Ash_Blowing_Firey01
0.15	1	4	4	4	Effects_Level_SP3_04.SP_Hanover.Effects.P_Fire_Debris_Area_Small
0.06	1	1	1	1	Effects_Level_SP3_04.SP_Hanover.Effects.PS_Fire_Flames_Ground_Area_Small_01
12.82	8	2.1	27	627	Totals

Figure 14: Unreal Engine profile results of particle systems

2.2. OpenGL optimizations

We get throughout two technologies of OpenGL that help move all the process from CPU to the GPU by reducing the communication calls between them and yield more number of particles.

2.2.1. Shader Storage Buffer Object

There is a buffer where we can define variables stored inside the GPU, it is possible to store large arrays of information, like colors or positions. Furthermore, the fact to fill the buffer needs many resources just like read back the information. That is why it is important to upload all the information once and avoid reading from the CPU. (*Shader Storage Buffer Object - OpenGL Wiki*, n.d.)

The buffer is defined as Interface Block, that uses a specific syntax for being readable at GPU. It lies to defined memory layouts, for instance std430, which follow a memory alignment. (*Interface Block (GLSL) - OpenGL Wiki*, n.d.)

2.2.2. Compute Shaders

Compute Shaders are like multithreading of CPUs but on GPU. It defined as a method with the definition of how many executions needed. Also, it uses a shader store buffer object for calculate and save the changes and there is no way to send or return values. (*Compute Shader - OpenGL Wiki*, n.d.) (*ARB_compute_shader*, n.d.)

The main use of compute shaders is to process huge amounts of calculations, apart from the rendering. The practical application is the compute shader that does the calculations to obtain the final point of each particle on the world space, then the vertex shader calculates the four points of the plain to draw the particle and finally the fragment shader draws the particle. (*Compute Particles Sample*, n.d.)

2.2.3. Conclusions

The combinations of these two technologies leads to the performance for keeping all the information and calculations inside the GPU, therefore it can have an important impact on our product or gain more milliseconds to employ on other systems from freeing the particle processing cost.

In future projects, to develop an improved particle rate, we are going to use SSBO at the GPU, and we can import Ruben's Physics Particle System to compute shader for reaching a higher number of particles.

There is a visual example of how it looks using these techniques and technologies at *annex 8.1. Scratch of Compute Shader - 2000000 particles*.

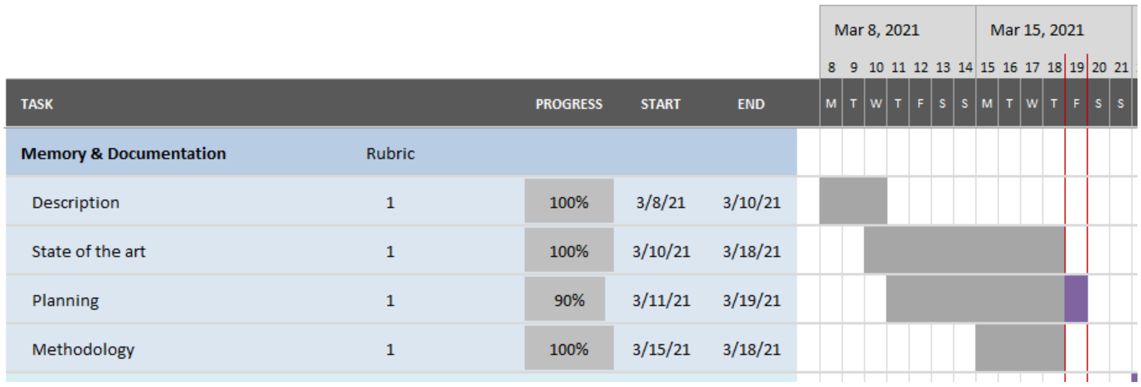
3. Project Management

After analyzing other products with the objectives, we have the next defined tasks:

1. Add support for particle rendering at our RedEye engine pipeline.
    1. Implement normal render.
    2. Implement deferred light render for particles emitting light.
    3. Implement primitives and meshes support.
    4. Implement material support.
2. Add particle system tools.
    1. Implement color and gradient attributes.
    2. Implement alpha on normal render.
    3. Implement color over lifetime.
    4. Implement color over distance.
    5. Implement color by speed.
    6. Implement curves API.
    7. Support interaction of curves with gradients and alpha.
3. Implement particle system workspace.
    1. Create a new window with a viewspace rendering the particle system.
    2. Implement controller with play and pause.
    3. Implement user interface with the tools.
    4. Testing phase.
        i. Loop: Apply feedback and iterate testing phase.
4. Add shader support.
    1. Create a shader template.
    2. Particle system workspace adapts to custom shader.
5. Release & Webpage
    1. Publish particle systems release.
    2. Update the webpage with all RedEye engine content.

3.1. GANTT

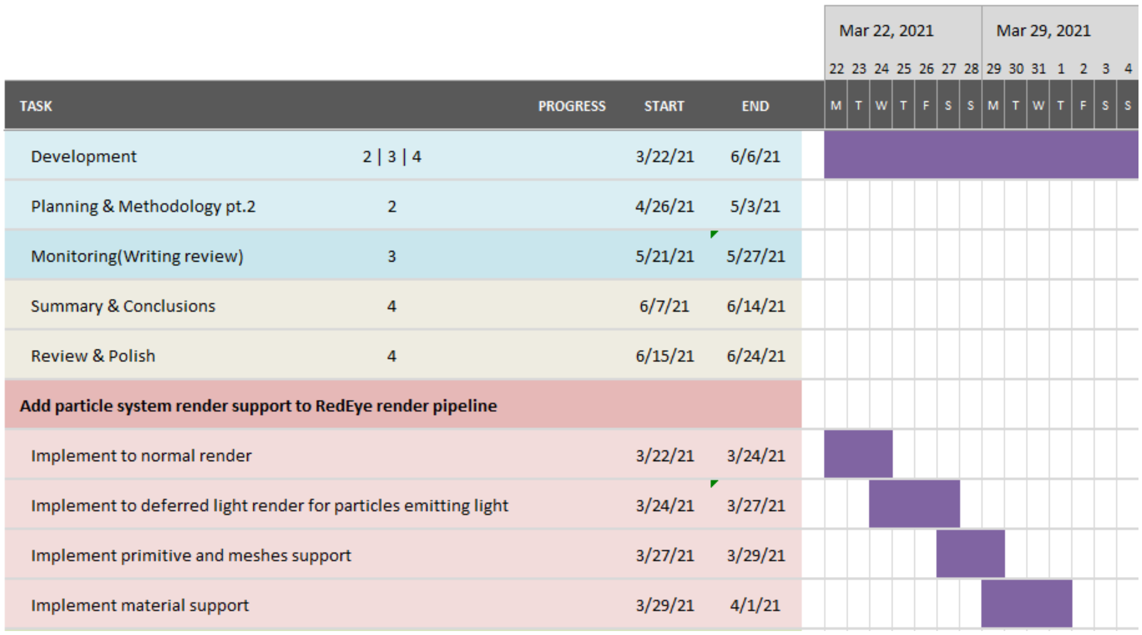
The first part of that thesis is the rubric 1. The state of the art we keep during all the other tasks for knowing what exactly tasks we want to do.



TASK		PROGRESS	START	END
Memory & Documentation	Rubric			
Description	1	100%	3/8/21	3/10/21
State of the art	1	100%	3/10/21	3/18/21
Planning	1	90%	3/11/21	3/19/21
Methodology	1	100%	3/15/21	3/18/21

Table 1: First section Gantt

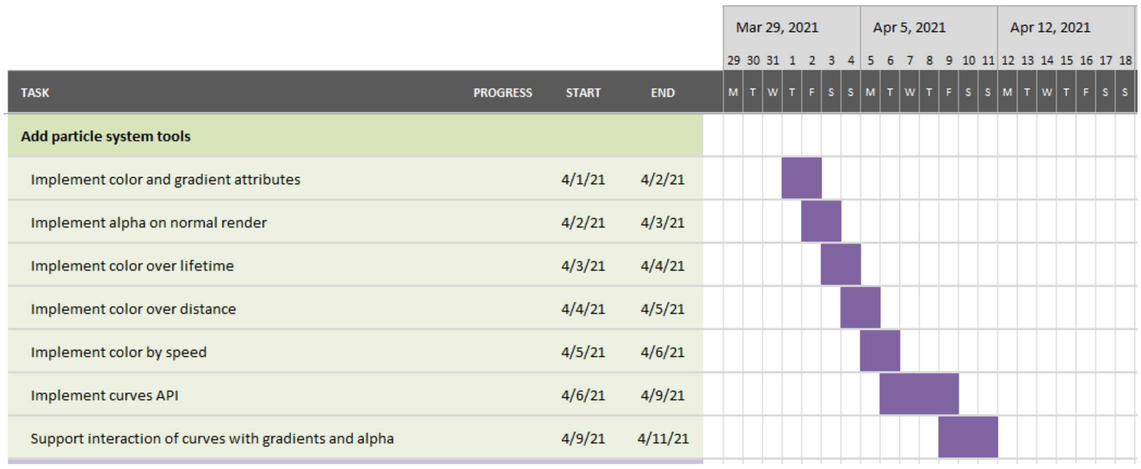
After the first rubric starts the first part of development with the thesis content at same time while implementing features.



TASK		PROGRESS	START	END
Development	2 | 3 | 4		3/22/21	6/6/21
Planning & Methodology pt.2	2		4/26/21	5/3/21
Monitoring(Writing review)	3		5/21/21	5/27/21
Summary & Conclusions	4		6/7/21	6/14/21
Review & Polish	4		6/15/21	6/24/21
Add particle system render support to RedEye render pipeline				
Implement to normal render			3/22/21	3/24/21
Implement to deferred light render for particles emitting light			3/24/21	3/27/21
Implement primitive and meshes support			3/27/21	3/29/21
Implement material support			3/29/21	4/1/21

Table 2: Second section Gantt

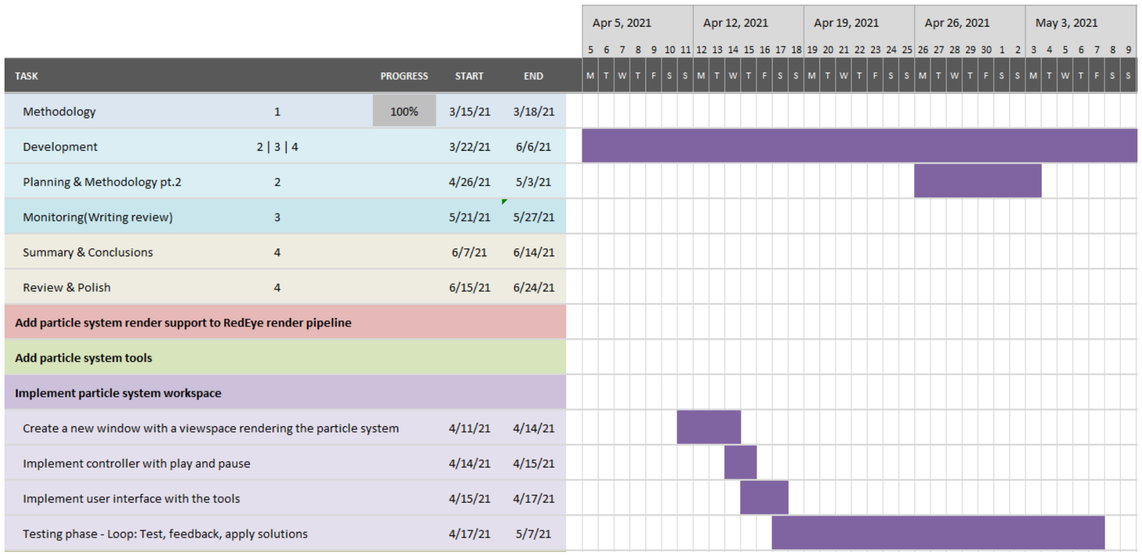
When we finish the base for working with particle systems, we are going to start implement the tools for customizing them.



TASK	PROGRESS	START	END
Add particle system tools			
Implement color and gradient attributes		4/1/21	4/2/21
Implement alpha on normal render		4/2/21	4/3/21
Implement color over lifetime		4/3/21	4/4/21
Implement color over distance		4/4/21	4/5/21
Implement color by speed		4/5/21	4/6/21
Implement curves API		4/6/21	4/9/21
Support interaction of curves with gradients and alpha		4/9/21	4/11/21

Table 3: Third section Gantt

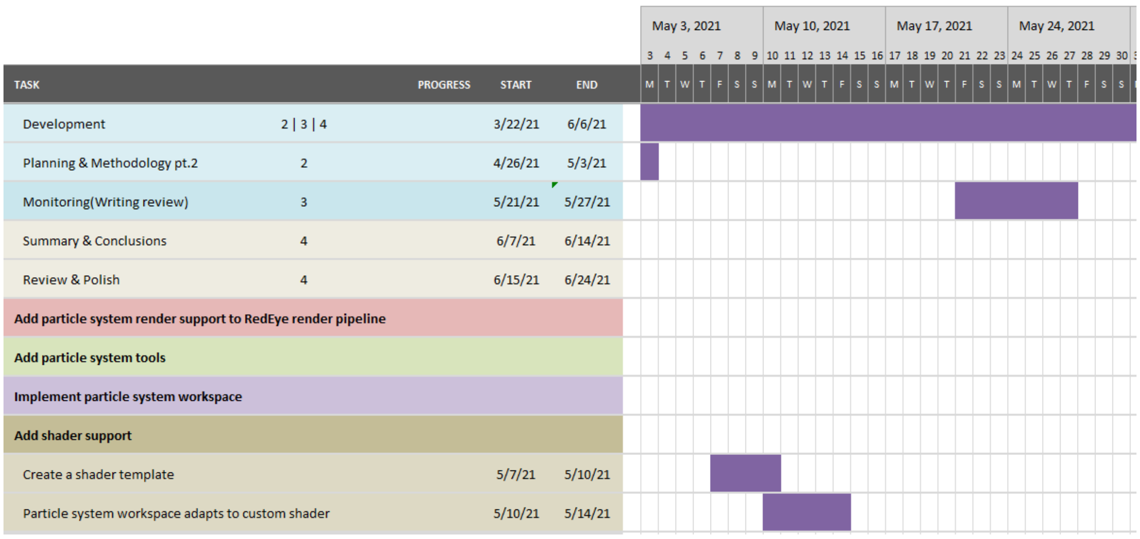
Then comes the user interface development, we are going to apply all the tools into a window to edit each particle system. After the first implementation we start the testing task.



TASK		PROGRESS	START	END
Methodology	1	100%	3/15/21	3/18/21
Development	2 | 3 | 4		3/22/21	6/6/21
Planning & Methodology pt.2	2		4/26/21	5/3/21
Monitoring(Writing review)	3		5/21/21	5/27/21
Summary & Conclusions	4		6/7/21	6/14/21
Review & Polish	4		6/15/21	6/24/21
Add particle system render support to RedEye render pipeline				
Add particle system tools				
Implement particle system workspace				
Create a new window with a viewspace rendering the particle system			4/11/21	4/14/21
Implement controller with play and pause			4/14/21	4/15/21
Implement user interface with the tools			4/15/21	4/17/21
Testing phase - Loop: Test, feedback, apply solutions			4/17/21	5/7/21

Table 4: Fourth section Gantt

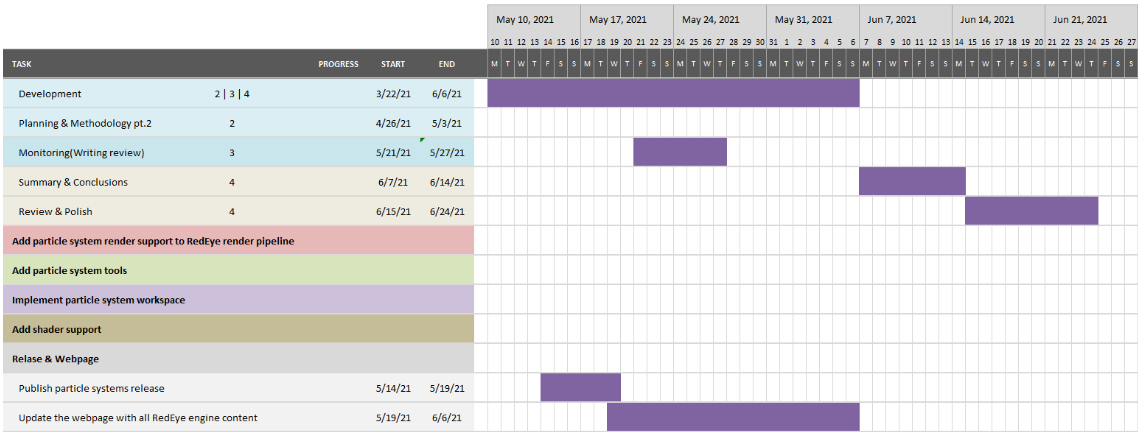
When the tests of user interface give good enough feedback, we are going to add the shader support.



TASK	PROGRESS	START	END
Development	2 | 3 | 4	3/22/21	6/6/21
Planning & Methodology pt.2	2	4/26/21	5/3/21
Monitoring(Writing review)	3	5/21/21	5/27/21
Summary & Conclusions	4	6/7/21	6/14/21
Review & Polish	4	6/15/21	6/24/21
Add particle system render support to RedEye render pipeline			
Add particle system tools			
Implement particle system workspace			
Add shader support			
Create a shader template		5/7/21	5/10/21
Particle system workspace adapts to custom shader		5/10/21	5/14/21

Table 5: Fifth section Gantt

Finally, we are going to publish the particle system release.



TASK	PROGRESS	START	END
Development	2 | 3 | 4	3/22/21	6/6/21
Planning & Methodology pt.2	2	4/26/21	5/3/21
Monitoring(Writing review)	3	5/21/21	5/27/21
Summary & Conclusions	4	6/7/21	6/14/21
Review & Polish	4	6/15/21	6/24/21
Add particle system render support to RedEye render pipeline			
Add particle system tools			
Implement particle system workspace			
Add shader support			
Relase & Webpage			
Publish particle systems release		5/14/21	5/19/21
Update the webpage with all RedEye engine content		5/19/21	6/6/21

Table 6: Final section Gantt

3.2. SWOT

	Advantages	**Disadvantages**
Internal	**Strengths** - High programming experience coding complex structures and understanding libraries/external code. - Great coworking with Ruben at RedEye Engine development. - RedEye Engine has a solid base of resources and render pipeline that help us implement the particle render.	**Weaknesses** - Redeye Engine cannot build playable demos. We can only do releases of the editor with some scenes, which we will create and save as a resource, showing the results.
External	**Opportunities** - We are going to increase our knowledge on particle systems. - We are going to complete our webpage and use it to expose our work by featuring it as part of our portfolio.	**Threats** - The problem has been solved by mainstream engines like Unity or Unreal Engine. - There are libraries like Nvidia FleX that help developers implement particle systems onto their engines. - We are two developers combining the rendering and physics pipeline to compose the particle system. Tasks are bound to have dependencies.

Table 7: SWOT

3.3. Risks and Contingency Plans

Risk	Solution
Not possible to add support of custom shader for particle render.	We still have the workspace for edit particles and we will be able to plan other tasks like optimization.
Tests from user interface do not prosper properly.	We will be able to redo the planning Gantt for continue testing in short phases during other development phases. If not still reach the desired goal, we will keep the current state of workspace developed.
Particles do not work with deferred lighting pipeline.	We can draw the particles in another framebuffer with the other pipeline render and then apply to the final draw from deferred light without the possibility to implement particle lighting.
Hardware limitations	The case of test the user interface with different collaborators, they computer specifications can be incompatible with some technologies. I developed an application that gets all the graphics information with the extensions, with that we can track errors if some computer do not executes our engine or do not render particles. (*Juliamauri/GetOpenGLInfo: Generate a *.Txt File with Your Hardware Info and Gl Integers*, n.d.)

Table 8: Risks and Contingency Plans

3.4. Initial Cost Analysis

Is we buy all the equipment for develop that project with a contemplation of salary and costs, the average overall costs will come from salary and equipment and almost all the software are free of charge. The employee is going to work 3 months with an average of 4 hours every day.

If we contemplate the project costs as a student doing the thesis, there will not costs apart from some software and services. Also, the equipment will be the student have or the university computer.

	Type	Subject	Units	Pay once	Monthly pay
Direct cost	Employee	Julià	120	- €	960.00 €
	Equipment	Computer	1	2,500.00 €	- €
		Chair	1	220.00 €	- €
		Table	1	80.00 €	- €
		Monitor	1	110.00 €	- €
		Mouse & Keyboard	1	30.00 €	- €
	Software	Visual Studio	1	- €	- €
		Github	1	- €	- €
		HaknPlan	1	- €	- €
		Google Drive	1	- €	- €
		Tableau	1	- €	60.00 €
		Discord	1	- €	- €
Indirect cost	Services	Water	1	- €	15.00 €
		Electricity	1	- €	60.00 €
		Internet	1	- €	50.00 €

Price / hour	8.00 €
Months of development	3

Monthly costs	1,145.00 €
First month pay	4,085.00 €
Total	6,375.00 €

Table 9: Initial Cost Analysis

We consider our costs of development like a junior programmer.

4. Methodology

Our work consists of two parts. One for prove our advancements on code and the other one for design the user interface.

4.1. Tracing Procedures and Tools

In this part we are going to explain all the procedures with a Gantt and the tools that will help us to develop the project.

4.1.1. RedEye Engine

RedEye engine is our platform that we worked for three years and it forms the base for implement the particle system. (*Juliamauri/RedEye-Engine: Engine 3D(OpenGL)*, n.d.)

One of the best things of our engine is the render pipeline with the resource system. (*RedEye Engine*, n.d.)

4.1.2. Visual Studio

Visual Studio is our main tool for developing the engine. It compiles and show compilation errors and debugging. Also there is a tool that create a diagram of our whole system. (*Visual Studio IDE, Code Editor, Azure DevOps, & App Center - Visual Studio*, n.d.)

4.1.3. Discord

Discord is a voice, video, screen share and instant messaging multiplatform for communities, available on their webpage, pc and smartphone.

In our case we create a server, where we can track and keep communication of our work. We defined three roles for us, the tutor and people that want to help us testing our advancements, the UI for create particles. (*Discord | Your Place to Talk and Hang Out*, n.d.)

4.1.4. HacknPlan

HacknPlan is a platform that helps to organizes the milestones. It uses cards as a task and we place them in four different stages: planed, in progress, testing and complete. Moreover, everything can contain descriptions, images attached for visualize clearly and the cost time, when we do a task. We can log the spent time and get a statistic for know if I spend more time to get the milestones. (*Project Management for Game Development - HacknPlan*, n.d.)

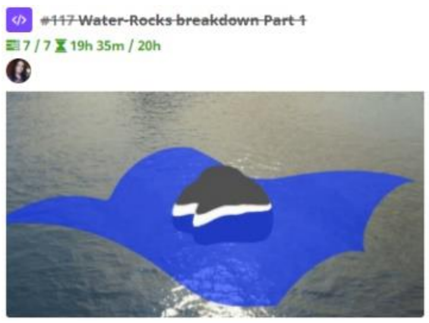

~~#117 Water-Rocks breakdown Part 1~~
7 / 7 19h 35m / 20h

Figure 15: HacknPlan task from RedEye engine

4.1.5. GitHub Repository and Versioning

GitHub is a platform where we can upload our code, which logs all the changes and helps to some possible errors if the applications have bugs, and the possibility to publish versions with all the content explained. (*GitHub*, n.d.)

4.1.6. Google Drive & Google Forms

Google drive is a cloud storage, where we can store files online, which we are going to save all the documents with references and multimedia from the engine, there.

Moreover, we are going to use Google Forms for doing surveys about the user experience tests. (*Google Drive*, n.d.)

4.1.7. Tableau

Tableau is a platform that we can visualize our data how we want to. For instance, using different types of graphics, maps, bars..., etc.

This application plays an important role in our project due to helping us to demonstrate our advancements. (*Software de Análisis e Inteligencia de Negocios*, n.d.)

4.2. Validation Tools

Our method for validate the goals is to profiling the engine, currently we support Optic and custom profile with Tableau. The engine generates a ''json'' file that contains all the time of processes that are defined on the code, with this we can understand how many milliseconds is spending for one frame.

In addition, we will be able to do the tests with taking advantages of collaborating with others. We are going to do UX testing.

4.3. Code progression methodology

We are going to use the fourth phases of hacknplan for task's structure: planned, in progress, testing and completed. Every main goal is defined as a table with dividing into different tasks to do.

When we are working with one task it will be in progress tab, when we complete it, we change to testing. On testing part, we are going to specify the actions that the users need to do for testing, if there are bugs or it is necessary to do changes. Also, we are going to do a profiler test to obtain an output which shows what we have done is working or if we have any performance losing in the process. Finally, if there is no bug and the profiler result is reasonable, by reasonable we mean there is not a big loss of performance considering what is done or gain performance if we do optimizations.

Moreover, on hacknplan we can specify ~~the~~ times that is needed to do the task and also the times we registered for doing it. If we spend more time than we expected, we can do changes to avoid having delay on the project.

On the other hand, I assigned hours for adding or changing usability features that I am going to analyze the editors with testers. In case of wanting to add some new features, I need to study and figure out if it is possible to implement it. This part is the results of the next methodology plan about user interface.

Furthermore, when we are going to do the usability tests, we will gather information of the hardware for detect software limitations with hardware if something goes wrong. Also, it helps us to make the products more compatible with different hardware.

Finally, if some bugs appear after the tasks is done or during the task, we are going to write an issue on the Github repository for keep control what currently is needed to fix.

4.4. Tests of User Interface Design

We want to test the editor with external people for prove and upgrade the user interface that will be able to create particles.

During development we create a basic editor using all the features while we are implementing the particle render system.

If there is a basic tool for elaborate particles, we need to test it with testers. First, we explain in a document all the directions that the users need to do for create a particle without explaining exactly what they need to do. Also, the testers can use the tool as they want, could be to do something more than the instructions, like a playground. During testing we will record our screen for collecting information in case something goes wrong. Finally, we will pass a survey for answering some questions that will helps us to find the errors or create a new usability feature.

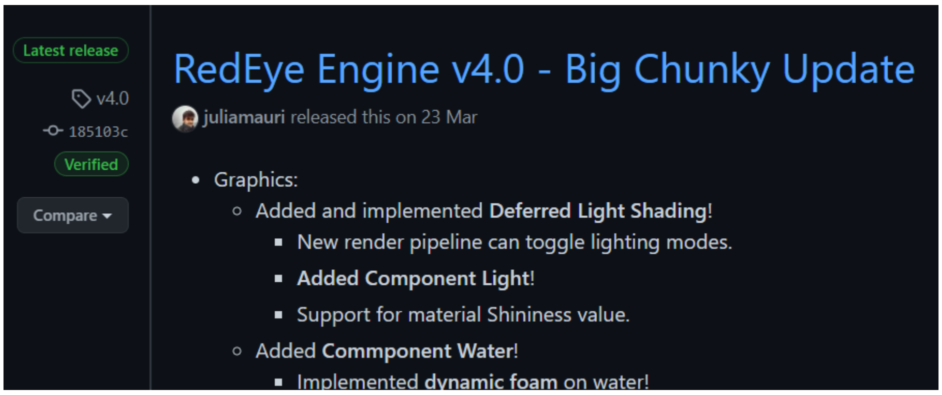
To sum up, the steps that we are going to do are: create editor, test it with users and fix errors or analyze them for helping us to implement usability features, adding new tasks to the planning.

5. Project Development

5.1. Project Setup

The base for working with that project is RedEye Enigne and for start work with particles first we need to create a release, which is a start point for working that contains all that changes done in the past, the engine has changes since year ago without doing a release. Releases is important to make a start point of work and we have the security that all changes work correctly, and we do not need to code again that specific thinks until a bug appears, or we need to change something for optimize purpose.

The release that we create is the 4.0 version that contains one year of changes from render, performance, utility, new components, audio module, profiling…



RedEye Engine v4.0 - Big Chunky Update

juliamauri released this on 23 Mar

- Graphics:
  - Added and implemented **Deferred Light Shading**!
    - New render pipeline can toggle lighting modes.
    - **Added Component Light**!
    - Support for material Shininess value.
  - Added **Commponent Water**!
    - Implemented **dynamic foam** on water!

Figure 16: RedEye engine release before starting the implementations.

Before start coding we will create a new branch named Particle-Pipeline and upload all the changes, when we finish the project, we will move all the changes to master and create a final release.

5.2. RedEye Engine modules

RedEye Engine works with modules that all together makes a sophisticated game editor.

- Input handles and send all events received from user input signal from keyboard, mouse, gamepad, dropped files or custom, which we send from other modules, and send it or make readable to other modules.

- Scene handles all the objects that we introduce and modify them.

- Physics is the new module that Ruben will work for simulate the particles and other objects for future use.

- Editor uses ImGui library for display windows and tools that we create and with them we edit the scene or modify resources.

- Renderer is the responsibility to draw all elements from scene and editor tools to our display monitor.

- Audio reproduces sounds to our speakers.

Independent modules:

- Filesystem handle all the files that contains our project directory and read all the changes we made and add the files that we drop to our application. There is a window that display all assets, which is resources, that we created on our engine, we use thumbnails for display an image that represents the contents.

- Resource system handles all the resources that we create or generate by the files we dropped. There are materials (render information), shaders (render code), 3d models, meshes, scenes, skyboxes, textures… Also, our particle emitter will be a resource. There is a count reference that when we load an object to the scene if the resource wasn’t loaded we load it and when we delete the object if there are no other object using that resource we unload it.

5.3. Renderer Pipeline

We use shaders that is a code for graphics card unit and use it for draw all the information that we send it.

Using frame buffer objects, which is an array of buffers that are textures that be useful for share to different shaders or extract as an image, for drawing the scene content and extract as an image to display it on a viewport at editor.

VBO contains vertex information and save position, normal, texture coordinates, tangents, bitangents, also color.

EBO index and order the vertex, from VBO, to draw.

VAO links VBO, EBO and arrays of different information that contains VBO. Separate every vertex to different imaginary buffers.

Every object is linked to a material that contains information like the color, textures, transparency and when we draw the object, reads the material and upload to the shader.

We use flags for let the renderer know how we want to render the scene.

- Frustum culling is a method to grab all the elements that contains the camera and draw. If it is disabled the renderer will draw all the elements, inside and outside the camera, and low the performance.
- Outline selection is for visualize the selected object from scene, the renderer will draw a line at the object edge.
- Debug draw is for display the information from scene like the Axis-Aligned Bounding Boxe, frustum from camera, etc.
- Skybox is a big sphere that englobes all the scene and simulates an environment using textures.
- Blended is to enable the transparency of objects is their material use that.
- Wireframe only draws the edges of all objects on scene.
- Face culling calculates to draw the correct face of the objects when the camera is inside or outside.
- Depth test helps to draw the correct distance of the elements and don't draw is some object is behind. There are different modes for depth. As example, when we draw the skybox, we use the mode that only draws if there not other object drawn on the pixel.
- Clip distance draws all objects to a specific altitude.

The render pipeline situates almost at end of the main loop of the engine. First the engine process all the information and then draws it. The render process consists the next steps:

1. Draw thumbnails: when we add some new resource or it changes we accumulate a queue of thumbnails to render. We use the basic render operations and then extract it to a file for avoid rendering every time the engine starts. Currently that method do not work correctly and almost all the thumbnails draws incorrectly or directly is an black image.

2. Scenes draw: we always draw two times the engine. First the editor view and then the game view, that use a camera object located inside the scene. We can render the scene with two different methods. The normal render and deferred light render.

    - The normal render do not draw lights.
    - The deferred light render first stores all the objects information at different textures located at frame buffer objects. Then we do a light pass that use all the information stored with the lights for render the final output.

    The next steps define the order that we draw the scene:

    1. Reads the flags and set up the graphics card unit with OpenGL calls.
    2. Fill the shader parameters with basic information of camera, delta time, etc.
    3. Get the objects from scene if the frustum culling is enabled we process the dynamic aabb tree for get the objects inside the frustum of the camera if not we get all the objects.
    4. Set the light mode.
    5. Draw the scene objects, transparency objects will be separate and draw at end. If normal render the draw will be the final color, if it is deferred mode will draw the object information on the frame buffer objects.
        i. Light Pass (if deferred render enabled)
            1. Draw transparency elements because we can not draw transparency on deferred then we fill the objects information without transparency.
            2. Light Pass, we get all the lights and draw the final color with the influence of light. After light pass we draw the elements that we do not want to be affected by light.
            3. Debug draw.
            4. Draw skybox.

ii. Normal Render.
   1. Debug Draw.
   2. Draw Skybox.
   3. Draw transparency objects.

6. Draw the outline selection.

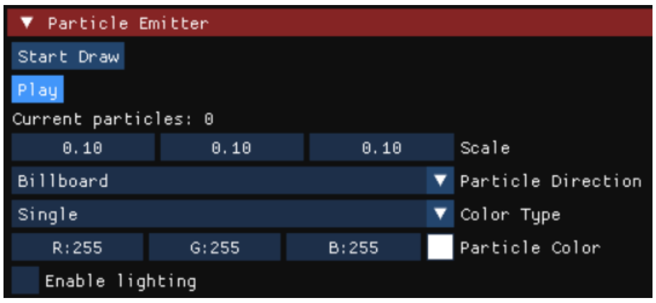
3. Draw editor, all the ImGui user interface elements and scene viewports, that we draw to a frame buffer object previously.

5.4. Adding component particle emitter

All the scene objects are store in a pool, we have pools of gameobjects and every type of components. That helps to iterate everything more faster and void to allocate and deallocate every time when add or delete things. Moreover, we avoid the recursive methods. Also, our pools are stored in a hash map structure for find fast a specific object, we use UIDs, of 64 bits, for identification.

Then, for adding a new component we need to define a new pool and implement it to our scene manager.

All components have properties, and we display it to the user interface for modify them. The particle emitter component we do a playback state to control if we want to play, pause or stop the simulation. Also, we do a temporally button that enables the particle draw.



Current particles: 0
0.10 0.10 0.10 Scale
Billboard ▼ Particle Direction
Single ▼ Color Type
R:255 G:255 B:255 Particle Color
Enable lighting

Figure 17: Particle emitter properties.

The emitter information will be stored as a resource and then the component contains the reference of that emitter. For now, every component has their own simulation and no share it to other components.

5.5. First particles and billboard

The first draw of particle will be a point, which is a small triangle because the graphic card unit only works drawing triangles, and every component loads their own triangle. A possible optimization is moving that triangle as a primitive component and avoid loading one triangle every new component.

For now, we will use the normal render and use the basic shader that we have on our engine.

We use billboard for display correctly the triangle, that means that the triangle always will point to the camera. It is because is a plane and maybe in some camera positions we cannot watch the particle correctly.

The render pipeline first will get the particle manager and then draw it as last, like a blend image because we want to profile and give priority to the other objects of the scene before drawing particles.

As we mentioned before our pipeline, we will situate the particle systems draw before the debug draw. In the case that we use transparency will be draw after the other transparency objects.

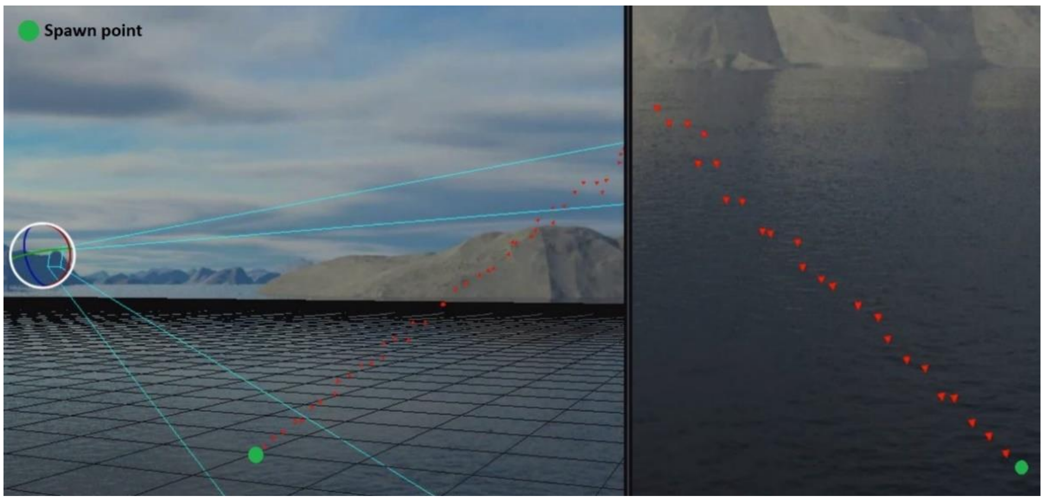


Figure 18: First particles with billboard.

5.6. Rendering particles on deferred light pipeline

The purpose to adding particles to deferred light pipeline is for display the particles with the lights we add on scene and the possibility to emit light with the particles.

On deferred light pipeline we will add the particle systems draw before the transparency elements draw inside of light pass part, the reason is the same as the normal render.

The next image we show the result of some particles being affected by some point lights.

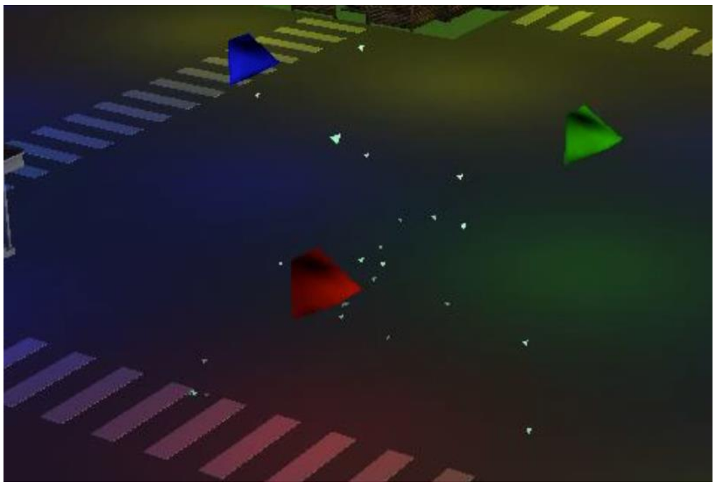
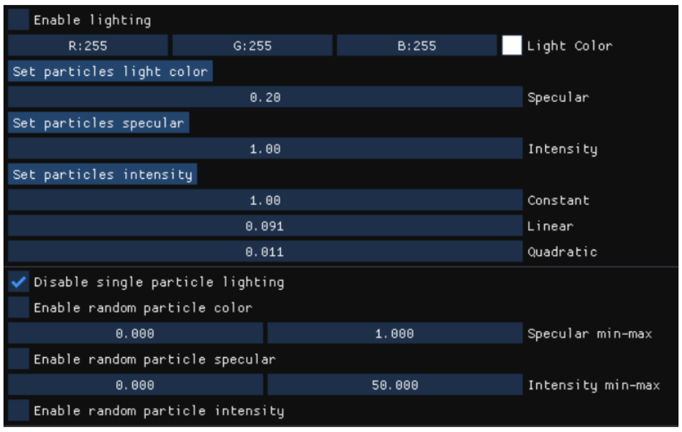


Figure 19: Particles being iluminated.

In the case that we want to emit light of that particles we need to pass all positions of the particles and light values, form the simulation, to the light pass. For now, we interact every particle as a point light. On proprieties window we can add a checkbox for enable the light particle and all the parameters to modify the light values, such as color, intensity, specular, constant, linear and quadratic. Furthermore, we can individualize each particle light parameter and randomize color, intensity and specular.

Moreover, we randomize some values, when we activate the checkpoint will modify all the particles and the new spawn particles will set up with a random variable.



```
Enable lighting
R:255    G:255    B:255    Light Color
Set particles light color
0.20                       Specular
Set particles specular
1.00                       Intensity
Set particles intensity
1.00                       Constant
0.091                      Linear
0.011                      Quadratic
✓ Disable single particle lighting
Enable random particle color
0.000        1.000         Specular min-max
Enable random particle specular
0.000        50.000        Intensity min-max
Enable random particle intensity
```

Figure 21: Particle light properties.

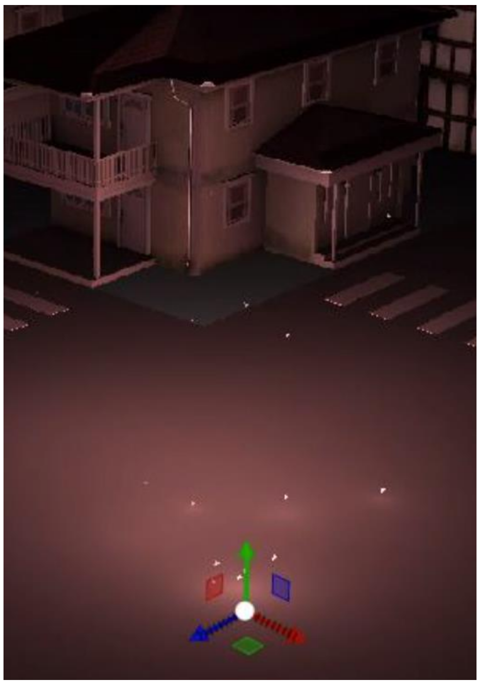


Figure 20: Lighting particles.

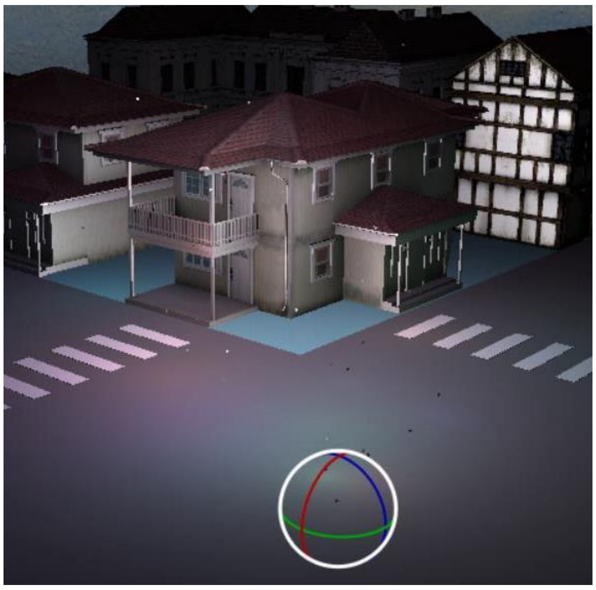


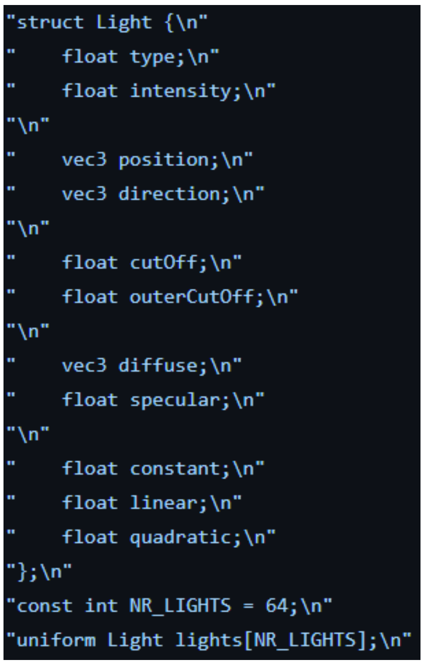
Figure 22: Random lighting particles.

Currently we can only emit 64 lights. The next statement we will see how to increase significantly that number.

5.7. Increasing the capacity of light focusing on shader memory

The fact that we will increase the number the lights is for use tons of lights on different places, as example the shots of guns of a multiplayer game or the sparks of some electronics. Now with 64 lights we are limited to use a lot of particles emitting light.

On the next image we can see our current structure where we upload the light info, we can watch that the number of lights is 64 and if we augment it the shader will not be working and throw an error.

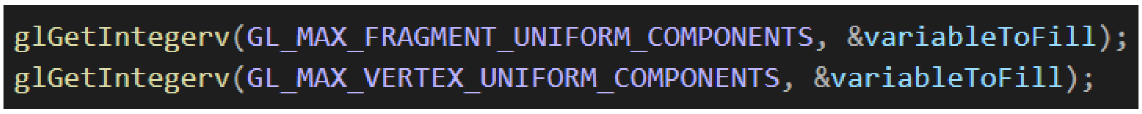


```
"struct Light {\n"
"    float type;\n"
"    float intensity;\n"
"\n"
"    vec3 position;\n"
"    vec3 direction;\n"
"\n"
"    float cutOff;\n"
"    float outerCutOff;\n"
"\n"
"    vec3 diffuse;\n"
"    float specular;\n"
"\n"
"    float constant;\n"
"    float linear;\n"
"    float quadratic;\n"
"};\n"
"const int NR_LIGHTS = 64;\n"
"uniform Light lights[NR_LIGHTS];\n"
```

Figure 23: Struct of light.

That is because the uniforms buffers (the array of lights) are limited to a specified memory. The uniform is the parameter to specify that this we will use for upload at the shader from the CPU to the GPU. When we want to upload several data, we use uniform buffer object.

The problem is that the total memory of uniforms is 16 kilobytes, 16.384 bytes. The method to know that value is calling an OpenGL function that give information about the GPU, the method is glGetIntegerv and the flag that we use is GL_MAX_FRAGMENT_UNIFORM_COMPONENTS. That parameter give to you the max components that can handle one fragment shader, normally the common value is 4096, each component is 4 bytes, then we get the 16 kilobytes.



```
glGetIntegerv(GL_MAX_FRAGMENT_UNIFORM_COMPONENTS, &variableToFill);
glGetIntegerv(GL_MAX_VERTEX_UNIFORM_COMPONENTS, &variableToFill);
```

Figure 24: OpenGL parameter calls.

Then in GPU memory one float is 4 components, it means 16 bytes. With that we need to ask why 16 bytes when in CPU one float is 4 bytes. Four floats in CPU are one float in GPU, it is expensive. Then if we declare an uniform of vec4 we expected that is 16 components, but is not the case, in reality is 4 components.

The explanation of that is the GPU memory works with an alignment and the minimum alignment is 16 bytes, 4 components.

Now that we understand how works the GPU memory, it is necessary understand how the memory works at uniforms buffers. The moment that we declare a structure if we do not define the type of layout will be “std 140” by default that seeks the rules that is mentioned before.

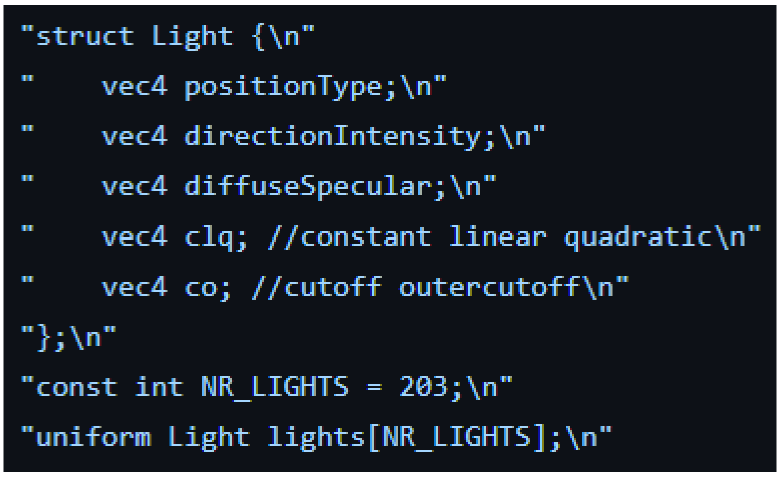
A good example for explain our current state of the struct of light is to compare like a closet with shelves. Every shelve is a capacity of 4 components and only can be declared one variable, the next table will introduce our current state of our closet.

float type	<- 4 components
float intensity	
vec3 position	
vec3 direction	
float cutoff	
float outercutoff	
vec3 diffuse	
float specular	
float constant	
float linear	
float quadratic	

Table 10: Closet example.

In that case we have 11 shells that means 44 components and we declared 64 lights. The max components that the shader can contain is 4096, then in total we are using 2816 components and we can increase the number of lights until 92, which is 4048 our limit because there are other uniforms using on the shader code.

From here we can optimize that code and we can use almost the double of lights. The parameter that can use completely our shelve is a vector of 4 floats, which is 4 components. Now that we know that we can reduce our code to the next image:



```
"struct Light {\n"
"    vec4 positionType;\n"
"    vec4 directionIntensity;\n"
"    vec4 diffuseSpecular;\n"
"    vec4 clq; //constant linear quadratic\n"
"    vec4 co; //cutoff outercutoff\n"
"};\n"
"const int NR_LIGHTS = 203;\n"
"uniform Light lights[NR_LIGHTS];\n"
```

Figure 25: Light structure optimized.

That is mean the next closet:

vec4 positionType
vec4 directionIntensity
vec4 diffuseSpecular
vec4 clq
vec4 co

Table 11: Closet from optimized structure.

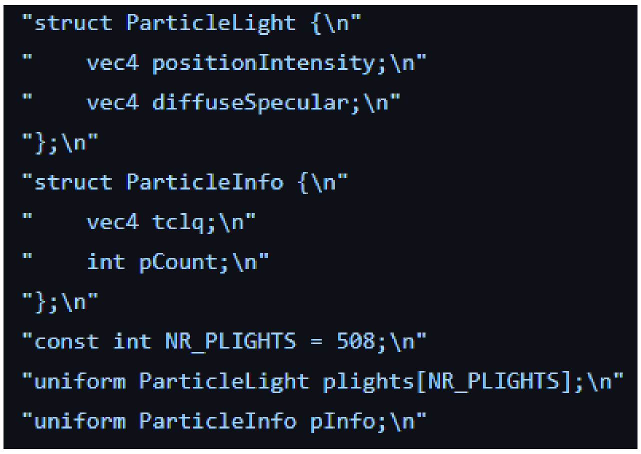
Now we are using 5 shells, which is 20 components, and we can use a total of 203 lights, 4060 components. We upgrade 2.2 times the total of lights changing the structure.

If we want to declare a matrix 4x4 will be 16 components, it is like 4 vectors of 4 floats.

In that moment we have one light pass that can handles 203 lights from components lights or particles.

The next step is to create a light pass shader only for particles because on particles we only do point lights. In that case, we can do double render light pass, first for the component lights and then for particles lighting.

We know that our particles only emit point light, we can define the next structures.



```
"struct ParticleLight {\n"
"    vec4 positionIntensity;\n"
"    vec4 diffuseSpecular;\n"
"};\n"
"struct ParticleInfo {\n"
"    vec4 tclq;\n"
"    int pCount;\n"
"};\n"
"const int NR_PLIGHTS = 508;\n"
"uniform ParticleLight plights[NR_PLIGHTS];\n"
"uniform ParticleInfo pInfo;\n"
```

Figure 26: Particle light structure.

Every particle has their position, intensity, diffuse and specular and the other light variables will be the same for all, which is declared in another structure.

That particle light uses a total 8 components, then we can draw tons of lighting particles, a total of 508 particles, which is 4064 components.

On that point we can configure our render settings to share the same light pass for lights and particles or do two times the light pass and use tons of lights.

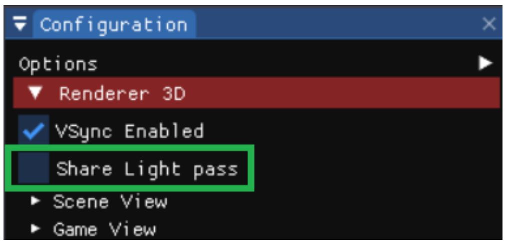


Figure 27: Share light pass check button.

The below image I show an example of the engine running 203 points lights and 508 particles lights at the same time.

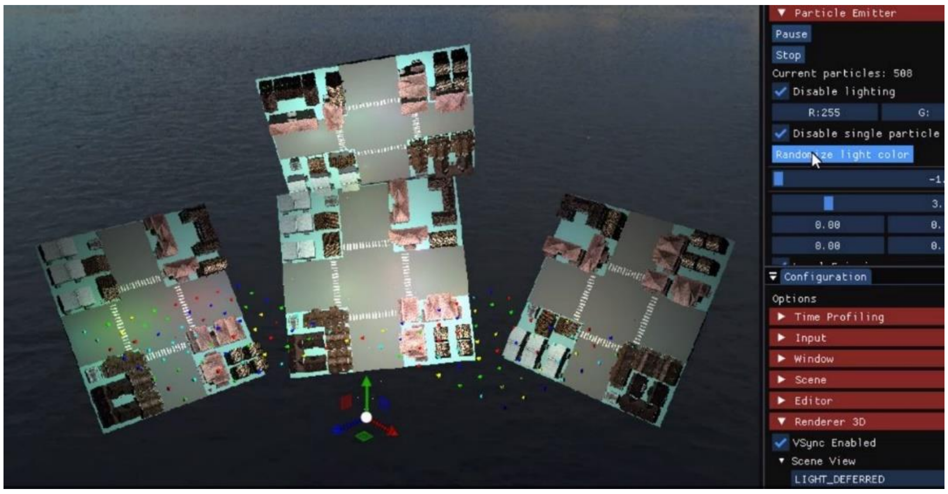


Figure 28: Scene using all lights capacity.

The frame rate is 26 per second but this build of the engine is running as debug, then we can assume that can be running at good performance. Moreover, when we do the profiling, it will show how many resources the particles are using.

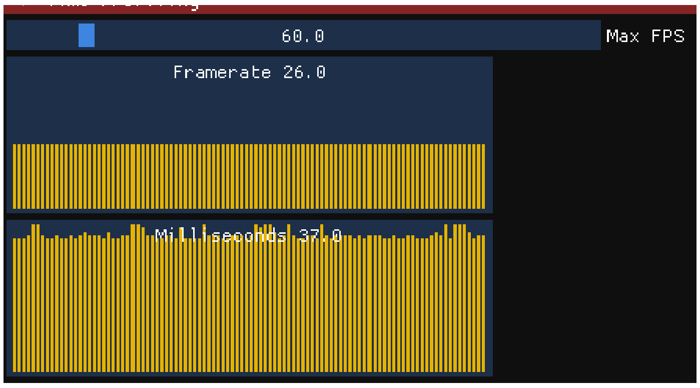


Figure 29: Performance from a scene full of lights.

5.7.1. Render debug window

We create a new window for display some render information, on that case we will display the number of lights is currently displaying for profiling reason. It changes if we change the share light pass mode.

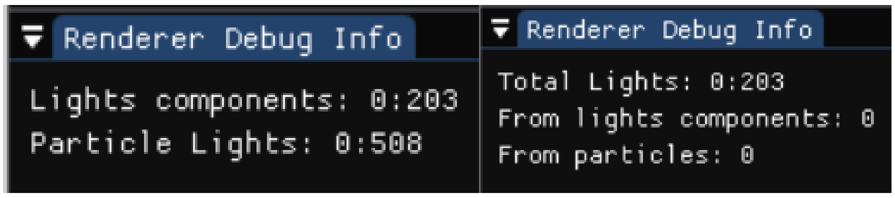

```
Renderer Debug Info
Lights components: 0:203
Particle Lights: 0:508
```
```
Renderer Debug Info
Total Lights: 0:203
From lights components: 0
From particles: 0
```

Figure 30: Render debug info window.

5.7.2. Shader Memory Access Synchronization

Until now we have been working with a Nvidia Titan X (Pascal) every implementation works without any problems, but when we worked with two another's laptops that uses Nvidia GTX 970M and 980M the deferred light pipeline started to work with problems.

The final rendered texture started to fail showing artifacts and drawing everything in white, when it would be completely dark because there are not lights on the scene.

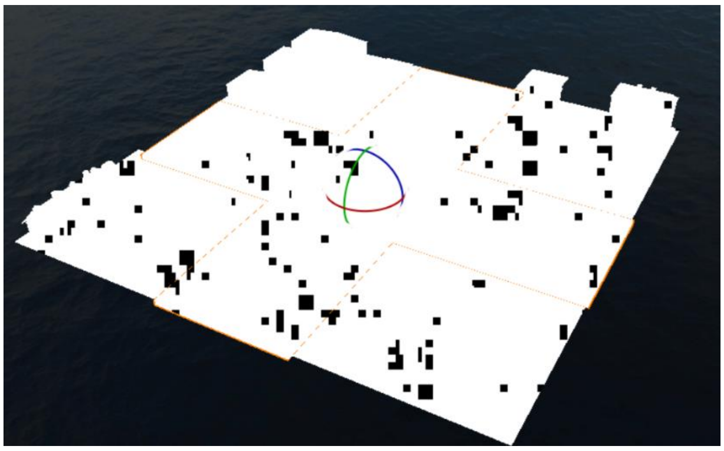


Figure 31: Renderer artifacts.

The explanation of that problem is when we do a shader operation which modify any buffers, it costs a lot of performance if we need to use it on another process at the renderer pipeline, like read the buffer filled on another shader. OpenGL do not wait to the buffer for avoiding this loss of performance.

Now that we comprehend this process, our problem appears because the laptop GPU have a heavy workload, compared with other powerful GPUs, and when we do the double light pass, one for component lights and then the particle lighting, appears that artifacts because the buffer was not ready for being readable.

The solution of that problem is calling a specific method from OpenGL before using our framebuffers to other shaders which is

glMemoryBarrirByRegion(GL_FRAMEBUFFER_BARRIER_BIT) (The OpenGL Graphics System: A Specification (Version 4.6 (Core Profile)) Segal et al., 2019, Chapter 7.13.2 Shader Memory Access Synchronization)

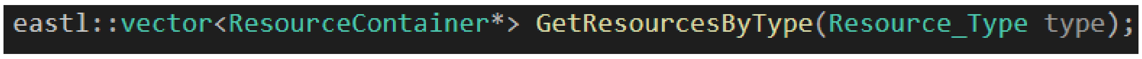
That method is telling to the GPU wait until the framebuffers are ready for being use. It is like a barrier that do not let continue until it is ready.

During the search of that problem there are not clear solutions that solves that problem. We asked a community of graphic programmers and they helped us understand what happened, where we can find the explanation and solution of that problem. The community is located on Discord.

5.8. Particle meshes

On that point we have particles, which are triangles that simulate points, and can emit light. Now we are going to use our meshes loaded as resource for drawing as particle. It is a way to make a customizable particle.

Our resource system we have a method to get all resources specifying a type, then on the properties to modify the emitter we get all meshes and select the desired one.



```
eastl::vector<ResourceContainer*> GetResourcesByType(Resource_Type type);
```

Figure 32: Get resources method from resource manager.

The only problem is that the mesh resource come from model resources, when we load a resource, our importer generates a mesh resource if it was not created previously and that generation do not specify the name and where comes, only the model resource know which meshes resources owns. Now the emitter properties only display a list of meshes numerated and we cannot know which mesh is without selecting. We will fix that part at the particle emitter viewport window, where we will implement usability features.

Finally, our mesh resource has its method for draw the mesh, then at the emitter draw will call that method all the time that the particles are in the simulation with a previously shader parameters upload for each particle.

5.9. Particle primitive

5.9.1. Emitter settings

Like the particle meshes we will implement the particle to draw all the primitive that we have implemented. We are going to setup our emitter to display a list of all primitive available, the only primitive that it is not really implemented is the point, when we select point primitive will use the vertex data stored inside the emitter component. Also, there are the optimization to implement that point primitive as a component and work exactly like the other ones.

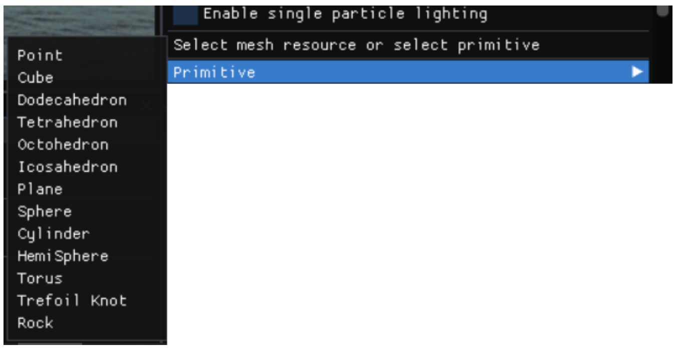


Figure 33: Primitive selection from emitter properties.

The emitter draw will get the VAO and the triangle count from the primitive for call the OpenGL draw method directly.

5.9.2. Primitive resource system

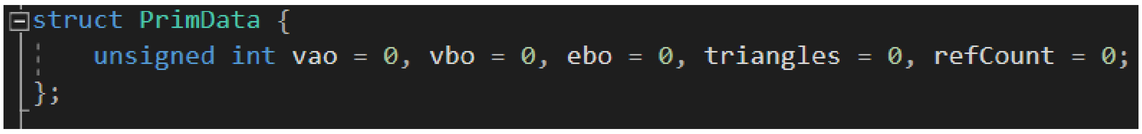
The moment that we increase the quantity of lights, which component light contains a sphere primitive, which contains the same properties and mesh vertices, when we spawn the maximum of lights the engine closes. (*Crash: Instanciate Max Lights (64 Lights) * 3 Times · Issue #4 · Juliamauri/RedEye-Engine*, n.d.)

That crash is because every time we spawn a particle component will load all the vertex data at GPU, then when we spawn 200 lights will take up all the GPU vertex memory.

The solution is creating a resource system of primitives for avoiding uploading the same buffer of vertices at GPU.

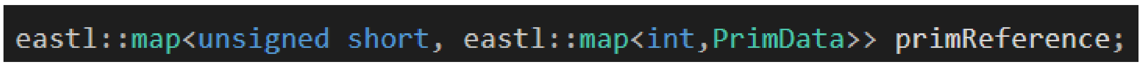
When a component primitive is generated or the user select a primitive on the emitter, it will request the primitive VAO and triangles count giving its properties. If it was not created, it would generate the primitive and upload to the GPU and the case that the primitive changes or is deleted will unload from the GPU if there no other one using the same primitive, we use a count reference for knowing that.

The platonic primitives will contain the same primitive data and the parametric will use an id, which changes with the parameters specified and we can know if it was generated previously.



```
struct PrimData {
    unsigned int vao = 0, vbo = 0, ebo = 0, triangles = 0, refCount = 0;
};
```

Figure 34: Primitive data structure.



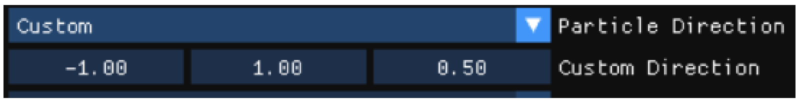
```
eastl::map<unsigned short, eastl::map<int,PrimData>> primReference;
```

Figure 35: Parameter of primitives resource system.

5.10. Particle direction

Now that we can use primitive or meshes for particles. Billboard will be a bad use and we cannot appreciate the object because it is pointing to the camera and we can only watch the same part for all particles, then we will add two more options for calculating the particle direction.

We can calculate from the particle system direction or adding a custom direction.



Custom | Particle Direction
-1.00 | 1.00 | 0.50 | Custom Direction

Figure 36: Custom particle direction properties.

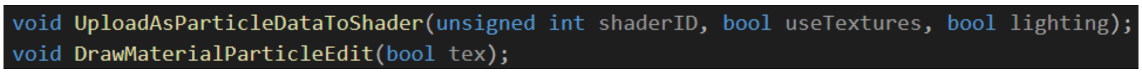
From particle system direction will use their front vector for calculate the rest of the transform. Otherwise, the custom will explicitly specify the front vector that the particle uses.

5.11. Material support (temporally)

We want to use our material resource to leverage its use to particles and the possibility to get one material already created and link it to at particle emitter simulation.

The material resource has its own method to upload all variables to giving shader.

The case that we adapt the material to the particle emitter we need to add new methods for not changing the current state of the material and concern all the render pipeline. We must implement two new methods at the shader for display the parameters that we want to use only to the particles and a new upload shader method.



```
void UploadAsParticleDataToShader(unsigned int shaderID, bool useTextures, bool lighting);
void DrawMaterialParticleEdit(bool tex);
```

Figure 37: Material methods for particle emitter system.

The particle emitter system component proprieties will show a list of all materials resources and when we select one of them will display our desired values.

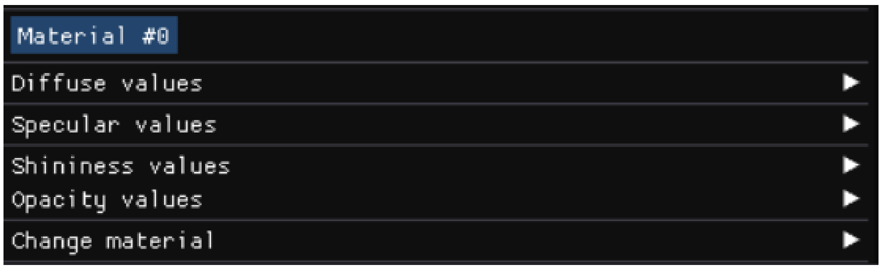


Figure 38: Material properties inside of particle emitter component.

The fact that we want to add gradient or other render parameters at the particle emitter we need to adapt the material for save that values and it means change its serialization. The best solution is removing that implementation and make a new resource of a particle system and the component will contain the reference, that resource will contain only the values of the particle system and it can be loaded on different simulations at the same time.

5.12. Coloring particles

This statement we will implement different ways to give a color at particles.

5.12.1. Single color

We define one color which will be used for all particles at simulation.

5.12.2. Gradient

We use two colors for define the start and end color. We can use it in multiple ways knowing the next formula.

$$Final\ color = start\ color * weight_1 + final\ color * \ weight_2$$

Equation 1: Final color from gradient.

$$weight_2 = 1 - \ weight_1$$

Equation 2: weight 2 calculations.

$Weight_1$ is a value that we need to calculate, it goes from 0 to 1. The next statements we will see how we calculate the weight for different purposes. All the applications we need to define a maximum value, that value can be defined by the user or calculated inside the simulation.

5.12.2.1. Color over lifetime

The particle will change the color over his lifetime. We need to know the max lifetime which the particle will be spawned at the simulation.

$$weight_1 = \max lifetime \div current\ particle\ lifetime$$

Equation 3: Weight from lifetime.

5.12.2.2. Color over distance

The distance will be the factor that change the color of the particle, that we need to define the max distance and get the distance from the particle to his spawn.

$$weight_1 = \max distance \div (particle\ global\ position - particle\ system\ position)$$

Equation 4: Weight from distance.

5.12.2.3. Color over speed

It is different because we can define the max speed, but the particles are moving in 3d then their speed is a vector. Then, we need to get the average value from that vector.

$$weight_1 = \left(\frac{|speed_x| + |speed_y| + |speed_z|}{3}\right) \div max\ speed$$

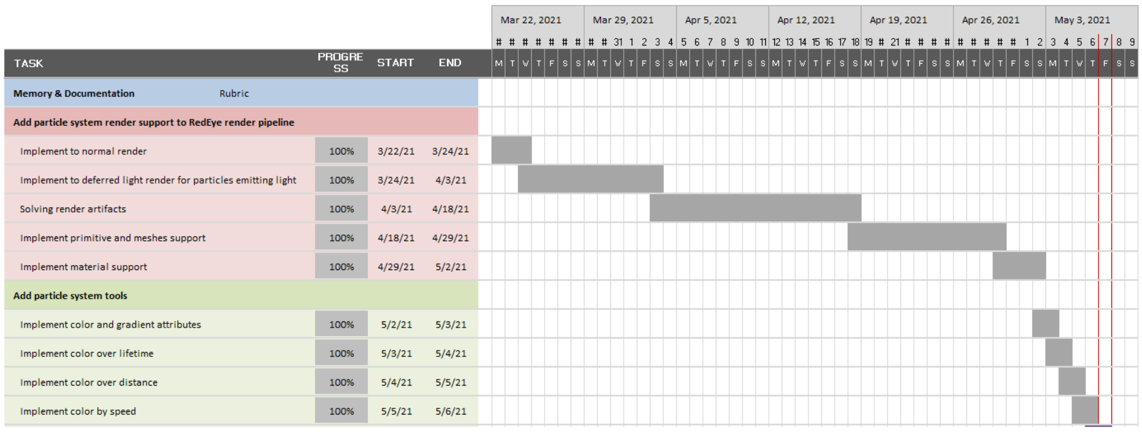
Equation 5: Weight from speed.

That we inverted the division we need to swap the color position from the main gradient formula.

5.13. Progress iteration

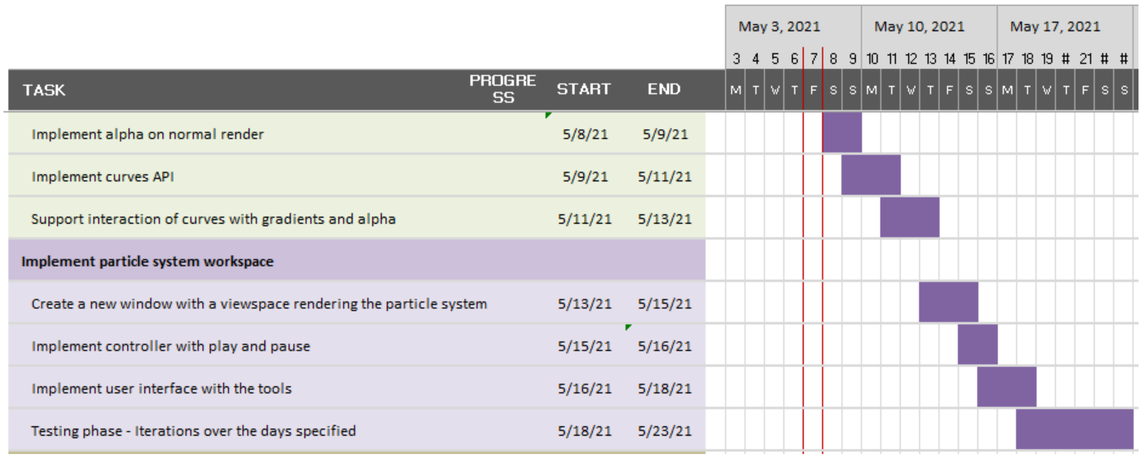
Because we go in depth on shader memory for obtaining a good result of lights, at 5.7, and bugs appears like the artifacts when we test it on laptops, at 5.7.2, with the primitive resource system, at 5.9.2, it took more time than the expected.

We do a Gantt changes reducing the days to some tasks, the important reductions are the testing workspace for particle system, 15 days reduced, the webpage content, 13 days reduced, and the project release, 3 days reduced. Also, we introduce a new task for implementing the particle system resource, for 3 days, mentioned at 5.11.



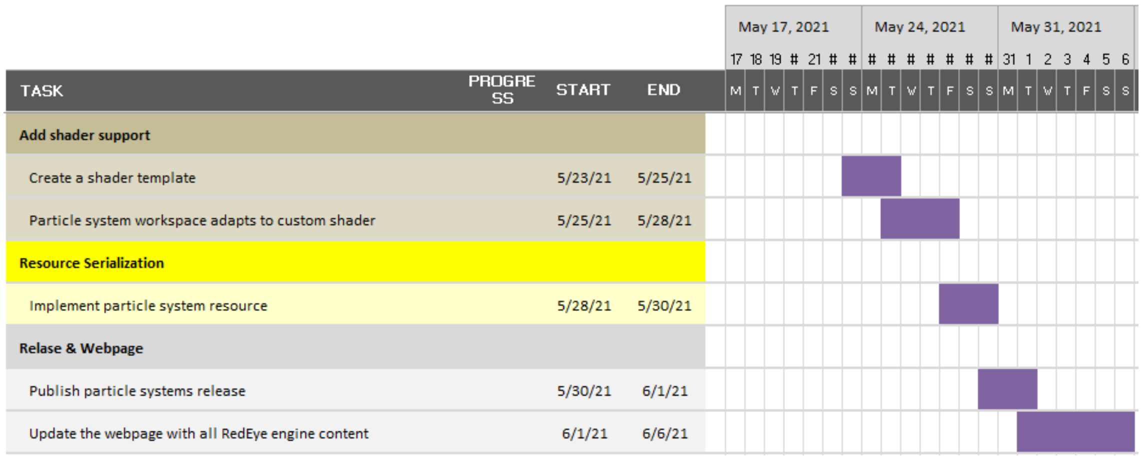
TASK	PROGRESS	START	END
Memory & Documentation	Rubric		
Add particle system render support to RedEye render pipeline			
Implement to normal render	100%	3/22/21	3/24/21
Implement to deferred light render for particles emitting light	100%	3/24/21	4/3/21
Solving render artifacts	100%	4/3/21	4/18/21
Implement primitive and meshes support	100%	4/18/21	4/29/21
Implement material support	100%	4/29/21	5/2/21
Add particle system tools			
Implement color and gradient attributes	100%	5/2/21	5/3/21
Implement color over lifetime	100%	5/3/21	5/4/21
Implement color over distance	100%	5/4/21	5/5/21
Implement color by speed	100%	5/5/21	5/6/21

Table 12: Gantt iteration 1.



TASK	PROGRESS	START	END
Implement alpha on normal render		5/8/21	5/9/21
Implement curves API		5/9/21	5/11/21
Support interaction of curves with gradients and alpha		5/11/21	5/13/21
Implement particle system workspace			
Create a new window with a viewspace rendering the particle system		5/13/21	5/15/21
Implement controller with play and pause		5/15/21	5/16/21
Implement user interface with the tools		5/16/21	5/18/21
Testing phase - Iterations over the days specified		5/18/21	5/23/21

Table 13: Gantt iteration 2.



TASK	PROGRESS	START	END
Add shader support			
Create a shader template		5/23/21	5/25/21
Particle system workspace adapts to custom shader		5/25/21	5/28/21
Resource Serialization			
Implement particle system resource		5/28/21	5/30/21
Relase & Webpage			
Publish particle systems release		5/30/21	6/1/21
Update the webpage with all RedEye engine content		6/1/21	6/6/21

Table 14: Gantt iteration 3.

On the one hand, the approve of those changes are that our product will work on different hardware. It is important to test on different computers for detecting compatibility problems and solve it. Furthermore, there are not more days added to the development and it means that the costs will not be changed.

On the other hand, the disadvantage is that we are going to do less iterations for testing the workspace and the webpage will contains the basic content without doing big design improves.

5.14. Curve implementation

We studied and trying to combine with an ImGui curve editor with an API called tinyspline but it is not what we want because that library only modify the giving curves, do not give the value which we sent from the color calculations. (*Msteinbeck/Tinyspline: ANSI C Library for NURBS, B-Splines, and Bézier Curves with Interfaces for C++, C#, D, Go, Java, Lua, Octave, PHP, Python, R, and Ruby.*, n.d.)

Then, we tested different curve editor made by the community from a post at the GitHub issues. After that, we select the editor that it will be useful for that implementation. (*Cubic Bezier Widget / Curve Editor · Issue #786 · Ocornut/Imgui*, n.d.)

The curve that we use it works with a giving points and the user can do a custom curve or use templates that defined its widget. Then we pass the calculated weight curve to a method inside the widget that returns the value from the curve. We can use the normal method that use the literal curve or the smooth method which transform the giving transform smoother (the grey curve from the giving image).

We use the curve at the color calculations, the giving weight of different types, like color over lifetime, will pass to the curve method and we obtain a new weight.

We introduced more implementations with the curve like the user can modify the size of the curve editor, the number of points, use the smooth calculations and modify the alpha with the curve, which is more explained at next point.

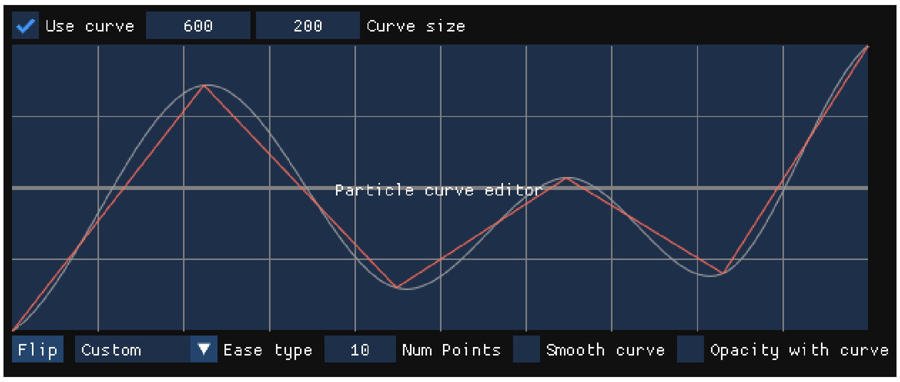


Figure 39: Curve editor.

5.15. Alpha on normal render

We send a value at the shader from 0 to 1 that specifies the particle opacity. It only works on normal render because on deferred render use planes with textures for display the scene and there not are depth. Also, we link the opacity with the curve, that means when the user selects one of the color configurations the weight value will be the alpha.

Figure 40: Opacity at editor.

5.16. Primitive point

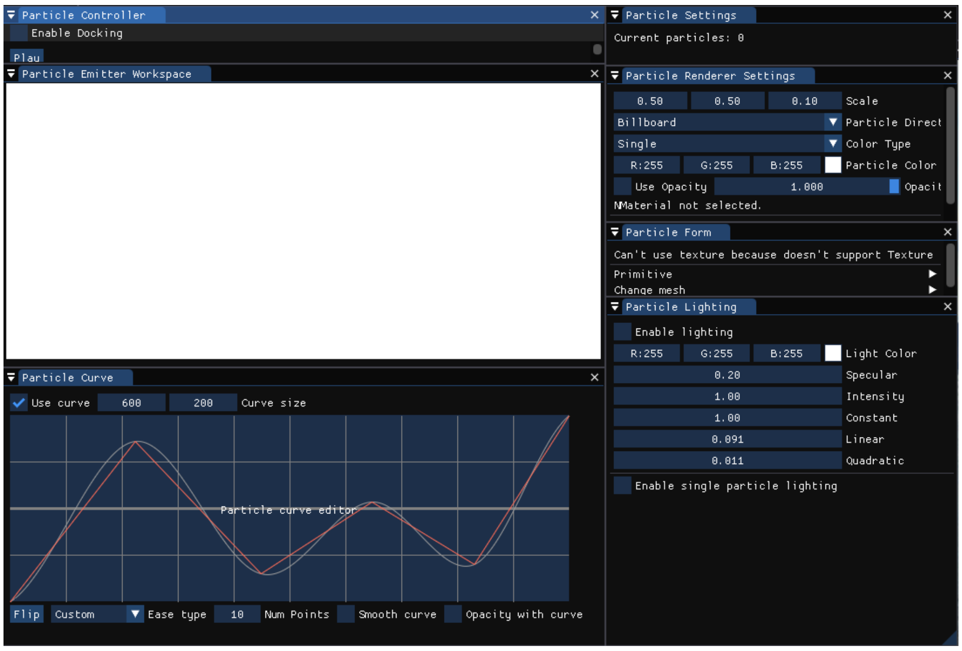
We deleted the mesh buffers of a point inside the particle emitter component and we create a new primitive component of point, which it works with the primitive resource system.

5.17. First workspace iteration

5.17.1. Basic Structure

We separate all the configurations from the editor with different windows, then using the docking feature from ImGui we group those windows to build a workspace.

Then we create a viewspace for only display the particle emitter. It contains a camera and framebuffers objects. Finally, the renderer fill the buffers and it is use at the workspace. There is a little problem that the background of the viewspace is white and there can not change to another color, we tried to fix that problem but for now we do not find the solution.



```
Use curve   600   200   Curve size
Custom ▼ Ease type   10 Num Points   Smooth curve   Opacity with curve

Current particles: 0

0.50   0.50   0.10   Scale
Billboard            Particle Direct
Single               Color Type
R:255  G:255  B:255  Particle Color
Use Opacity   1.000  Opacit
NMaterial not selected.

Can't use texture because doesn't support Texture

Enable lighting
R:255  G:255  B:255  Light Color
0.20                 Specular
1.00                 Intensity
1.00                 Constant
0.091                Linear
0.011                Quadratic
Enable single particle lighting
```

Figure 41: Particle emitter workspace.

5.17.2. Docking

There are some docking problems, the workspace can dock inside any little window at the engine. Then when we move to fix, only one window will move.

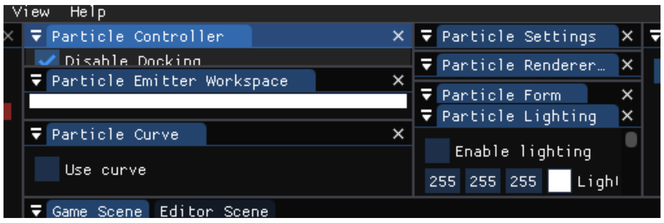
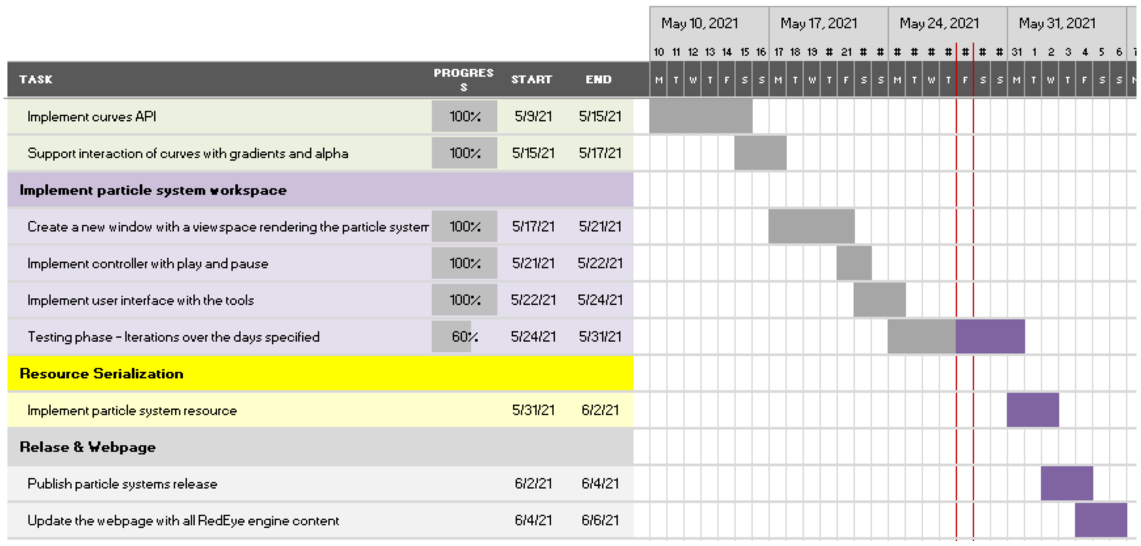


Figure 42: Docking problem.

The solution is disabling the docking on the workspace, and it cannot be placed on any other window, but one problem is the user can drag one tab of any window forming the workspace and unlink from it. Then, we add a checkbox for enable docking and link the window again and the user can move the window at the workspace again or make some modifications.

5.18. Last progress iteration

We decided to quit the shader support implementation because we are using the most common renderer tools for particles at the editor, and we prefer giving more time implementing the editor than add shader support.



TASK	PROGRESS	START	END
Implement curves API	100%	5/9/21	5/15/21
Support interaction of curves with gradients and alpha	100%	5/15/21	5/17/21
Implement particle system workspace			
Create a new window with a viewspace rendering the particle system	100%	5/17/21	5/21/21
Implement controller with play and pause	100%	5/21/21	5/22/21
Implement user interface with the tools	100%	5/22/21	5/24/21
Testing phase - Iterations over the days specified	60%	5/24/21	5/31/21
Resource Serialization			
Implement particle system resource		5/31/21	6/2/21
Relase & Webpage			
Publish particle systems release		6/2/21	6/4/21
Update the webpage with all RedEye engine content		6/4/21	6/6/21

Table 15: Final Gantt iteration

We take more time at the curve implementation because we search APIs and testing curves made by the ImGui community. Also, we took longer the implementation for the viewport cause of adding their functionalities at the renderer module and the input of the camera.

Finally, we add more time for the testing phase and reducing the webpage task.

5.19. Implement particle system resource

5.19.1. RedEye´s resource system

5.19.1.1. Resource structure

The base class of resource is a container that store the main information and important methods, then that class inheritance to each specific resources with their specific implementations. That class contains:

- MD5 is the resource identification where is based by the asset file.
- Paths (asset, library and meta)
- Control variables like is in memory, internal (from engine) …

Also, it has defined some virtual methods that each resource uses it for the same purpose but different way. There are like load or unload in memory where reads the information store on hard drive, the meta file store the information from resource base and maybe some resources need to save more info.

The resource serializes as two formats, first json on meta files and if we save it on assets, because on assets folder there are the files which is readable for the user. The other format is memory serialization where it is on library folder, that is the fastest way for loading the resource, but it is not legible.

The MD5 is stored as a pointer and at the same time is the identification of all resources at engine runtime.

The moment that the resource loads in memory, it will check if exits the library file, if do not exits it will load first the asset file and then serialized it on library.

To avoid serialize every time the resource from any component, our entity component system serialization before save checks if there are some resources using, serializes and give one identification for the components and they save it.

5.19.1.2. Resource manager

The resource manager handles all the resources inside a map, where the key is the MD5 pointer, and the content is the resource.

There is a reference counting that manages resources if they need to load or unload: The components use or unuse their referenced resources to count from the resources manager how times it is being referenced as to not allocate its memory more than once.

There are some methods for find resources by giving paths or md5, if it does not return the resource, it means that is not referenced at the resource manager.

The user can delete any resource as wish, the resource manager checks all the resources and the scene that will be affected by the deletion and shows a popup showing the resources affected. If the user deletes, it will erase it from the other resources (like a texture or shader on material) and save with the changes.

5.19.1.3. Read assets changes

The filesystem owns a self-directory system thar replicates the asset folder with the information of each file and resources. ReadAssetsChanges is the method that manage our directory system detecting new files, importing new resources and changes for hot reloading.

The engine initialization runs it entirely for creating all the directory system and imports the new resources detected, which do not have meta file. Then, every update that method is called but giving a time that will stop and continue at next frame when the process time is long, with that we can avoid low framerate peaks.

5.19.1.4. Panel assets and panel inspector

The panel assets show own directory system based on assets folder, there are all the resources with thumbnails, and it is the place where the user can delete the resource. Finally, when we click at one resource, the properties will display at the inspector panel.

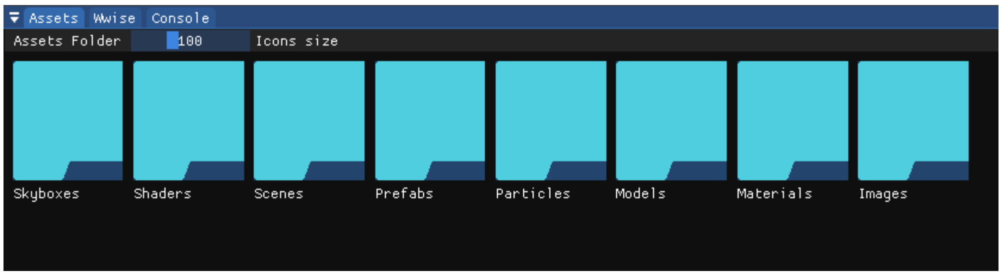
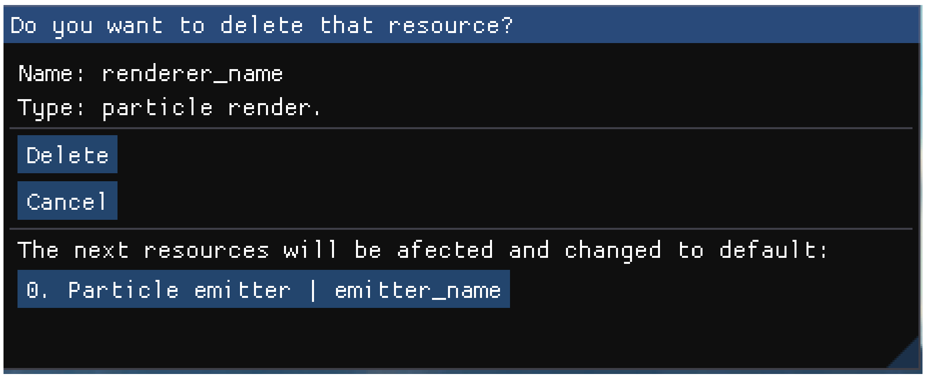


Figure 43: Panel assets

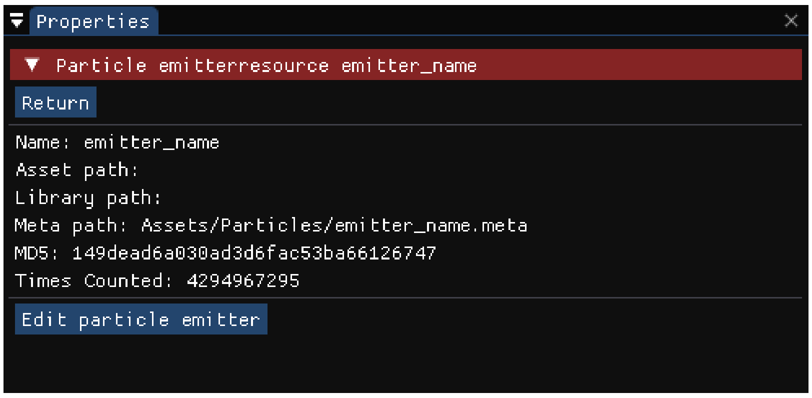


Do you want to delete that resource?

Name: renderer_name
Type: particle render.

The next resources will be afected and changed to default:
0. Particle emitter | emitter_name

Figure 44: Delete resource popup



```
Name: emitter_name
Asset path:
Library path:
Meta path: Assets/Particles/emitter_name.meta
MD5: 149dead6a030ad3d6fac53ba66126747
Times Counted: 4294967295
```

Figure 45: Resource properties

5.19.2. Particle emitter resource structure

The particle emitter resource, which the particle workspace generates that resource, is like a container that handles two sub resources:

- Particle emission resource.
- Particle render resource.

Then the main resource only will be a meta file which contains the references of the sub resources, we do not need to do asset and library save. The MD5 generated is the combination of the MD5 from sub resources.

The component particle emitter will contain the md5 reference of the particle emitter resource and when we use resources it will load the resource on memory and obtain a simulation by the combination of the two sub resources from the particle emitter resource.

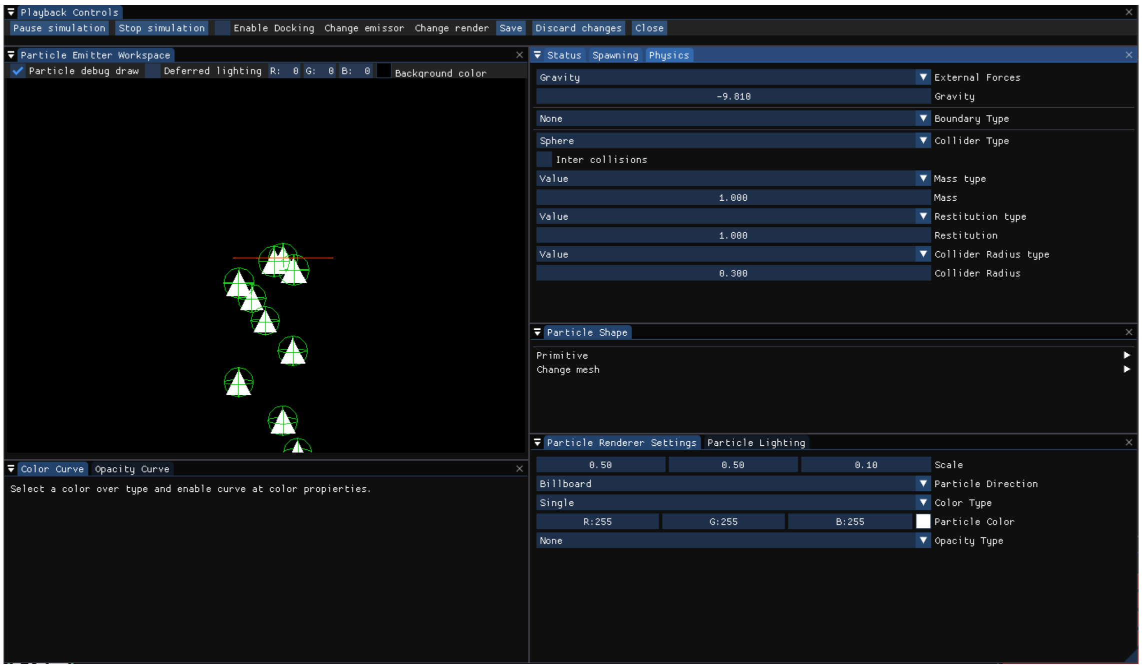
The main advantages are that the component does not serializes all the particle emitter information and with the same resource can be used on different components and the why we separate the emission, and the render variables is that we can create different emitters sharing the same render or emission.

Finally, the disadvantage of implement a resource with all functionalities requires more time than expected for coding on a great number of different files, because we do a complete usability with the resource, and test the resource generation, modification, hot reloading, and deletion.

The next statement we explain the resource implementation on the workspace.

5.20. Final workspace iteration

We mention that the testing of the user interface from the particle workspace was informal, where was a user testing, sharing the screen and talking about the implementation. We need to know that is necessary to do a specialized test, as we mention on methodology.



```
✓ Particle debug draw    Deferred lighting  R: 0  G: 0  B: 0    Background color

Gravity                          External Forces
-9.810                           Gravity
None                             Boundary Type
Sphere                           Collider Type
  Inter collisions
Value                            Mass type
1.000                            Mass
Value                            Restitution type
1.000                            Restitution
Value                            Collider Radius type
0.300                            Collider Radius

0.50     0.50     0.10           Scale
Billboard                        Particle Direction
Single                           Color Type
R:255    G:255    B:255          Particle Color
None                             Opacity Type

Select a color over type and enable curve at color propierties.
```

Figure 46: Last iteration particle emitter workspace

5.20.1. Viewport solution

We solve the problem mentioned at first iteration that the background color is always white.

The problem was that we call first glClear, which reset the framebuffers, and then glClearColor. We only need to invert the order call because the glClearColor specifies the color that glClear will use, do not color the buffer. (*GlClear - OpenGL 4 Reference Pages*, n.d.) (*GlClearColor - OpenGL 4 Reference Pages*, n.d.)

5.20.2. Resource tools

The user can open the workspace empty for creating a new particle emitter or open it from generated resource. We add new user interface for interacting with the resource:

- Save: the user saves the particle emitter and the sub resources, if any of them are new they need to introduce a name.
- Discard changes: avoid save and returns to the initial state of the emitter.
- Select sub resources: can use sub resources that was created before.

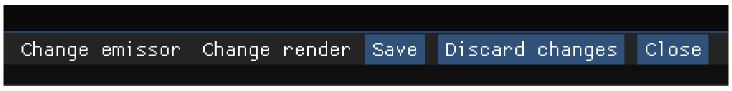


Figure 47: Particle resource tools

The workspace detects any changes made for knowing that the resource needs to be save. When saves appears a popup window that the user writes the names of the new resources and confirm the save, also there are a security purposes like to no generate a resource with the same name and when the user closes the workspace or opens another resource emitter for editing, it will appear the popup save if there are any changes.

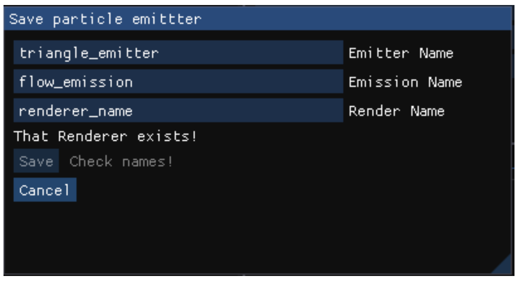

Save particle emittter
triangle_emitter Emitter Name
flow_emission Emission Name
renderer_name Render Name
That Renderer exists!
Save Check names!
Cancel

Figure 48: Particle emitter popup save

5.20.3. Viewport implementations

We add new functionalities to the displaying particle emitter at the workspace. The user has direct inputs for change few render options:

- Can enable debug draw for showing the simulation debug.
- Can change deferred or normal render.
- Can change background color if not deferred render.

Finally, the focus works fine thanks to use the bounding box inside the simulation.

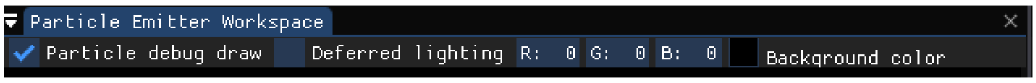


Figure 49: Particle renderer options

5.21. Particle render profiling

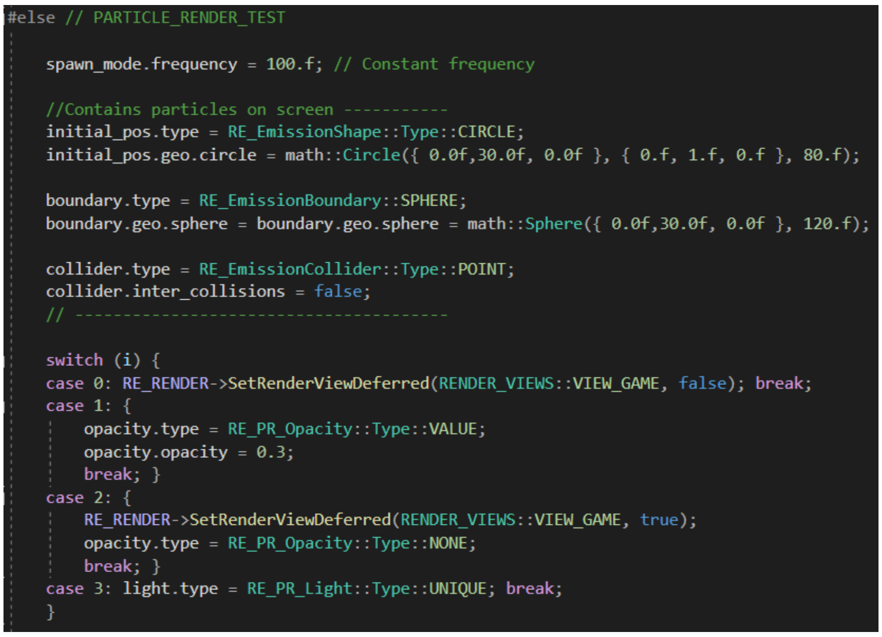
RedEye engine uses an own profiling, which records every frame the time of each macro we defined on the code and save it as a json file.

Then, we defined four simulations of rendering for profiling, it changes to the next simulation when the rendering costs 33 milliseconds.

We setup a basic emission configuration for keeping all the particles on screen, a circle boundary and constant frequency of spawning particles. Then, different render types:

- Normal render.
- Normal render with opacity.
- Deferred render without lights,
- Deferred render with lights.

The profiler records on the particle manager, where is define the render method of a particle system and the module render where we upload and draw the lights.



```
#else // PARTICLE_RENDER_TEST

    spawn_mode.frequency = 100.f; // Constant frequency

    //Contains particles on screen -----------
    initial_pos.type = RE_EmissionShape::Type::CIRCLE;
    initial_pos.geo.circle = math::Circle({ 0.0f,30.0f, 0.0f }, { 0.f, 1.f, 0.f }, 80.f);

    boundary.type = RE_EmissionBoundary::SPHERE;
    boundary.geo.sphere = boundary.geo.sphere = math::Sphere({ 0.0f,30.0f, 0.0f }, 120.f);

    collider.type = RE_EmissionCollider::Type::POINT;
    collider.inter_collisions = false;
    // --------------------------------------

    switch (i) {
    case 0: RE_RENDER->SetRenderViewDeferred(RENDER_VIEWS::VIEW_GAME, false); break;
    case 1: {
        opacity.type = RE_PR_Opacity::Type::VALUE;
        opacity.opacity = 0.3;
        break; }
    case 2: {
        RE_RENDER->SetRenderViewDeferred(RENDER_VIEWS::VIEW_GAME, true);
        opacity.type = RE_PR_Opacity::Type::NONE;
        break; }
    case 3: light.type = RE_PR_Light::Type::UNIQUE; break;
    }
```

Figure 50: Setup profiling particle render

The next graphic shows our profiling results, and we can define that the quantity of particles affects the performance as linear over the quantity of particles. The quantity of particles drawing is 4000. However, when we do particle lighting, where the max particles lights are 508 and then keeps spawning but without light, the performance lows significantly, almost the half from the other tests.

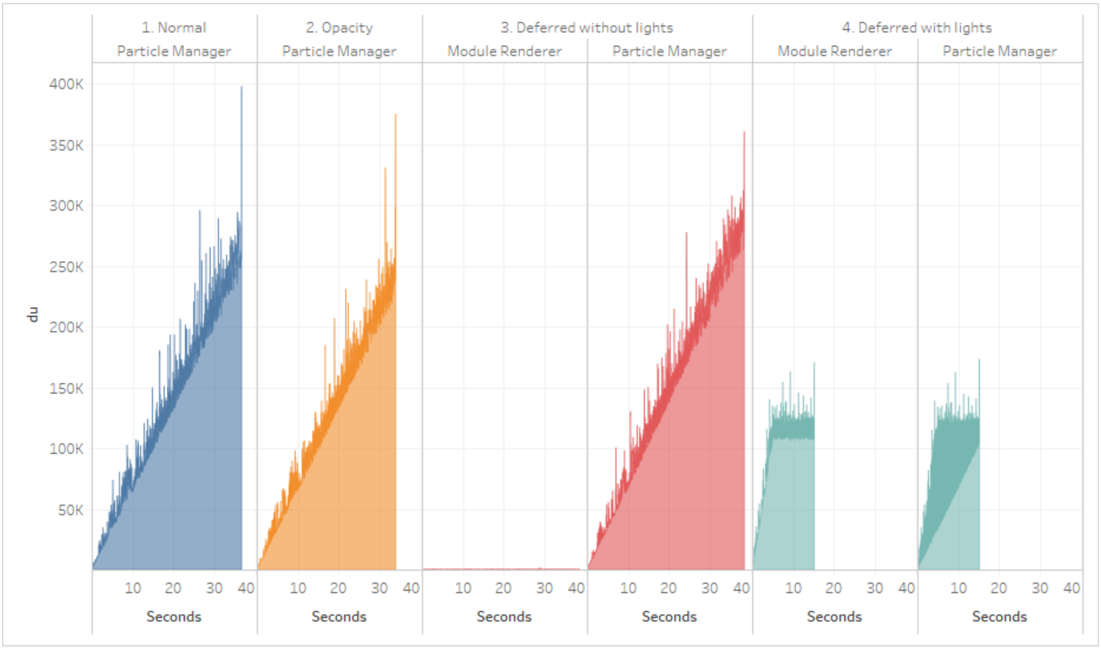


Figure 51: Particle render graphics results

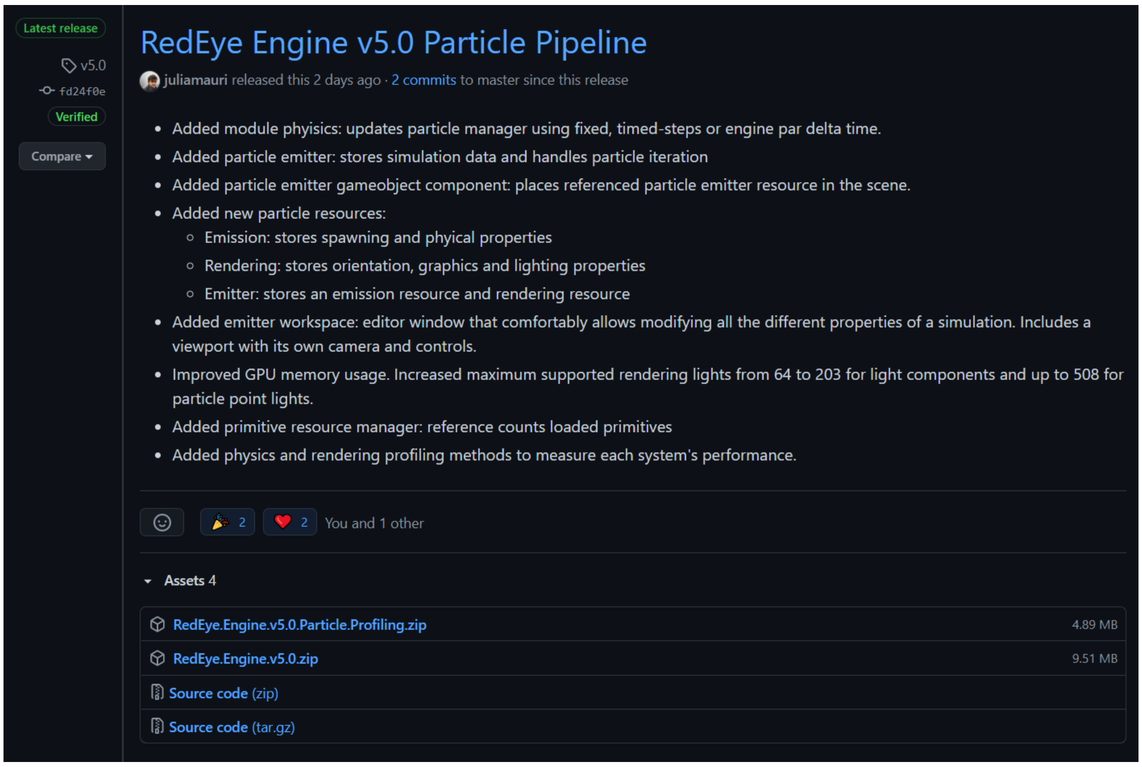
5.22. Publishing project

We start to publish the project when the development ends, and we use GitHub for publishing a release and we write all the information of that system on the repository Wiki and our custom webpage.

5.22.1. GitHub Relsease & Wiki

The release is the version 5.0 and contains the dynamic link libraries files from some third-party code, the engine binaries, engine data and the ImGui file that contains the user interface data and a file that we describe our project, which is name readme.md and contains a brief description of RedEye engine, links, versions, user actions, description of the user interface with systems and mention of all external libraries we used.

Also, we upload another compressed folder that contains the engine profiling binaries, tableau workspaces and our profile results. Everyone can do the same profile as we did on their computer.



Latest release
v5.0
fd24f0e
Verified
Compare

RedEye Engine v5.0 Particle Pipeline

juliamauri released this 2 days ago · 2 commits to master since this release

- Added module phyisics: updates particle manager using fixed, timed-steps or engine par delta time.
- Added particle emitter: stores simulation data and handles particle iteration
- Added particle emitter gameobject component: places referenced particle emitter resource in the scene.
- Added new particle resources:
  - Emission: stores spawning and phyical properties
  - Rendering: stores orientation, graphics and lighting properties
  - Emitter: stores an emission resource and rendering resource
- Added emitter workspace: editor window that comfortably allows modifying all the different properties of a simulation. Includes a viewport with its own camera and controls.
- Improved GPU memory usage. Increased maximum supported rendering lights from 64 to 203 for light components and up to 508 for particle point lights.
- Added primitive resource manager: reference counts loaded primitives
- Added physics and rendering profiling methods to measure each system's performance.

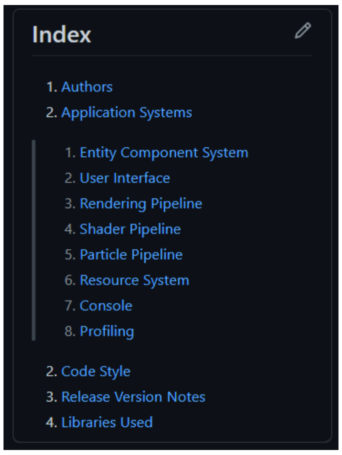
2 2 You and 1 other

Assets 4

RedEye.Engine.v5.0.Particle.Profiling.zip	4.89 MB
RedEye.Engine.v5.0.zip	9.51 MB
Source code (zip)	
Source code (tar.gz)	

Figure 52: Particle Pipeline release

The GitHub Wiki is an extender readme.md, we explain each system detailed and a code style, which we define how we code and commit on the repository. This is essential for the community that will help to our project following some instructions and rules for avoiding mess on the repository.



Index

1. Authors
2. Application Systems
   1. Entity Component System
   2. User Interface
   3. Rendering Pipeline
   4. Shader Pipeline
   5. Particle Pipeline
   6. Resource System
   7. Console
   8. Profiling
2. Code Style
3. Release Version Notes
4. Libraries Used

Figure 53: RedEye wiki index table

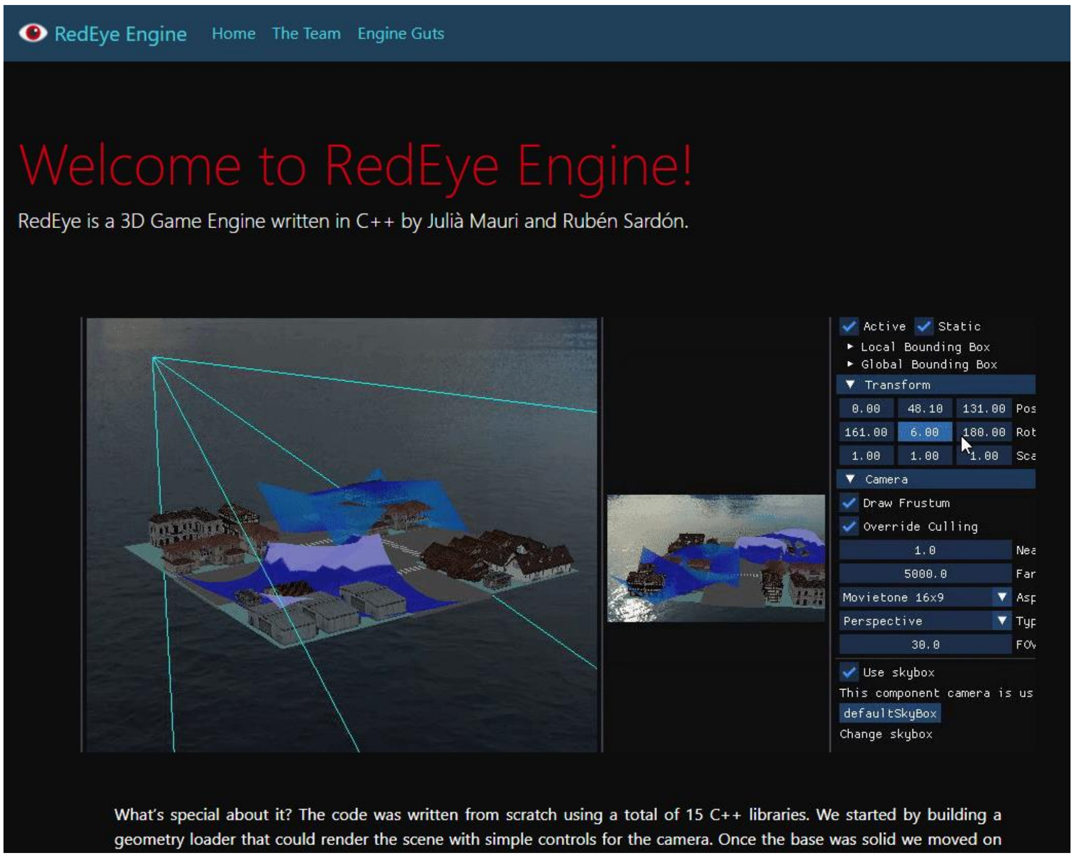
5.22.1. Webpage

The webpage is the external and visual view from our repository. It contains all the major information write on wiki with video and images. It is a good way for externalize our project to a target that does not into GitHub development.

The code of the webpage is html with CSS using a library named bootstrap, where contains elements for build the webpage and it is responsive for different displays. (*Bootstrap · The Most Popular HTML, CSS, and JS Library in the World.*, n.d.)

The map of our webpage is:

- Home: brief intro of our project with lasts implementations.
- The Team: about the developers that makes this product.
- Engine Guts (systems explanations): like the wiki with more media.



Welcome to RedEye Engine!

RedEye is a 3D Game Engine written in C++ by Julià Mauri and Rubén Sardón.

What's special about it? The code was written from scratch using a total of 15 C++ libraries. We started by building a geometry loader that could render the scene with simple controls for the camera. Once the base was solid we moved on

Figure 54: RedEye Engine webpage

It is optimized with search engine optimization "SEO" for gaining a good positioning on search engines. Currently if we mix RedEye with engine or c++ words, the webpage position is on first position.

6. Conclusions

We improved RedEye Engine with a particle pipeline during three months of development. We focused on solving technical graphics programming and implementing the resource and were not able to reach the user interface testing stage.

The implementation of the particle workspace has tons of different options for editing ranging from emission to rendering, internal profiling testing, debug drawing, and resource engine usability. Engine resource workings include hot-reloading, resource management (delete and generation), and use of sub resources using different emitters to avoid allocating more memory than needed on the hard drive.

Starting with a basic render particle implementation and later adding color and opacity with a curve editor, using meshes or primitives, which we can set their direction, and particle lighting with our deferred light pipeline. We got more into on technical implementations like GPU memory optimization and how the GPU needs to be synchronized before render to understand some calls of OpenGL.

We wrote a wiki where we defined all the engine's systems and how the code operates. These published resources are available for everyone that would participate implementing new functionalities on their own project. Also, the engine's webpage externalizes our product for a target that is not into programming development.

Through this development we improved our programming, understand how the GPU's intricacies worked and analyzed profiling results to learn how our implementations affected the engine's performance.

We were two programmers coding at the same files during all the development, working side-by-side on complicated implementations and avoiding stepping on each other. We planned the best way to procure our implantations and adapted through our development. Because we did technical and resource implementations, our monthly development time increases to 200 hours from the starting 120 hours, we increased 2.5 hours every day, and the budget raised to 8.295 € from an initial of 6.375 €.

7. Future Projects

We will continue with the engine development. It is an amazing project through which we can implement systems and keep growing the engine. We improve our skills and can share our approach with fellow engine developers who could be interested in implementing their own particle pipeline.

The next big pending implementation is a multi-renderer system. Now we only use OpenGL, but we wish to implement DirectX and Vulkan with all the engine's rendering implementations.

Also, we want to make our engine multiplatform, starting with Linux, and 64 bits systems support. If we understand how multithreading works and implement it on all our systems, the performance will increase significantly.

Finally, it is important to finish the most common game engine functionalities like game mode, which we can make game builds without displaying the engine, and scripting, which is the system that will link everything together. In conclusion, RedEye Engine is an endless project we love to work on over the years.

7. Bibliography

ARB_compute_shader. (n.d.). Retrieved March 19, 2021, from https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_compute_shader.txt

Beam Modules | Unreal Engine Documentation. (n.d.). Retrieved March 19, 2021, from https://docs.unrealengine.com/en-US/RenderingAndGraphics/ParticleSystems/Reference/Modules/Beam/index.html

Bootstrap · The most popular HTML, CSS, and JS library in the world. (n.d.). Retrieved June 27, 2021, from https://getbootstrap.com/

Cascade Particle Systems | Unreal Engine Documentation. (n.d.). Retrieved March 19, 2021, from https://docs.unrealengine.com/en-US/RenderingAndGraphics/ParticleSystems/index.html

Color Modules | Unreal Engine Documentation. (n.d.). Retrieved March 19, 2021, from https://docs.unrealengine.com/en-US/RenderingAndGraphics/ParticleSystems/Reference/Modules/Color/index.html

Compute Particles Sample. (n.d.). Retrieved March 19, 2021, from https://docs.nvidia.com/gameworks/content/gameworkslibrary/graphicssamples/opengl_samples/computeparticlessample.htm

Compute Shader - OpenGL Wiki. (n.d.). Retrieved March 19, 2021, from https://www.khronos.org/opengl/wiki/Compute_Shader

*Crash: Instanciate max lights (64 lights) * 3 times · Issue #4 · juliamauri/RedEye-Engine*. (n.d.). Retrieved May 7, 2021, from https://github.com/juliamauri/RedEye-Engine/issues/4

Cubic Bezier widget / Curve editor · Issue #786 · ocornut/imgui. (n.d.). Retrieved May 28, 2021, from https://github.com/ocornut/imgui/issues/786

Discord | Your Place to Talk and Hang Out. (n.d.). Retrieved March 19, 2021, from https://discord.com/

Essential Material Concepts | Unreal Engine Documentation. (n.d.). Retrieved March 19, 2021, from https://docs.unrealengine.com/en-US/RenderingAndGraphics/Materials/IntroductionToMaterials/index.html

GitHub. (n.d.). Retrieved March 19, 2021, from https://github.com/

glClear - OpenGL 4 Reference Pages. (n.d.). Retrieved June 27, 2021, from https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glClear.xhtml

glClearColor - OpenGL 4 Reference Pages. (n.d.). Retrieved June 27, 2021, from https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glClearColor.xhtml

Google Drive. (n.d.). Retrieved March 19, 2021, from https://drive.google.com/

Interface Block (GLSL) - OpenGL Wiki. (n.d.). Retrieved March 19, 2021, from https://www.khronos.org/opengl/wiki/Interface_Block_(GLSL)

*juliamauri/GetOpenGLInfo: Generate a *.txt file with your hardware info and Gl integers*. (n.d.). Retrieved March 19, 2021, from https://github.com/juliamauri/GetOpenGLInfo

juliamauri/RedEye-Engine: Engine 3D(OpenGL). (n.d.). Retrieved March 19, 2021, from https://github.com/juliamauri/RedEye-Engine

msteinbeck/tinyspline: ANSI C library for NURBS, B-Splines, and Bézier curves with interfaces for C++, C#, D, Go, Java, Lua, Octave, PHP, Python, R, and Ruby. (n.d.). Retrieved May 28, 2021, from https://github.com/msteinbeck/tinyspline

Particle Lights | Unreal Engine Documentation. (n.d.). Retrieved March 19, 2021, from https://docs.unrealengine.com/en-US/RenderingAndGraphics/ParticleSystems/ParticleLights/index.html

Particle System User Guide | Unreal Engine Documentation. (n.d.). Retrieved March 19, 2021, from https://docs.unrealengine.com/en-US/RenderingAndGraphics/ParticleSystems/UserGuide/index.html

Post Process Effects | Unreal Engine Documentation. (n.d.). Retrieved March 19, 2021, from https://docs.unrealengine.com/en-US/RenderingAndGraphics/PostProcessEffects/index.html

Project management for game development - HacknPlan. (n.d.). Retrieved March 19, 2021, from https://hacknplan.com/

RedEye Engine. (n.d.). Retrieved March 19, 2021, from https://www.redeye-engine.es/

Segal, M., Akeley, K., Frazier, C., Leech, J., & others. (2019). The OpenGL Graphics System: A Specification (Version 4.6 (Core Profile)). *Graphics System*, 852. https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.withchanges.pdf

Shader Storage Buffer Object - OpenGL Wiki. (n.d.). Retrieved March 19, 2021, from https://www.khronos.org/opengl/wiki/Shader_Storage_Buffer_Object

Shared Output Settings and Properties | Visual Effect Graph | 10.4.0. Particle Options Settings. (n.d.). Retrieved March 19, 2021, from https://docs.unity3d.com/Packages/com.unity.visualeffectgraph@10.4/manual/Context-OutputSharedSettings.html

Software de análisis e inteligencia de negocios. (n.d.). Retrieved March 19, 2021, from https://www.tableau.com/

SubUV Modules | Unreal Engine Documentation. (n.d.). Retrieved March 19, 2021, from https://docs.unrealengine.com/en-US/RenderingAndGraphics/ParticleSystems/Reference/Modules/SubUV/index.html

Unity - Manual: Choosing your particle system solution. (n.d.). Retrieved March 19, 2021, from https://docs.unity3d.com/Manual/ChoosingYourParticleSystem.html

Unity - Manual: Color By Speed module. (n.d.). Retrieved March 19, 2021, from https://docs.unity3d.com/2021.1/Documentation/Manual/PartSysColorBySpeedModule.html

Unity - Manual: Color Over Lifetime module. (n.d.). Retrieved March 19, 2021, from https://docs.unity3d.com/2021.1/Documentation/Manual/PartSysColorOverLifeModule.html

Unity - Manual: Particle System. (n.d.). Retrieved March 19, 2021, from https://docs.unity3d.com/2021.1/Documentation/Manual/class-ParticleSystem.html

Unity - Manual: Particle System Main module. (n.d.). Retrieved March 19, 2021, from https://docs.unity3d.com/2021.1/Documentation/Manual/PartSysMainModule.html

Unity - Manual: Renderer module. (n.d.). Retrieved March 19, 2021, from https://docs.unity3d.com/2021.1/Documentation/Manual/PartSysRendererModule.html

Unity - Manual: Using the Built-in Particle System. (n.d.). Retrieved March 19, 2021, from https://docs.unity3d.com/2021.1/Documentation/Manual/PartSysUsage.html

Unity - Manual: Visual Effect Graph. (n.d.). Retrieved March 19, 2021, from https://docs.unity3d.com/2021.1/Documentation/Manual/VFXGraph.html

VFX Optimization Guide | Unreal Engine Documentation. (n.d.). Retrieved March 19, 2021, from https://docs.unrealengine.com/en-US/RenderingAndGraphics/ParticleSystems/Optimization/index.html

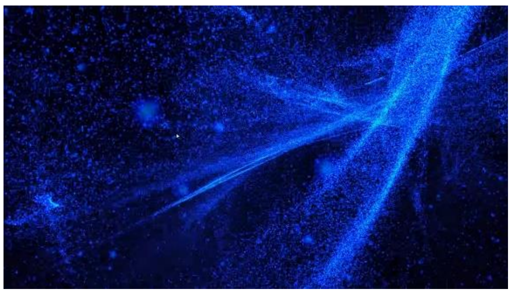
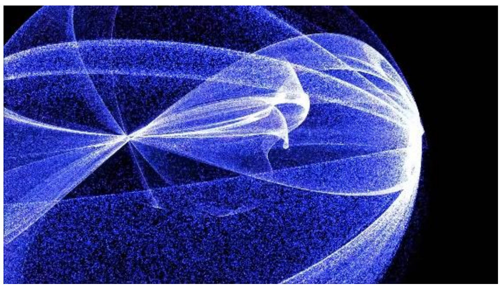
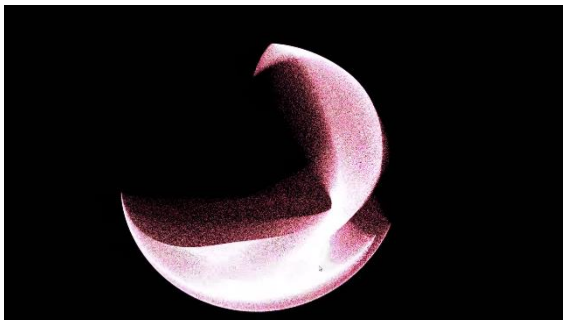
Visual Effect Graph | Visual Effect Graph | 10.4.0. (n.d.). Retrieved March 19, 2021, from https://docs.unity3d.com/Packages/com.unity.visualeffectgraph@10.4/manual/index.html

Visual Effect Graph Logic | Visual Effect Graph | 10.4.0. (n.d.). Retrieved March 19, 2021, from https://docs.unity3d.com/Packages/com.unity.visualeffectgraph@10.4/manual/GraphLogicAndPhilosophy.html

Visual Studio IDE, Code Editor, Azure DevOps, & App Center - Visual Studio. (n.d.). Retrieved March 19, 2021, from https://visualstudio.microsoft.com/

8. Annexes

8.1. Scratch of Compute Shader - 2000000 particles

Stainislaw Eppinger. Particle System using compute shader in OpenGL (n.d.). Retrieved March 19, 2021, from http://www.youtube.com/watch?v=jwCAsyiYimY