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

Node-Based Particle System

Degree in Video Games Design and Development

Author: *Aitor Luque Bodet*

Director: *Jesús Díaz García*

Year: *2021-22*

University: *CITM*

Index

Abstract ---- 4

Keywords ---- 4

Links ---- 5

Index of Tables ---- 6

Index of Figures ---- 7

Glossary ---- 9

1. Introduction ---- 11

1.1 Motivation ---- 11

1.2 Problem presentation ---- 12

1.3 General Objectives ---- 13

1.4 Specific Objectives ---- 14

1.5 Project Scope ---- 15

2. State of the art ---- 16

2.1 Houdini ---- 16

2.2 Unity VFX Graph ---- 17

2.3 Blender ---- 19

2.4 Unreal Engine ---- 20

2.5 Conclusion ---- 22

3. Project Management ---- 23

3.1. GANTT ---- 23

3.2. GANTT Update ---- 26

3.3. SWOT ---------- 27

3.4. Risks and Contingency Plan ---------- 28

3.5. Cost Analysis ---------- 31

4. Methodology ---------- 33

4.1 Pre-Production ---------- 33

4.2 Production ---------- 34

4.3 Post-Production ---------- 36

5. Project Development ---------- 37

5.1 ASE - Another Small Engine ---------- 37

5.2 Adapting the engine ---------- 42

5.3 Particle System ---------- 44

5.4 Rendering and Instancing ---------- 51

5.5 Node Editor ---------- 54

5.6 Node Editor Window ---------- 65

6. Conclusions ---------- 73

7. Next Steps ---------- 76

8. Bibliography ---------- 78

8.1 Libraries ---------- 80

Abstract

Particle systems attempt to recreate complex natural phenomena through graphical rendering techniques. Nowadays Particle Systems are used in numerous digital applications such as video games, animations, films and all kind of digital art creations.

This project aims to develop an open-source engine application to serve as a fast and understandable editor tool for creating complex Particle Systems simulations, working towards a process speed up of the big time investment this task usually takes. The tool will be completely managed by a nodal interface, with all the necessary nodes and components required to create such simulations.

The engine uses C++ as the main programming language, OpenGL for 3D graphics rendering and ImGui to build the user interface and the nodal system. That system will be able to create and edit a wide amount of nodes storing parameters and particle systems components along their interactions, intended to be designed for users without any previous technical experience.

The structure of the tool is built as a resemblance of already existing engines with node-based procedural generation tools such as Unreal and Houdini.

Keywords

- VFX
- C++
- OpenGl
- ImGui
- Particle System
- Emitter
- Node Editor
- Houdini

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

Links

Showcase Video:

https://drive.google.com/file/d/1X0vKVlSsJH3Y7mxXUOwSbhdcx0f8Jjfs/view?usp=shar
ing

GitHub Repository:

https://github.com/Aitorlb7/Node_Based_Particle_System/

Latest Release:

https://github.com/Aitorlb7/Node_Based_Particle_System/releases/tag/v1.0/

Index of Tables

Table 1: SWOT--Pag. 27

Table 2: Average Salary and Invested hours estimation ---------------------------------Pag. 31

Table 3: Overall Expenses---Pag. 32

Index of Figures

Figure 1 - Example of Unity VFX Graph use of particles. ---------- 12
Figure 2 - Example of Houdini use of particles. ---------- 13
Figure 3- Geometry vanishing effect using particles inside Houdini. ---------- 17
Figure 4 - Example of a Particle System inside the Unity VFX Graph along some simple nodes. ---------- 18
Figure 5- Example of a blender Particle System and in the bottom-right a material editor. ---------- 19
Figure 6 - Unreal Engine Cascade editor with multiple Emitters. ---------- 20
Figure 7- Niagara Particle Modules Workflow ---------- 21
Figure 8 - Gantt Chart ---------- 23
Figure 9 - Pre-Production Gantt Chart ---------- 24
Figure 10 - Production Gantt Chart ---------- 24
Figure 11 - Post-Production Gantt Chart ---------- 25
Figure 12 - July - September Gantt Chart update ---------- 26
Figure 13 - GitHub branching feature ---------- 28
Figure 14 - Agile sprint based workflow pictogram ---------- 34
Figure 15 - Project Hacknplan ---------- 35
Figure 16 - Core Modular Structure Diagram ---------- 39
Figure 17 - Full ASE Graphical User Interface ---------- 42
Figure 18 - UML of the Particle System and the Node editor implementation. ---------- 44
Figure 19 - Example of the Screen Aligned and Camera Aligned Billboard. ---------- 47
Figure 20 - Example of the Axis Aligned Billboarding from Unity Engine. ---------- 47
Figure 21 - Particle Mesh vertices buffer ---------- 51
Figure 22 - Particle Mesh UVs buffer ---------- 51
Figure 23 - UpdateParticlesBuffer function used to fill the buffers with the new dat. - 52
Figure 24 - Use of the OpenGL function glVertexDivisor. ---------- 52
Figure 25 - Use of the OpenGL function glVertexDivisor. ---------- 53
Figure 26 - Performance without Instancing ---------- 53
Figure 27 - Performance with Instancing ---------- 53

Figure 28 - Example of a complete Node Based Particle System window. ---------------- 54
Figure 29 - Core Modular Structure Diagram. -- 55
Figure 30 – Emitter node with all the necessary Input Pins---------------------------------- 57
Figure 31 - Velocity and Color Nodes with two float4 as inputs. --------------------------- 58
Figure 32 - Color and Color Overtime Nodes with two float4 as inputs.------------------ 59
Figure 33 - Alignment Node with all the possible alignments. ------------------------------ 59
Figure 34 - Positioning of the Assets Explorer Window and the Node Editor Window next to each other. -- 60
Figure 35 - Texture Node. -- 60
Figure 36 - Spawn From Model Node.-- 61
Figure 37 - Boolean Node.--- 62
Figure 38 - Float Node. -- 62
Figure 39 - Vector3 Node -- 63
Figure 40 - Gravity Node. -- 64
Figure 41 - Gravitational Field Node -- 64
Figure 42 - Group selection inside the Grid panel. --- 65
Figure 43 - Example of the resulting Line rendered when two Pins are Linked. --------- 66
Figure 44 - UML of the Draw function of the Window Node Editor.------------------------ 67
Figure 45 - Left Panel -- 69
Figure 46 - Direction of the Flow between nodes. -- 70
Figure 47 - Style Editor of the Window Node Editor. --- 70
Figure 48 - Node creation Node (Right Click inside the Grid)------------------------------- 72
Figure 49 - Results achieved using the final version of the Node-Based Particle System. --- 73
Figure 50 - WickedEngine GPU Particles simulation -- 76
Figure 51 - Unreal Engine: Niagara editor window-- 77

Glossary

OpenGL: Stands for Open Graphic Library and works as an API (application programming interface) that allows direct communication with the graphics hardware.

Dear ImGui: C++ library focused on giving the user a large amount of methods and tools to build and customize your own Graphical User Interface.

Node: Graphical self contained piece of functionality and information storage of some parameter or module of a Particle System, bound inside a canvas and connected to other nodes.

VFX: (Visual Effects) Manipulate live-action shooting or created images via computer-generated imagery to enhance the result of a film, animations or cutscene.

Particle: Single element displayed by the emitter inside a Particle System, containing an image, velocity, color among other properties.

Emitter: Entrusted to spawn the necessary particles with the assigned values to recreate the desired effect, (there can be multiple inside a single Particle System).

Particle System: Collection of a large number of particles with the same behaviour as a whole although with particular random characteristics.

Billboarding: Automatically orient or align the rendered particles to the camera or any other point of reference defined.

Z-Buffer: Graphics programming technique used to determine whether an object is visible in the scene and the drawing order of each of those objects.

Soft Particles: Particles rendered by performing a depth test creating smooth intersections with other geometry.

Houdini: Procedural system tool created to allow developers and artists to create multiple iterations and work freely.

Buffer: OpenGL Objects capable of storing an array of unformatted memory allocated by the CPU.

Pure Virtual Function: It is a function which doesn't have an implementation in the declaration class, although is meant to be overridden by its children classes.

Pointer: It is a variable in charge of storing the memory address of another object. It is commonly used for allocating new objects and iterate over elements in arrays or data structures.

1. Introduction

Creating the proper tools is a must in videogame development in order to achieve the desired scope within the design of the game.

This research project seeks to solve one of the many VFX needs of a game while making it accessible and easy to use to all the non-technical developers.

This section will define the scope of the project and the motivations that took the initial idea to the actual development, while aiming to solve an actual problematic in the videogame industry.

1.1 Motivation

The first idea that would become the first draft of this research was born with the use of Houdini, its procedural nature and how every VFX project could become an independent tool by itself captivated me.

Houdini is built from the ground up to be a procedural tool, so every attribute from the first node is remitted down to all the other nodes which rely consecutively on the previous one, allowing for a fast iteration of possible outcomes

From all the possible aspects of graphical programming that could benefit from a node-based procedural editor I narrowed the scope to exclusively particles as they are an uncharted territory in my technical background yet a very appealing topic to learn and master.

1.2 Problem presentation

Nowadays particles are being rendered in real-time in our screens constantly to recreate distinct environment or special effects. AAA games usually require several VFX developers and artists to meet the desired result.

Within Unreal Engine, Unity and even Houdini lies vast and deep particle systems with numerous of modifiable properties or modules allowing the creation of those sophisticated and flashy effects like the ones in *Figure 1*. Although the initial learning curve in most of them may overwhelm the user with a huge amount of information to learn and retain in order to build those effects.

The technical accessibility for all developers along with the possibility of creation a tool for each effect allowing fast iterations to find the best fit are the main issues this research intends to solve.

The procedural nature of Houdini and the Unity VFX Graph already tackle that problematic each in their own way making possible complex simulations as shown in *Figure 2*, borrowing well executed ideas from both I intend to develop my own open source solution in a simpler way.

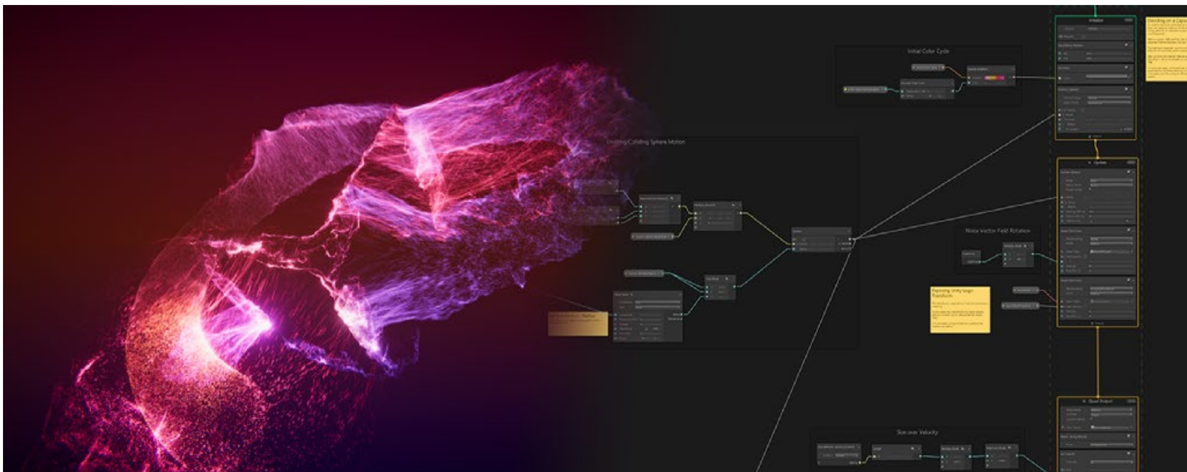

Figure 1 - Example of Unity VFX Graph use of particles.

Figure 2 - Example of Houdini use of particles.

1.3 General Objectives

The overall objective of the project is to solve the issues stated in the section above while making a friendly and intuitive node graph editor within the user interface.

The end goal of the project is to achieve a nodal system capable of creating a wide range of particle simulations possibilities. With the intention of setting a realistic scope for the project while achieving all the specific objectives and an appealing result, the development will be focused into making a display-only tool inside the ASE engine. It won't allow exporting the result into other software nor it will be built as a library to import.

The tool structure will be centered around two panels inside the engine viewport, one for the node-based editor and the other for the real-time renderer of the effect. In order to build the node editor panel, the nodal interface will be handled with the ImGui library (https://github.com/thedmd/imgui-node-editor), the serialization of each node or modules will be handled by the own ASE ([Another Small Engine]) serialization methods, and the rendering with OpenGL.

Furthermore, it is intended to document all the process, going through each step from the first implementation of the base Particle System to adapting it to the nodal structure and all the necessary interactions.

1.4 Specific Objectives

Building a procedural tool, it's a complex task thus the project will be split into several small systems or tasks, attainable and compact to solve one at a time, making certain about the fulfillment of the objectives mentioned in the previous points.

- Develop a solid base engine to build the tool with.
- Adapt the node library and expand it with the necessary features to support the particles interactions.
- Build the Node-Editor panel.
- Create the particle nodes with the necessary attributes.
- Implement a basic Particle System.
- Expand the Particle System and adapt it to the node graph workflow.
- Implement the serialization of the Particle System variables or attributes stored inside the nodes.
- Allow multiple particles spawner and a high number of particles in scene.
- Implement the integration and use of external geometry inside the particles simulations.
- Optimize the performance to ensure stable FPS.

1.5 Project Scope

This research project is meant to become a prototype tool only to be used inside the engine itself, there is no purpose to allow the exporting of the VFX created to other engines thus focusing on expanding the usability and simulation possibilities. Furthermore, the scope is limited to the time and resources restriction since it could be as extensive as the previously mentioned engines VFX tools, and even those are in constant development aiming to recreate as close as possible real effects and natural events.

The limitations of the project will vary over time, initially it is expected to have implemented the basic components needed to build most of the special effects an indie game could estimate, this will be regulated with various versions once achieved the preestablished minimum (v1.0) each finished feature will be added as a new version.

As mentioned previously the tool will be open source giving full access to any developer to use or even expand it in any desired way. The main beneficiaries will diverge from artist attempting to develop in a more accessible tool how to use Graph Nodes or create simple VFX down to programmers or even students taking the tool and the documentation to learn about any aspect from node visual programming or building a Particle System from scratch.

2. State of the art

This section will revolve around contextualizing how the videogame industry attempts to manage a viable solution to the topics and issues stated analyzing the most important aspects.

Nowadays there is a huge number of resources and tools thought to help solo developers and big companies although only a few solve the problematic this research attempt to tackle, these are the engines or tools focusing on real time rendering through a node-based interface.

2.1 Houdini

Houdini is a powerful software developed by SideFx initially designed for artists inside the film industry, either for animation or film VFX, which eventually would also englobe video games and virtual reality in a single tool.

Houdini supports plug-ins with 3D apps such as Maya and 3D Max from Autodesk or into game engines such as Unity and Unreal Engine, bringing in a fast and intuitive way to share the finished assets and effects ready to use.

Unlike other common 3D modeling or animation software such as the recently mentioned Houdini uses a unique procedural node-based workflow granting the possibility of fast iterations back and forth at any time, those other editors store the changes in a user history making it harder to revert or modify any previous version of your work.

Although Houdini features great modeling, animation, rigging and many other toolset systems that have nothing to envy other software, this project will focus on its advanced dynamic simulation and particle effects.

All geometry and more important Particles inside Houdini have attributes, important information storage passed down each node that will use it to handle interaction with other systems or attributes. For example, one of the end goals of this project is to emit

particles from the vertex or points defined by some geometry therefore in Houdini the particles emitted from geometry inherit the attributes of the point from which they are emitted allowing the creation of vanishing effects by transferring attributes between nodes as seen in *Figure 3*.

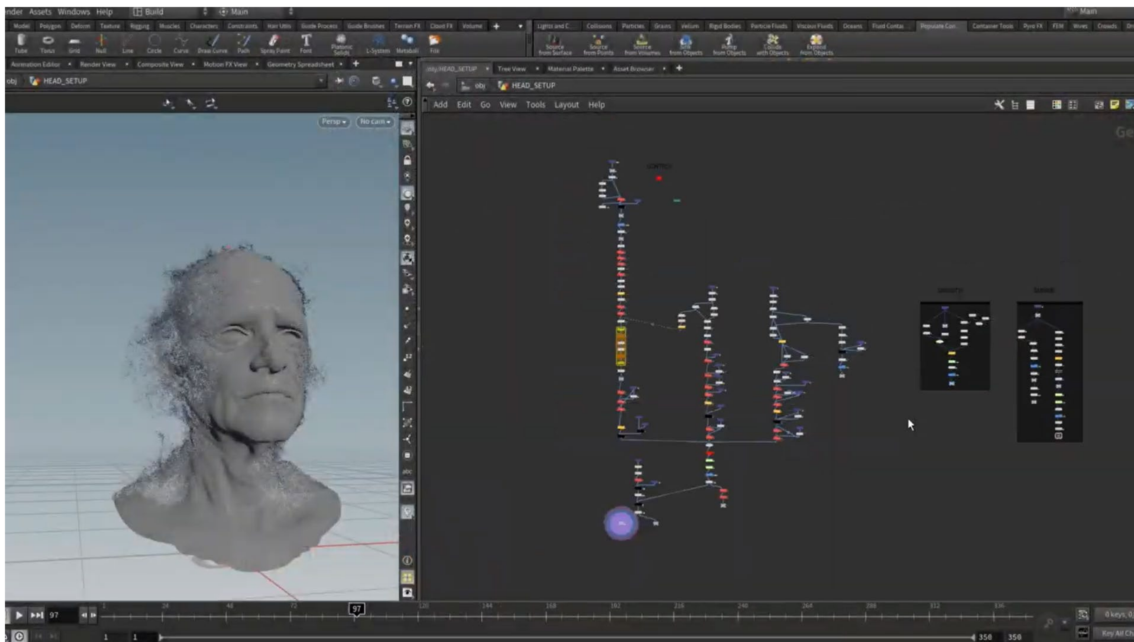

Figure 3- Geometry vanishing effect using particles inside Houdini.

However, it seems that the nodal approach only brings benefits, it's a false appreciation, as it requires previous extensive documentation reading, usually overwhelming the user with the amount of nodes available.

2.2 Unity VFX Graph

The Unity VFX Graph enables the developers to recreate multiple complex simulation effects using Node-based visual logic, previewing changes immediately and with a step-by-step simulation.

This new addition doesn't aim to replace the current Unity particle system as it doesn't support interaction with the current scene or even the in-game physics system yet expands the possibilities given to the developer as it has other advantages.

The VFX Graph swaps from the CPU usage for the particle calculations to the GPU allowing the simulation and rendering of millions of particles on screen, it works at a much lower level enhancing the process calculations agility in real-time as well as making it easier for the developer to iterate over an effect.

Unlike Houdini the VFX Graph doesn't use the simple looking nodes with all the information and attributes stored inside, it will come with four main nodes; the spawner which will mainly handle the number of particles spawned per second, the particle initializer with all the modifiable properties of the particles, the update to modify the behavior on runtime and finally the output in charge of how the particle is being rendered. Each of those nodes allow the implementation of blocks inside small modifiers to either state of the particle.

Internally this top-down approach with the module and attributes management is closest to the expected behavior of the tool.

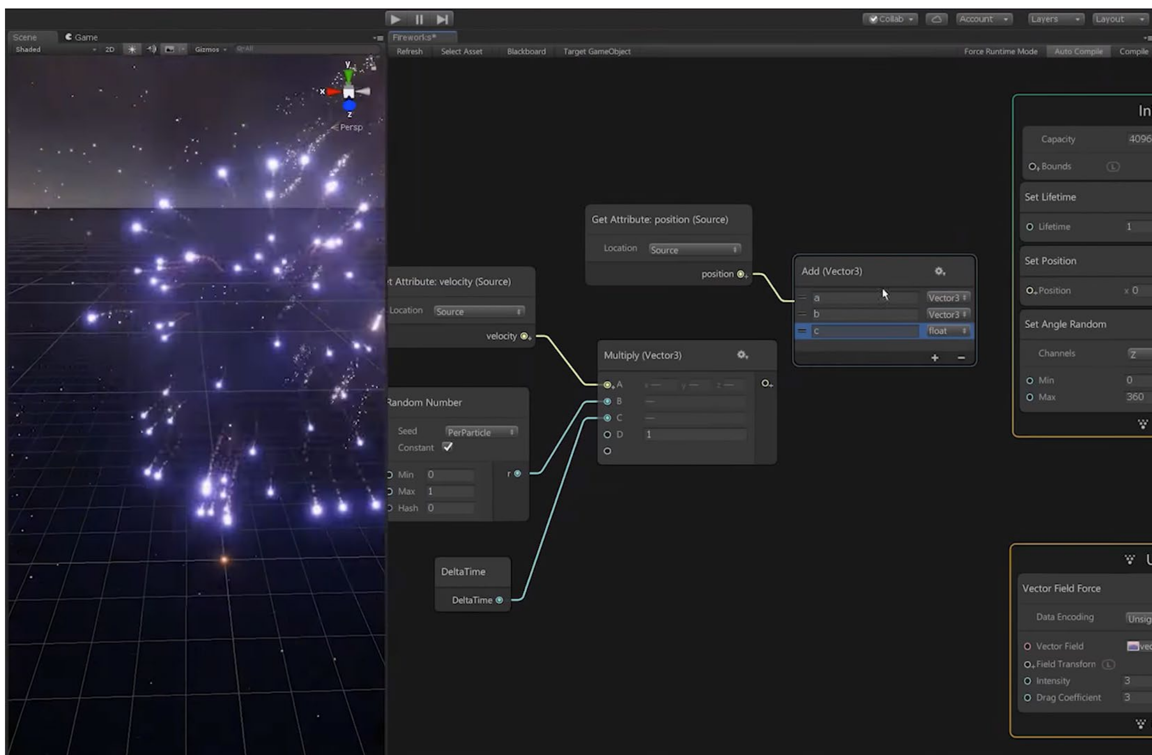

Figure 4 - Example of a Particle System inside the Unity VFX Graph along some simple nodes.

2.3 Blender

Blender is an open-source 3D software aiming to contain the entirety of the 3D pipeline this being modeling, animation, simulation, rendering, compositing, motion tracking and even video editing and game creation.

Since it is open-source users can freely develop their own tools within the software suiting solo developers and small indie studios who benefit from its unified 3D pipeline and responsive development process.

The Blender community is currently developing and adapting particle nodes into the node-based material editor, it is possible to recreate simple particle effects with a small degree of complexity with the available nodes yet is far from being a complete tool on its own.

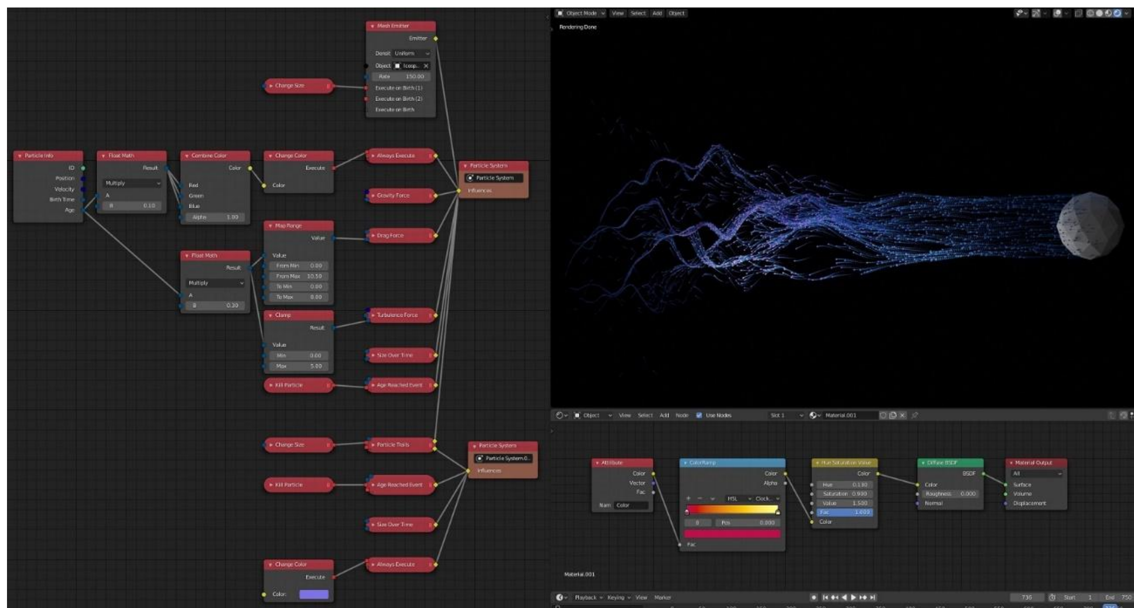

Figure 5- Example of a blender Particle System and in the bottom-right a material editor.

2.4 Unreal Engine

Unreal Engine like Unity is primarily a game engine although it posses a complete suite of development tools aimed towards working with anything requiring real-time rendering and execution.

Even tough Unreal doesn't have a node-based interface to edit particle systems, it is a necessary mention due to its great modular structure and how it handles the information similarly to the previous engines, they call it Cascade.

Cascade is Unreal Particle System editor; it offers real-time feedback and modular effects editing while seeking fast and easy creation of those effects. It allows several emitters (as shown in *Figure 6*) each with their own parameter configuration and consequently their particles which they will be in charge to spawn and even modify their behavior throughout the entire effect.

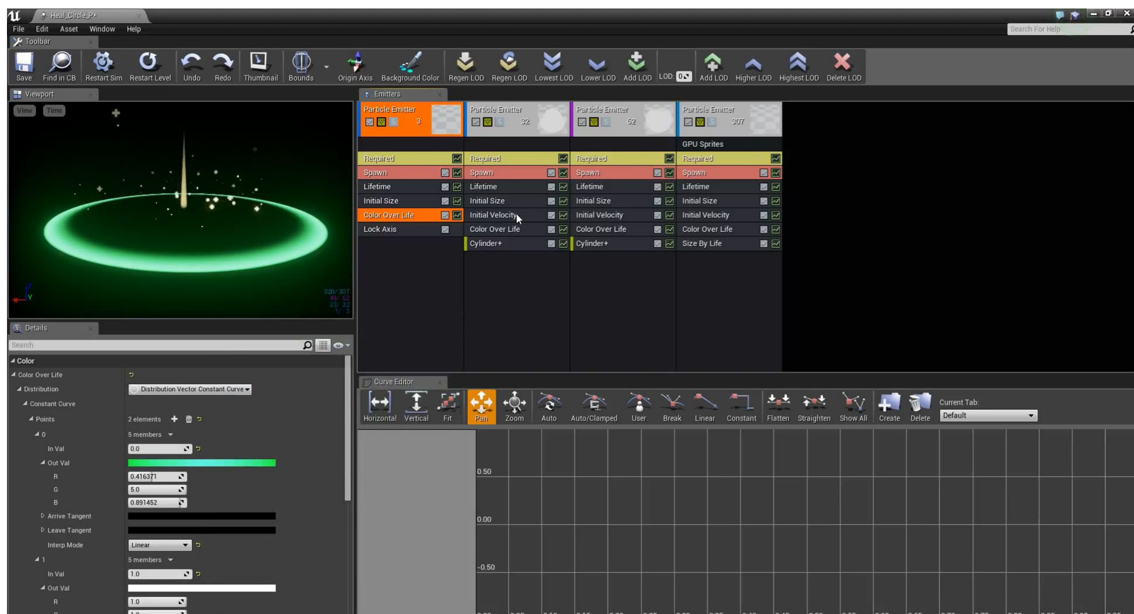

Figure 6 - Unreal Engine Cascade editor with multiple Emitters.

As we've seen before with Unity, Cascade is not the only way to create VFX in Unreal Engine with the release of the version 4 of the engine came Niagara.

Niagara is Unreal Engine's next-generation VFX System focused on giving technical artist the ability to create additional functionality on their own, without the assistance of a programmer.

Simulations in Niagara operate as a stack, executing modules in order from top of the stack all the way to the bottom.

Niagara splits in a top-down hierarchy of four core components:

- **Systems**: Containers for multiple emitters, all combined into one effect, with the possibility of modifying or overwriting anything in the emitters or modules inside.
- **Emitters**: Re-usable and single purpose containers for modules, allow multiple modules stacking and render the simulation in several ways within the same emitter.
- **Modules**: Programable code blocks equivalent of Cascade's behaviors (Logic inside each emitter visible in *Figure 6*), they encapsulate behaviors, stack with each other and operate with new functions.

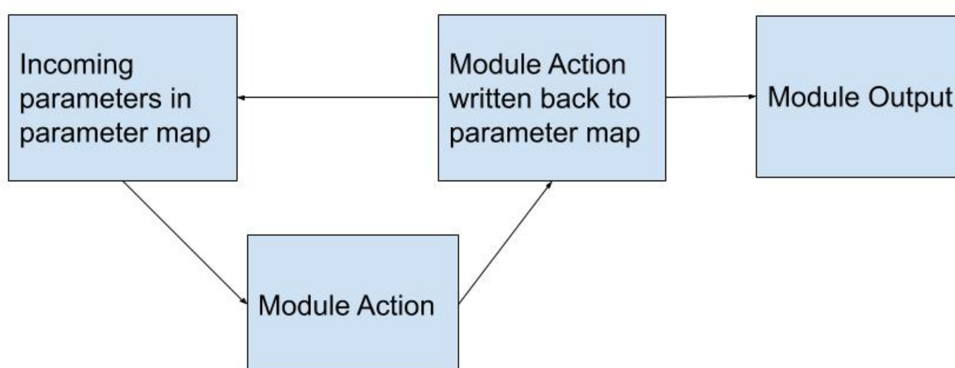

Figure 7- Niagara Particle Modules Workflow

As we can appreciate in *Figure 7* modules are accumulated into a temporary namespace, afterwards they can be stacked with more modules altogether, as long as they contribute to the same attribute, the modules will stack and accumulate properly.

- **Parameters**: Custom abstraction of data assigned to a specific type such as primitives, enumerators, structs and data interfaces. They define modules and particles characteristics.

2.5 Conclusion

To sum up the overview of the State of the Art we can appreciate there are plenty of VFX simulation tools and engines aiming to fill the same gap as this research, some are still in development while others already reached a gold (market product ready) state.

The market target matches with this project although the objectives previously stated remain the same, this tool aims to recreate some of the technical features we`ve just seen while keeping the user interface clean as a priority.

Competing to grab a portion of this market is not a realistic objective thus it is expected to expand the tool as much as possible trying to resemble the combination of the best features of these VFX tools and focus on the learning and documentation process.

3. Project Management

In this section we will go over the planification and management of the project, from the scheduling tools to organize and set milestones with Gantt and HacknPlan, to validation tools analyzing positive and negatives aspects of the project scope and objectives.

3.1. GANTT

The Gantt chart allow us to translate the objectives set up in previous sections into visual representation of the time span they take up. Splitting them into manageable sprints clarifies the chart and improves the task completion efficiency. The visual representation of time schedule allow the author to have a global consideration about the priority of each task or even the time assigned.

Before diving further into details, here, you can find the Gantt chart at full size and definition. For this project the available time was split into 3 major blocks:

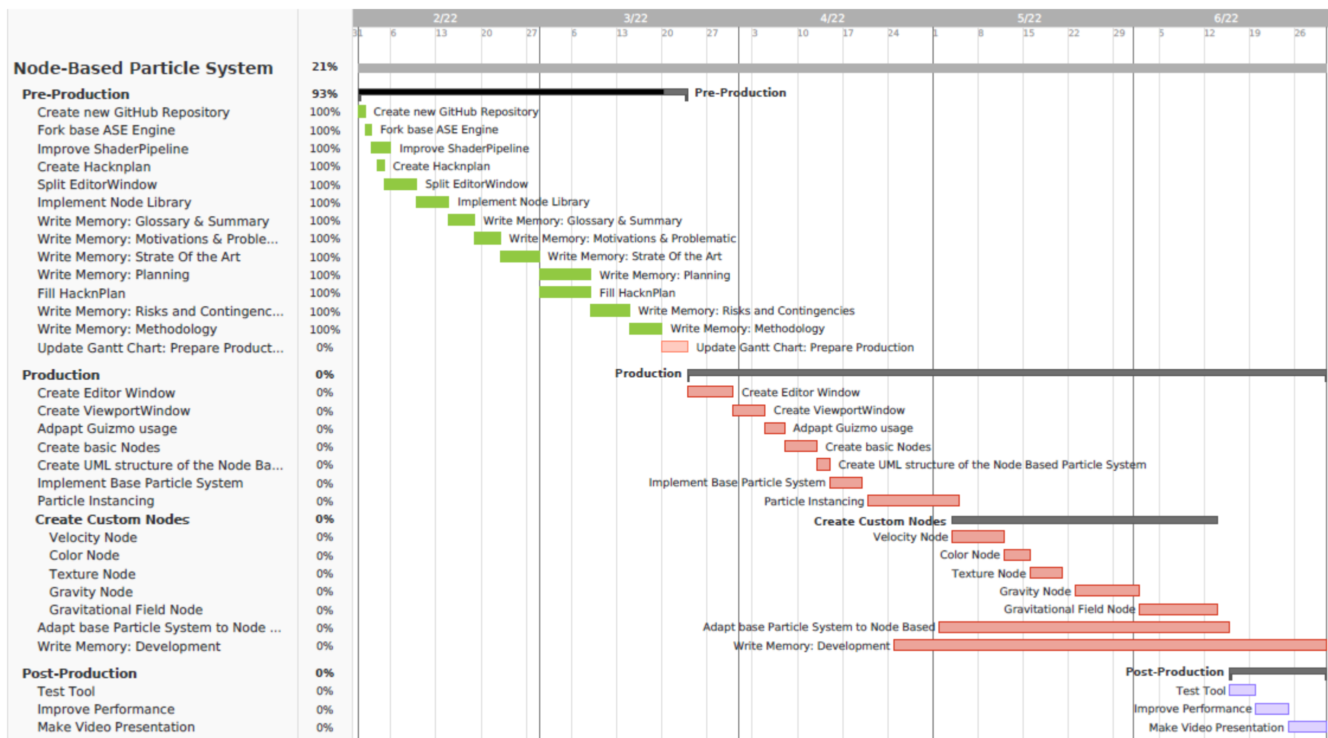

Figure 8 - Gantt Chart

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

3.1.1 Pre-Production

The pre-production block, is scheduled for building the core engine, adapting it to the features to come, and implementing the core library around which the nodal base of the project will be assembled. As we can appreciate in *Figure 9* the memory development was also included once the engine

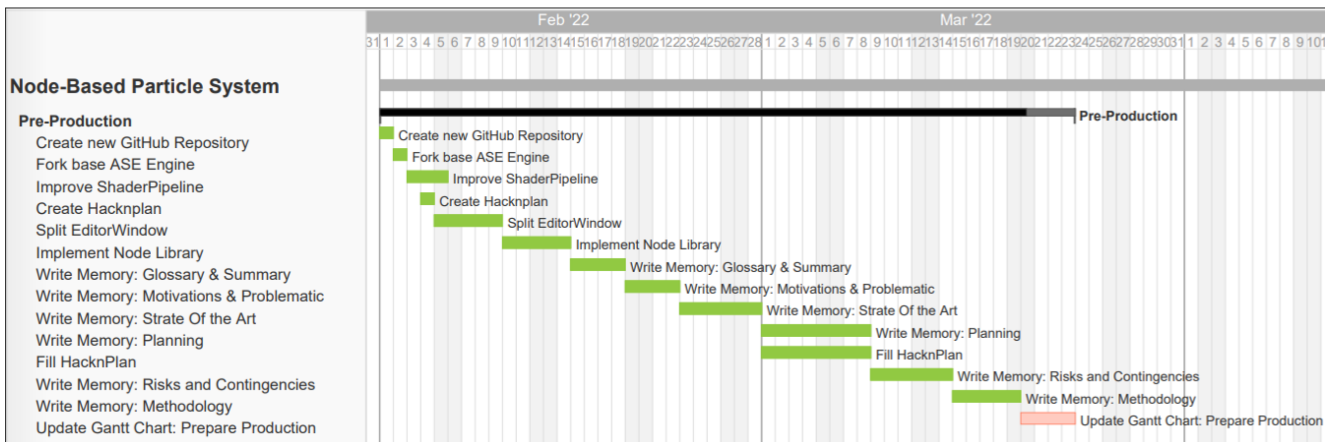

Figure 9 - Pre-Production Gantt Chart

3.1.2 Production

The production planning phase was challenging to fit into the time span available, some expected features due to develop were given less time than expected. However, as *Figure 10* shows the project is ready to go into Production.

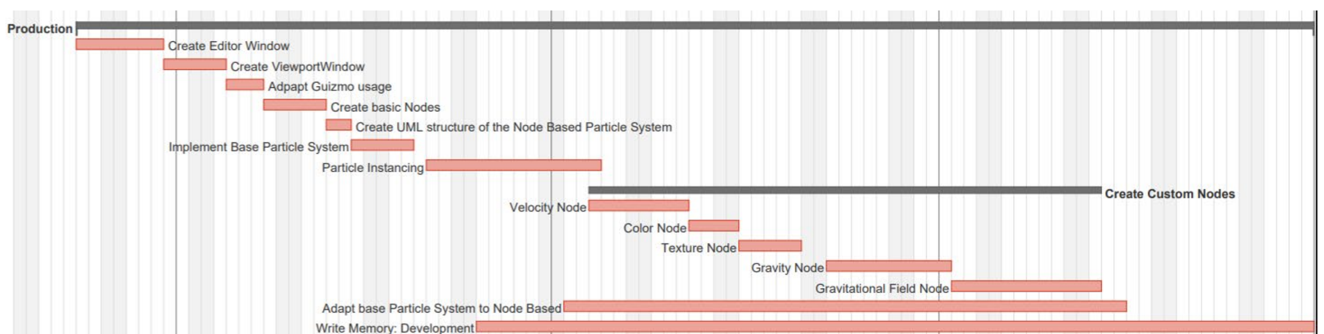

Figure 10 - Production Gantt Chart

In this stage the actual tool development takes places. First we will set up the ground work, creating the windows needed adapting a new viewport and the basic structure for the particle system and node editor communication.

After we have the essentials up and working, its time to implement all the custom nodes planned out, including improvement in the performance. Everything aiming to fulfill the objectives and obtain a well developed and usable tool.

3.1.2 Post- Production

By reaching the post-production block, the tool is expected to be fully implemented and documented in the memory. Now it is time to test the tool, finding its limitations while developing those VFX simulation. After testing it, if required polish any performance issue if may present or as a matter of fact any issue that may come up, that wasn't solved in production or was overlooked, should be fixed during this period.

Last focusing the scope on the presentation, the preparation of the video sample is included in this post-production period.

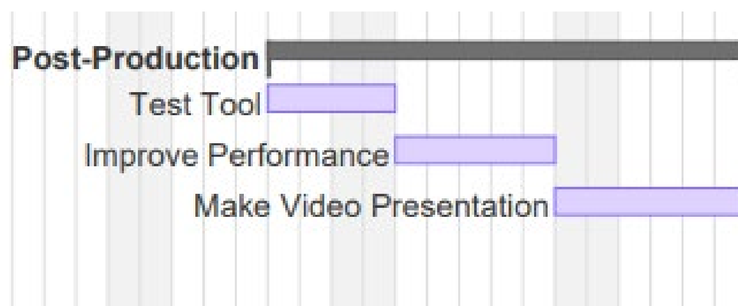

Figure 11 - Post-Production Gantt Chart

3.2. GANTT Update

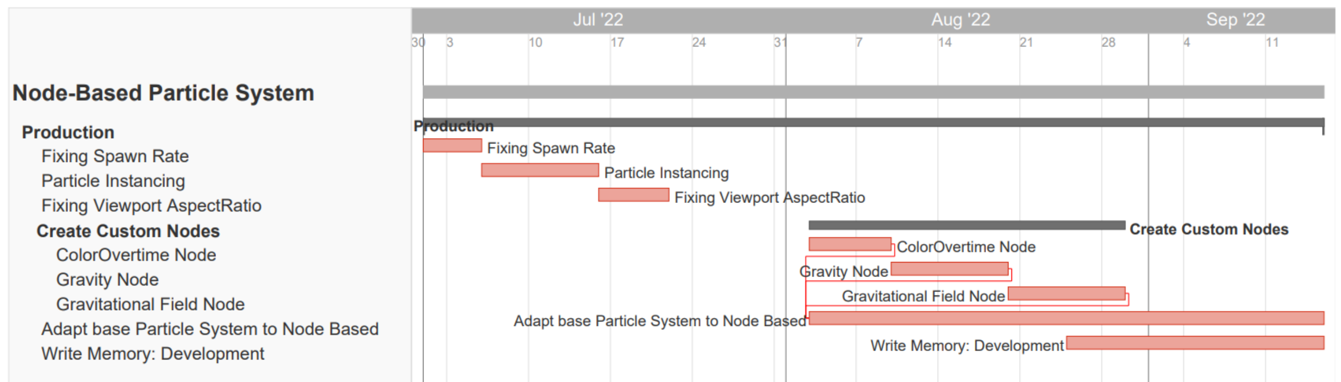

Figure 12 - July - September Gantt Chart update

Due to unexpected personal time schedule issues, mostly related to sustaining work and university studies simultaneously, we couldn't keep up with the initial planning, thus the project deadline was delayed until 16th of September.

Once the deadline was pushed the Gantt chart needed to be updated as well, all the unresolved tasks were adapted to the new time span available, and the new ones such as issues to be solved were prioritized to make the remaining development smoother. The instancing was completely relegated to this extension of the production, as well as the ColorOvertime, the Gravity and the GravitationalField nodes.

3.3. SWOT

This section focuses on analyzing several aspects of the project, in order to identify possible positive and negative points and act in consequence while planning. Structured in a table for better visualization of each aspect analyzed.

	Positive	**Negative**
Internal	**Strengths** • Built in its own engine with full control over all systems. • Previous knowledge about Particle Systems. • Previous knowledge about graphics and rendering. • Investment of resources into optimization.	**Weaknesses** • Time limitation. • Engine built from scratch requires a big amount of side work. • Hard to set realistic goals (project can be really extensive in many ways) • No previous knowledge about node based interaction handling.
External	**Opportunities** • Multiple development paths and choices. • Accessibility of the outcome tool, compared to other high-priced softwares • Learn and master a new game development field.	**Threats** • Multiple existing engines softwares addressing the same objective with outstanding results. • Big workload for a single person research project. • No documentation for this specific case.

Table 1 -SWOT project analysis

3.4. Risks and Contingency Plan

When addressing a project of this dimension as seen in the previous section, estimating the possible risks throughout the development and setting a contingency plan to mitigate or deflect them completely is necessary. In this section we will tackle those possible risks and the solutions applied in this project.

- **Time Constraint**

In the development of projects of this size it's common to underestimate the difficulty or time requirement the implementation of a single feature can take causing a delay upon completing that milestone as expected. When that happens the planning needs to be rearranged to fit the time limitation preestablished.

In order to prevent those setbacks the github branching is used. Split every important or time relevant feature into an independent branch, thus allowing for safe interactions over the implemented feature without compromising the overall project. Acts as a firewall and isolation of every task planned, furthermore it allows to swap instantly from the development of a feature to another if needed .

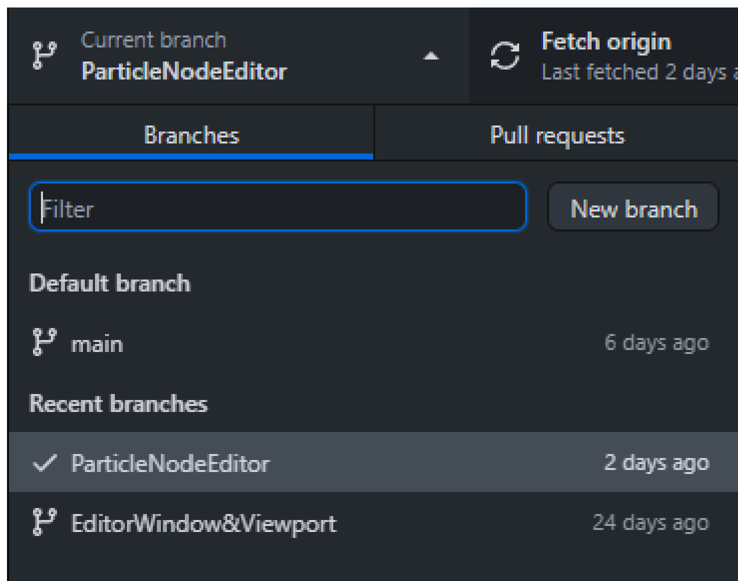

Figure 13 - GitHub branching feature

Another prevention method slightly less used in this project is booking empty space or extra space in the schedule for the harder or more time consuming tasks, thus avoiding the urge of rescheduling in case of any unexpected setback.

When planning and setting up each milestone in the Hacknplan as seen in 4.1 Production, assigning each feature to a single task or even splitting the bigger ones into several tasks works as a better time tracking technique.

- **Customization and adaptation of ImGui**

Using an external library to build up the Node Editor window, one of the core pillars of the project caps the level of design and development freedom, even leading to unexpected bugs regarding the integration and usage of those libraries.

ImGui and ImGui Node Editor offer a wide variety of functionalities, although this project requires an extensive customization level both in terms of user interface design and implementation.

Upon encountering such issues it may be necessary to deviate the direction of the feature or even cutback some aspects that could be way harder or more time consuming to develop.

- **Lack of knowledge and documentation to achieve the expected result**

The technical limitations of the development team (solo developer) is something to take into account while researching and planning. The lack of documentation and limited knowledge can directly influence the end result ensuing to not meet the expectations and goal set.

This risk requires to be tackled before planning by doing a previous research of the libraries expected to use, and comparing the usage capabilities of those with the outcome the project needs, splitting each feature and studying their viability.

- **Slow performance**

Particle Systems generally attempt to render a great number of particles in real time, even having several particle systems in a scene at the same time. Thus affecting the performance of the engine and the tool.

The project aims to have a stable 60 fps even though it remains as a secondary goal, focusing on delivering the expected end goal and afterwards optimizing it as much as possible.

To prevent a drop in performance there are several options, the one that has been already included in the planning of the project is the particles instancing, as mentioned in the article: (http://www.opengl-tutorial.org/intermediate-tutorials/billboards-particles/particles-instancing/) The idea behind this concept is to reduce the number of render calls, with some buffers in charge of defining a base mesh and others the small differences with the other particles instantiated.

3.5. Cost Analysis

In order to calculate and perform a proper cost analysis we will specify the starting date of the project in February 2022 and the end date in June 2022, also we need to specify an average salary of a game engine developer which will be extracted from further research (glassdoor, payscale and gamasutra).

AVERAGE SALARY		
	EUR / Year	**EUR / Month**
Game Developer	30.000,00 €	2.500,00 €
ESTIMATED HOURS		
	Hours / Month	**EUR / Month**
Full-Time	160	2.500,00 €
Own Schedule	60	937,50 €
TOTAL	**Total Hours**	**Total**
	300	**4.687,50 €**

Table 2 - Average Salary and Invested hours estimation

To make a proper estimation of the working hours it was taken into account the planned tasks and their expected duration (along with an empty space as a contingency plan as seen in 3.4 Risks and Contingency Plan) and the hours available in the developer schedule, coming up with an estimated total amount of 300 hours and 4.687,50€ in salary expenses.

COSTS							
	February	**March**	**April**	**May**	**June**	**Single Payment**	
Electricity	135,00 €	0,00 €	135,00 €	0,00 €	135,00 €	—	
Internet	30,00 €	30,00 €	30,00 €	30,00 €	30,00 €	—	
Desktop Computer	—	—	—	—	—	1.650 €	
Peripherals	—	—	—	—	—	450	
Salaries							
Developer	937,50 €	937,50 €	937,50 €	937,50 €	937,50 €	—	
Software							
Photoshop	12,09 €	12,09 €	12,09 €	12,09 €	12,09 €	—	
TOTAL:	**1.114,59 €**	**979,59 €**	**1.114,59 €**	**979,59 €**	**1.114,59 €**	**2.100,00 €**	**7.402,95 €**

Table 3 – Overall Expenses

Once we had the salary expenses it was left to calculate all the remaining costs that could take place during the development, such as the monthly expenses of electricity and internet along with single time payments for the equipment needed.

Software licenses used were mostly free libraries and cloud services such as Google Drive and Github while the only expense was Photoshop used to create the particle sprites.

Table 2 is the final expected expenses of the whole development reaching an approximate total of **7.5K** euros.

4. Methodology

This Project contains an extensive practical side with the release of a complete yet simple tool as an end goal and a documentation process of the whole development from the definition of the objectives, going through the State of the Art and the tool progression.

To reinforce and optimize the correct execution, the project will be structured in pre-production, production and post-production.

This methodology proposition will be complemented using Hackplan and the segregation of each milestone into sprints.

4.1 Pre-Production

The first need of the project is the correct planification and time management, clarify the objectives with all previous information and research needed to decide those with a solid understanding background on the topic.

By the end of pre-production, we will need a stable engine to work with and structure the tool around it, in order to achieve such thing the base engine needs some groundwork:

- Structure the ImGui user interface into several panels.
- Optimize the current rendering in OpenGL
- Implement the node base library
- Create the base particle system.
- Develop an UML (Unified Modeling Language) of the adaptation of the particle system into the node-based editor.

4.2 Production

At this point the project enters its main development stage, this section covers the organization of the biggest part of the project and how to schedule it efficiently.

Here comes Hacknplan, a project management tool specifically designed for game development bringing an improved workflow between managers, programmers, artists and designers. It follows agile methodologies by tracking the progress of each task structured in a Kanban board.

We will focus only on the technological side, splitting the tasks scheduled in the Project Management section into more manageable assignments, always following the deadlines established in the Gantt chart.

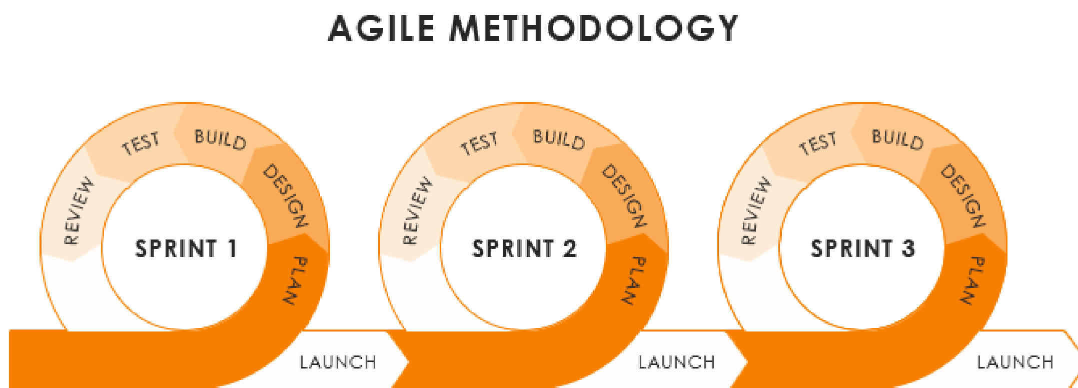

Figure 14 - Agile sprint based workflow pictogram

The Production of this project will follow a series of Sprints (It is expected to have around three or four) each with their corresponding releases in GitHub at the end thus being an iterative process they will split into the following:

- **Plan:** Estimate the overall time the task will take, the objectives and the procedure to follow. In this sprint it is expected to design the UMLs for the Particle System, the Node-Editor, and how will they communicate.
- **Design & Build:** The actual development and implementation process, following everything stated in the planning phase. This is the largest sprint, it

needs to manage from the basic Particle System implementation, going through Node-Based editor window, up to optimizing the completed tool.

- **Test:** Go over the result of the build phase looking for any mistakes and undesired behaviors that might impair the development of future sprints.
- **Review:** Evaluate if the resulting product is ready to be implemented into the tool and is ready to launch and release. This stage usually belong to the last section of the Production, although, due to time constraints it is scheduled to be handled in the Post-Production.

Each Sprint will have an independent board inside Hacknplan dividing the main objectives of the milestone into smaller tasks, each with their description and time limitation, the *Figure 15* shows the Pre-Production board. The tool offers a wide variety of task personalization and the information attached to it however only a few will provide what we need like tags to quickly identify the work group each task belongs, timestamps to keep close control over the hours expected for each task and the ones actually invested to finish it and a small description for each.

To keep the Hacknplan boards up to date at a relatively fast pace, the tasks added will have a Title self-explaining the assignment, a tag to differentiate the objective of the task (Design or Coding) and the time invested along the deadline.

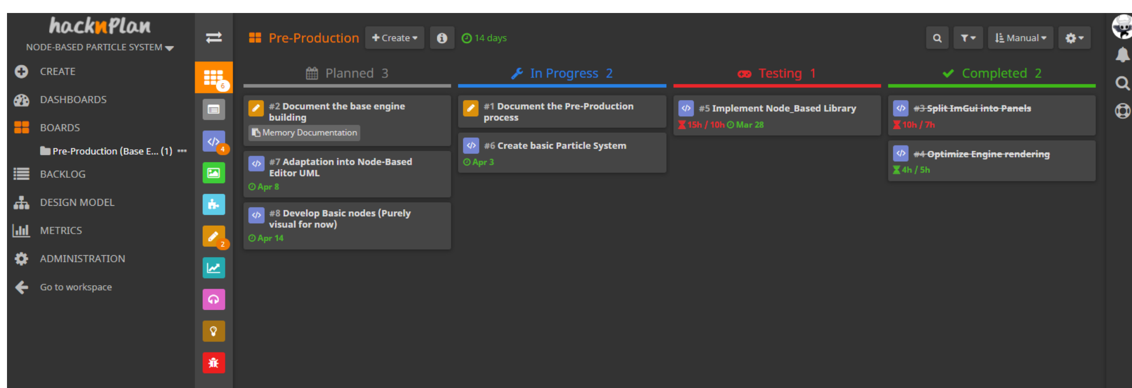

Figure 15 - Project Hacknplan

4.3 Post-Production

In this stage the project is expected to be finished however this stage is precisely to make certain that is in fact ready to release, it dwells mainly on testing and iterating over the tool.

If any issue or bug comes up during this process, it will be assumed and used to create a new task or small sprint if needed, resuming the workflow used in the production phase only this time with short periods and reduced scope.

In the case that the project is considered to be finished, the post-production will be aimed to expand the tool with extra nodes and functionality expanding the simulation possibilities and optimizing the interaction between nodes and the management of the particles.

5. Project Development

In this section we will dig right into the actual development of the project, how the tool is built and all the implemented features. It is intended to reflect and explain all the steps taken up to the finished project in an understandable and coherent way.

On the first place we will take a glance into the base engine upon which the tool is built.

5.1 ASE - Another Small Engine

ASE stands for Another Small Engine, an already developed engine from a previous subject in the degree, Videogame Engines in which we learned in pairs to read, handle and adapt models into a 3D viewer expanding with further features and core functionality it as the course went on, following the previously stablished course guide.

To better understand the base structure upon the tool will be developed, we will dive into the libraries used, the features of the engine and its core structure:

5.1.1 Libraries Used

- **Glew:** Graphic API that provides a high performance functionality for the interaction and usage of OpenGL rendering through the CPU or GPU.
- **SDL:** Library designed to provide low level access to all kinds of input and output (audio, keyboard, mouse, joystick, and graphics hardware via OpenGL).
- **Dear ImGui:** Library which allows to handle and render a customizable User Interface.
- **ImGuizmo:** Works within ImGui providing a visual representation of a geometry modifier (Translate, rotate and scale).
- **ImGuiColorTextEdit:** Works within ImGui providing a real time text editor, to modify and save any file.

- **ImGuiNodeEditor:** The main library used for this project, it works within ImGui and provides a nodal interface, the creation of simple nodes and all their connections.
- **MathGeoLib:** C++ library with a wide range of mathematical functionalities for linear algebra and geometry manipulation.
- **Assimp:** C and C++ library capable of importing and exporting 3D models, supporting numerous formats.
- **Parson:** C written library allowing an easy use and implementation of json, used for serializing.
- **Devil:** Developer Image Library is a simple syntax library to load, save, manipulate and display images.

5.1.2 Engine Core

The engine core section will cover everything contained from the main loop and application, its modular structure through each functionality up to a brief introduction of the Particle System and its Nodal implementation.

- Application and Main Loop

The main loop takes care of the creation, initialization, update and clean up of the application, through those states (Init, Update and CleanUp) it executes all the necessary tasks and modules to render the application window, maintaining all those processes with a proper performance and clean the memory used upon closing it.

The window and everything rendered within will be further handled and explained in deep in the next section, all the modules and their playing part in the bigger loop to bring together a functional engine.

Modular Structure

The application functionality is divided into several modules each one in charge of a specific aspect of the engine, containing all the logic related to the matter inside that same module, thus allowing for a cleaner and better structured code base.

The communication between modules goes through the application object containing a vector of all the modules and a pointer to each of them as seen in *figure 10*, allowing full access to each (if the accessed method or variable is public for external access from the class or module itself).

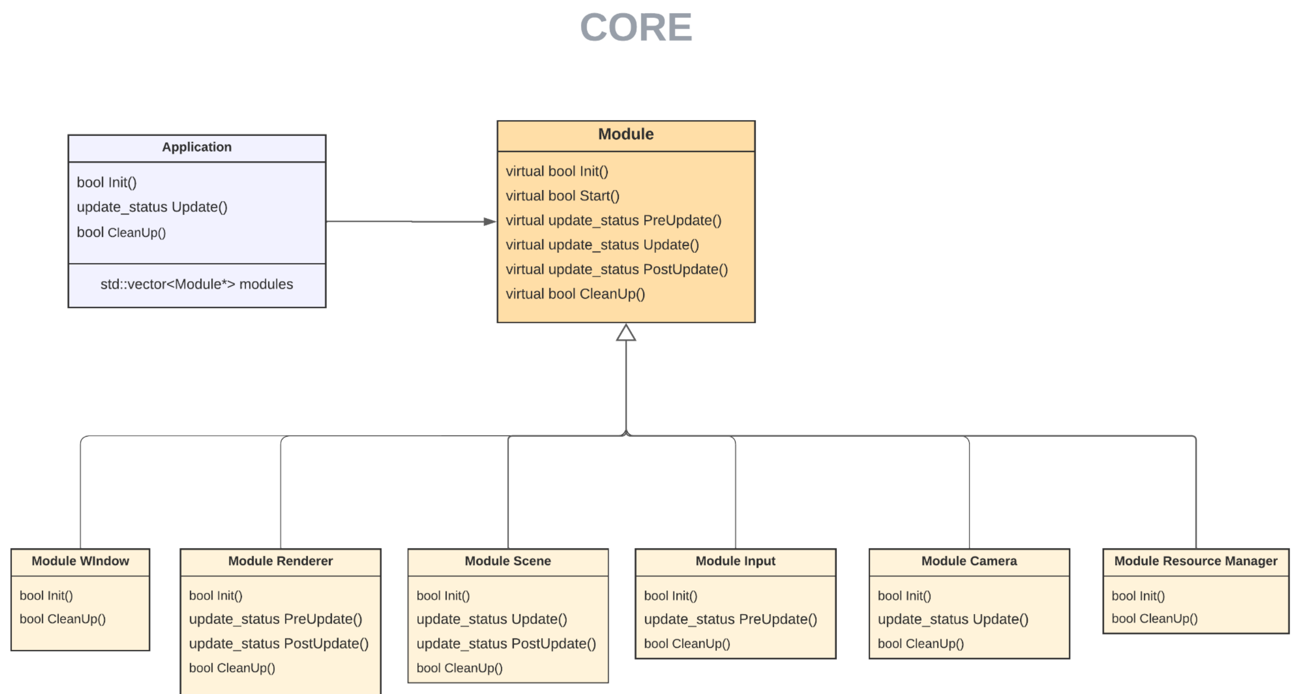

Figure 16 - Core Modular Structure Diagram

As shown in the *Figure 16* diagram of the modular structure, the Application object will hold a vector with all the modules objects instanced, being in charge to call all the methods defining each state, from Init to CleanUp declared in the base class and inherited in all the modules derived.

- ### GameObject and Components

The main idea behind this system is to mimic Unity's GameObject structure and Hierarchy. By having an object with the capability of holding several components it allows the user to add, subtract or easily modify any desired functionality.
Those components can vary from a must have Transform component containing the position, rotation, and scale of a GameObject inside the 3D space, up to the Particle System component an extra feature which will be explained in depth in further sections.

- ➔ Component Transform
- ➔ Component Mesh
- ➔ Component Material
- ➔ Component Camera
- ➔ Component Particle System

With a similar practice as the module system, the Module Scene will hold a vector containing all the created Objects to gain full control over their visibility, deletion and all modification over a single or several GameObjects can have.

- ### Resource Management

The Engine possesses a resource management system, which oversees the handling of all the assets inside the engine dependencies.
More specifically each asset must have a resource type class for process and organization purposes, on starting the engine, the manager will go through all the assets and serialize them properly into the Library carpet allowing for a fast importing and saving of any modification done in runtime, thanks to the full control of the serialization with json.
To facilitate the access to the files in library and register all the related information to each serialized asset we use meta files as Unity does, they contain the name of the asset serialized, the path in library and the UID (Unique Identifier) among others.

- Shader Pipeline

The rendering is supported and implemented through a shader pipeline. It gives the application the capability to send all the render calls containing all the models information inside the scene to the GPU, thus improving the performance of the engine along other advantages.

The GPU unlike the CPU is optimized for tasks such as graphical rendering with its parallel processing, meaning they are able to process several tasks simultaneously.

The shader pipeline system was implemented along an in-engine shader editor and a modifiable uniform system allowing a simple and accessible way of modifying any shader from the user interface.

5.1.3 Main Modules

- ➔ Module Window
- ➔ Module Renderer
- ➔ Module Scene
- ➔ Module Input
- ➔ Module Camera
- ➔ Module Editor

5.2 Adapting the engine

In order to start the actual tool development, the engine needed a few adjustments to fit the implementation of the expected new features. From the window management in the editor file, improvements in the shader pipeline to the initial implementation of the particle system.

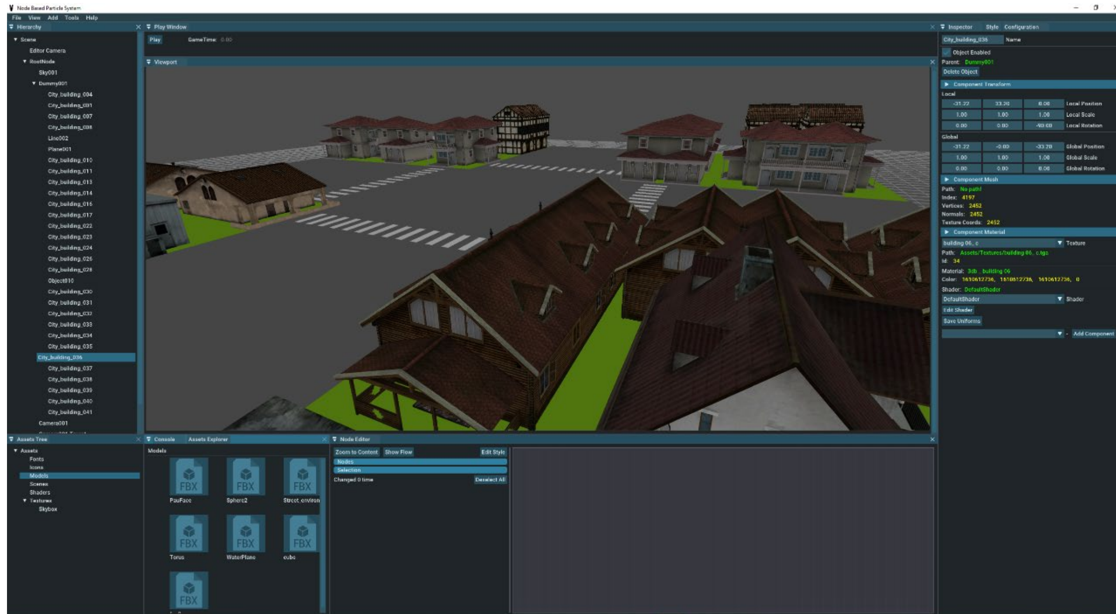

Figure 17 - Full ASE Graphical User Interface

5.2.1 Split Editor file

The class Module Editor contained all the window logic packed inside a single file, so it was divided into several classes and files, one for each window available. The Module Editor still had the hold of all the windows this being the main axis upon which the GUI logic was managed.

Having a better organized and cleaner code base allowed a smoother implementation of additional windows or any GUI logic along the development, such as the Viewport Window which was added shortly after the division was fully functional.

5.2.2 Viewport Window

Initially the scene was rendered directly into the window created using SDL, and the user interface on top of it not leaving much space to reposition the scene or even the user interface structure around it.

To allow the implementation of a node canvas window, the initial window structure of the user interface needed to be adapted, thus the viewport now has its own window.

The viewport is in charge of rendering the output into a texture which can be resized and docked into the user desired position, furthermore all the mouse interactions to select objects in the scene also Guizmos logic was adjusted to fit any desired size without losing the proportions.

5.2.3 Shader Pipeline optimization

The shader pipeline implementation had a few remarkable flaws that could impact the engine performance. For instance each object rendered in screen needed a shader assigned, to which all the vertex and transforms information will be send to render. This assignation was being called repeatedly, once assigned any further call was absolutely unnecessary thus it lead to repeated importing of the same asset or shader.

By assigning upfront the shader to the object material the number of calls was reduced to one per object, saving resources and improving the performance.

5.3 Particle System

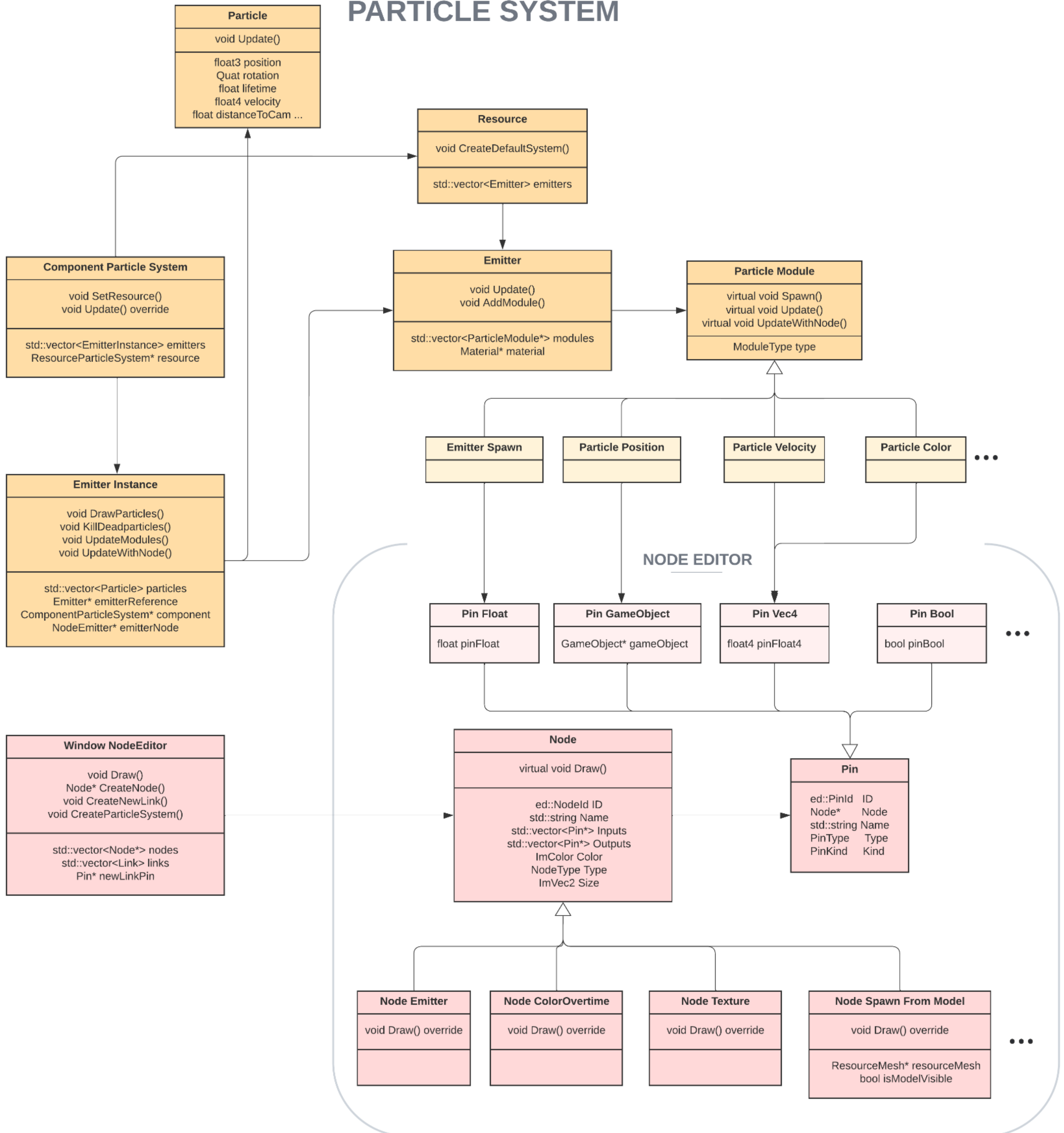

Figure 18 - UML of the Particle System and the Node editor implementation.

Before diving deeper into the development and structure of the implemented tool, we will take a general overview of the Particle System implementation. *Figure 18* we get at fast glance at how the Particle System implementation was designed and thought to interact with the Node Editor.

The core skeleton of the particle system is extracted from the Game Engines subject in the degree. As one of the possible High-level systems taught to the students and demanded to implement into their engines, we were taught how to structure and organize a Particle System. In favor of an easier comprehension following this somehow complex structure we will begin with the smallest element yet equally important the Particle.

5.3.1 The Particle

The particle is an element which we intend to instantiate hundreds or even thousands of times to render into the viewport on runtime, thus they need to hold the smallest amount of information possible, so they don't take all the space in memory and maintain a stable framerate.

Each particle has a position in world coordinates which will be updated every frame, a velocity to update that position, a rotation defining the alignment of the particle, a color applied over the texture (The texture is not stored inside the particle itself, instead it belongs to the entity managing them). The particles also have a lifetime regulating how long the particle is being rendered into the screen, once it surpasses that threshold the aforementioned particle is deactivated and repositioned to the spawn point, this is part of the pooling process which we will review later in this section. Lastly we have the distance to camera, each particle before the draw call calculates their distance to the current camera so they can sorted and rendered in the proper order.

To better understand how the Emitter Reference behaves we need to review other components of the Particle system.

5.3.2 Particle Modules

A Particle Module is a base struct with three pure virtual functions, meant to be overridden by the children structs, however with common logic in all of them. The methods are, Spawn, method to set the default values to the particle, the Update modifues the particles through their lifetime, and finally the UpdateWithNode will do the same as the Update however this time at the request of the EmitterNode. The base struct will also contain a variable of Type, defining the type of module each of the derived structs is, mostly used in the constructor.

Each derived Particle Module struct is in charge of one key aspect of the particles in screen, some will act influencing all particles equally and others will modify each particle individually. Here are all those included in this project.

- **EmitterBase:** The EmitterBase module modifies and adjusts the particle in several ways. First of all in the Spawn function sets as the origin position for every particle at the EmitterInstance position (The one managing all the active partcles). In the Update takes care of the billboarding or alignment of each particle in screen. The Billboard determines where the elements in it faces, it has many uses in videogames, in our case we are using it to define the particles orientation. We can split the types of Billboarding available in this project in three. Screen Aligned, the particles always face towards the screen, maintaining the proportions of the original texture or mesh as we can appreciate in the view plane aligned in the *Figure 19.* The Camera or World aligned rotates and transform the geometry to face towards a specific point, even deforming the geometry as it gets closer to the viewport bounds, in the viewport oriented of *Figure 19* we can appreciate the phenomenon.

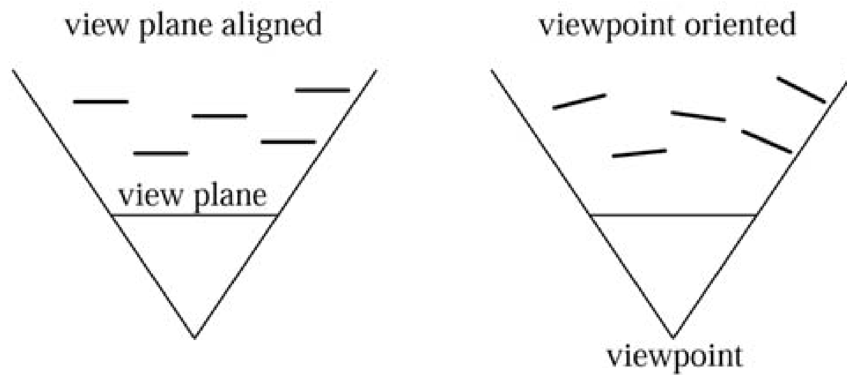

Figure 19 - Example of the Screen Aligned and Camera Aligned Billboard.

Finally we have the Axis Aligned which based on the axis given the billboard will make the proper transformations to face the particles towards it, as shown in *Figure 20.*

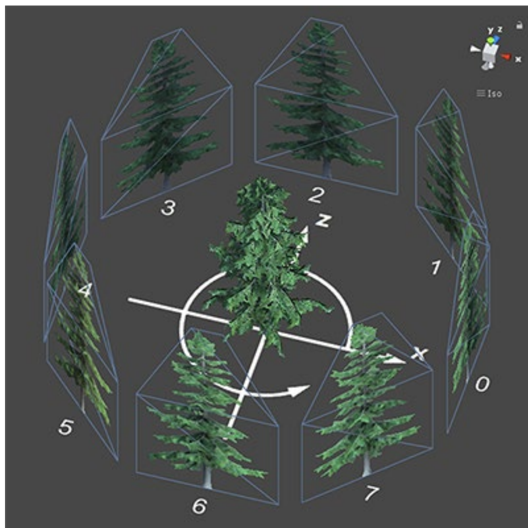

Figure 20 - Example of the Axis Aligned Billboarding from Unity Engine.

The EmitterBase also calculates the distance of each particle to the current camera and assign it to the own particle variable, distance to camera.

And last through the UpdateWithNode method with the provided information from the nodes, it checks if the EmitterReference is active or not, sets the texture to the material stored in the emitter reference, and it determines the kind of alignment.

- **EmitterSpawn:**

The Emitter Spawn only uses the Update functions, using a timer based on the application delta time, each time it reaches the amount specified by the spawn ratio it spawns a new Particle. The spawn ratio variable is modified through the UpdateWithNode method.

- **ParticlePosition:**

This module only modifies the spawn position of the particles, so it won't need to use the update functions at all.
It has two possibilities to modify the spawn position; with an straight given position from a node or receiving as input a game object.
When receiving a game object, the module looks for a random vertex from the model mesh and gives it as a position to the next particle to spawn, thus spawning particles from each vertex of the mesh at random intervals.

- **ParticleSize:**

Similar to the EmitterSpawn module, the ParticleSize only influences the particles right before spawning by adjusting the size.
This module applies the given size from a node, it uses the UpdateWIthNode function to check at runtime if the user modifies the size.

- **ParticleColor:**

The behaviour of this module resembles to the ParticlePostion, it can either receive a direct color for the next particle to spawn or a color meant to be modified overtime.
At the current stage of the project the color overtime can only go from one color to another, to apply that logic the module interpolates between the two colors (including the alpha transparency) with the normalized lifetime of each particle.

- **ParticleLifetime:**

The ParticleLifetime Spawn function assigns to each particle the lifetime value stablished by the user, and the Update increases a counter, also stored in the particle, each frame. Taking both variables into consideration the emitter can check when a particle reaches their lifetime limit and reset them, ready to spawn again.

- **ParticleVelocity:**

This module uses all of the base functions to give the particles the proper direction and speed logic.
To do that upon spawning the particle it assigns a given velocity from a node, more specifically it receives a float4 in which the first three components indicate the direction and the fourth component the speed.
This module also takes into consideration a force vector stored in the Emitter Reference, which will be modified by a few nodes. This force vector carries both direction and intensity of the pulling force, once again interpolating with the particle lifetime the module does the proper calculations to lerp from the particle velocity to the resulting force.

5.3.3 Emitter & Emitter Instance

The main components of the particle system are the Particle Emitter and Emitter Instance. The Particle Emitter will work as a placeholder of all the necessary data and modules to produce as many particles as needed with the desired configuration, and the Emitter Instance as its name suggests, instantiate a copy of the Particle Emitter and executes the actual logic of spawning, updating and killing the particles.

The Particle Emitter holds a vector filled with all the aforementioned modules, the actuals in charge of modifying the particle properties and give them the expected

behaviour by the user. It also has a Resource Material a specific resource of the ASE engine containing a texture, shared by all the particles spawned using this Particle Emitter as reference, a color specific for each particle managed by the ColorModule and a shader, a simple one in charge of rendering a texture into screen, also shared by all the particles. This material along with the modules stored in the Particle Emitter will be shared by all the particles spawned by the same Emitter Instance.

We just specified that the emitter instance is the one managing the production and removal of the particles, to do so it uses a pooling system. It consists of filling a vector with default particles up to a preestablished maximum, for performance purposes in this project the maximum amount of particles allowed in screen are 5000. Upon spawning a new particle, we retrieve the first empty particle in the vector using an index defined by the number of particles already spawned or alive, right afterwards we pass that particle as a parameter to the Spawn function of every module in the emitter filling the empty particle with proper values. On the other hand, to delete or destroy a particle that surpassed their lifetime limitation, we send it to the back of the vector swap it with the last one alive and subtract one to the active particles index, thus that particle won't be rendered anymore into the screen, and the next time the spawn function reaches out to spawn that same particle, it will override the data in it.

5.3.4 Resource Particle System

The resource is intended to work with the whole Resource Management system, saving the resource with all the data from the Particle System into a custom library file, so it can be loaded at any time with the previous modifications, and saving resources inside the engine dependencies. Although for this project, due to time limitations this feature was relegated out of the objectives scope, so the functionality of the Resource Particle System varies from the others in the engine. It also holds a vector (from the standard template library) of emitters and will be responsible of initializing those added to the vector with the necessary Particle Modules.

5.4 Rendering and Instancing

Once reviewed how the particle systems manages all the particles before sending them do the renderer, this section will cover how they are being rendered and the optimizations made to improve the engine performance and support more particles on screen.

In the first versions of the engine tool the rendering of the particles was done individually, meaning each particle with its own mesh and material had a draw call to render it into the screen. That method proved handy for the first versions; however it offered a poor performance, it could be substantially improved and optimized.

While exploring possible optimizations, this rendering technique came up. Instancing is based on the idea of having one object or in our case one mesh stored in a buffer, and use the data of that mesh to render as many instances of it as needed in a single draw call, thus taking the most profit out of our CPU and GPU. Also, in addition other buffers are required to describe the particularities of each instance of that mesh.

To Instantiate particles first of all we needed a base mesh, a square to render the appropriate texture, shaped with two triangles. To build this base mesh we stored inside a buffer the vertex positions defining both triangles as shown in *Figure 21*, and inside another buffer the UVs coordinates necessary to render the texture in it as shown in *Figure 22*. This buffer initialization is done once, at the start of the application, they won't be updated, thus to OpenGL they are declared static being reused as much times as necessary without modifying them.

```
particleVertices = {
    -0.5f, -0.5f, 0.0f,
    0.5f, -0.5f, 0.0f,
    -0.5f, 0.5f, 0.0f,
    0.5f, 0.5f, 0.0f,
};
```

Figure 21 - Particle Mesh vertices buffer

```
particleUVs = {
    0.0f, 0.0f, 0.0f, 1.0f,
    1.0f, 0.0f, 1.0f, 1.0f
};
```

Figure 22 - Particle Mesh UVs buffer

To differentiate the particles and add the proper exclusive properties to each, we make use of a couple more buffers although this time they need to be updated dynamically as they will be filled each draw call with new data.

```
glBindBuffer(GL_ARRAY_BUFFER, particleTransformBuffer);
glBufferData(GL_ARRAY_BUFFER, MAX_PARTICLES * sizeof(float4x4), NULL, GL_STREAM_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, activeParticles * sizeof(float4x4), particleTransformData.data());

glBindBuffer(GL_ARRAY_BUFFER, particleColorBuffer);
glBufferData(GL_ARRAY_BUFFER, MAX_PARTICLES * sizeof(float4), NULL, GL_STREAM_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, activeParticles * sizeof(float4), particleColorData.data());
```

Figure 23 - UpdateParticlesBuffer function used to fill the buffers with the new dat.

The *Figure 23* displays the necessary OpenGL functions calls to update the dynamic buffers with the new data. The *particleTransformBuffer* and *particleColorBuffer* are filled with the data stored in the vectors *particleTransformData* and *particleColorData*. These vectors are also updated at the beginning of the draw call with the information of all the particles to draw, they serve to fill the buffers with the collected data and at the end cleared allowing to write on it again on the next draw call.

Once the buffers contain the particle data, comes the assigning of each buffer so the particle shader receives the data organized to render. To do so there are couple of modifications or steps yet, first of all we need to tell OpenGL which buffer is for the base mesh and which will be modified dynamically, in *Figure 24* we can identify how after enabling the proper vertex attribute array and pointer we need to call an extra function. The glVertexDivisor, the first parameter is the index of the vertex attribute, it also points at the location of the layout in the shader. The second one is the divisor, it will tell OpenGL the rate at which Vertex Attributes advance, meaning that if it is 0 the index will advance once per vertex (used for the base mesh buffers) on the other hand if it is non-zero it will advance once per divisor instances of all the vertices being rendered (used for the specific characteristics buffers).

```
glEnableVertexAttribArray(2);
glBindBuffer(GL_ARRAY_BUFFER, particleColorBuffer);
glVertexAttribPointer(2, 4, GL_FLOAT, GL_TRUE, 0, (void*)0);
glVertexAttribDivisor(2, 1);
```

Figure 24 - Use of the OpenGL function glVertexDivisor.

Finally comes the last step to have the instancing ready, here instead of using the common *glDrawArrays* we swap, as shown in *Figure 25,* to *glDrawArraysInstanced* with the amount of particles to render as the new last parameter. This function is equivalent to looping through all the particles and calling *glDrawArrays* altough much fasted and efficiently.

```
glDrawArraysInstanced(GL_TRIANGLE_STRIP, 0, 4, particles.size());
```

Figure 25 - Use of the OpenGL function glVertexDivisor.

5.4.1 Instancing implementation results

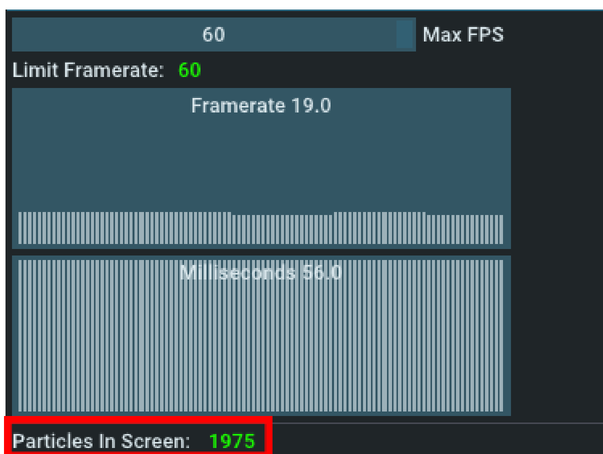

Figure 26 - Performance without Instancing

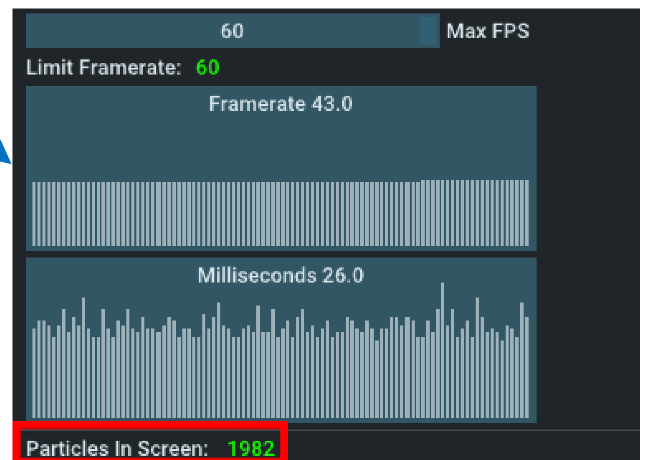

Figure 27 - Performance with Instancing

As we can observe in the previous *Figure 26* and *Figure 27,* the leap in performance once the instancing was working, was considerable.

While rendering the same amount of particles in screen (See the red square in *Figure 26* and *Figure 27)*, the performance in framerate grew by a 200% and the milliseconds it took to perform an update was reduced by 50%, allowing the engine to produce a wider amount of particles at the same time while keeping an stable framerate.

5.5 Node Editor

In this section of the documentation, we will go through all the features implemented in the Node Editor. From the implementation of the library into the engine and how it was modified to fit the project objectives, through the structure the Node Editor follows and the interactions with the Particle System.

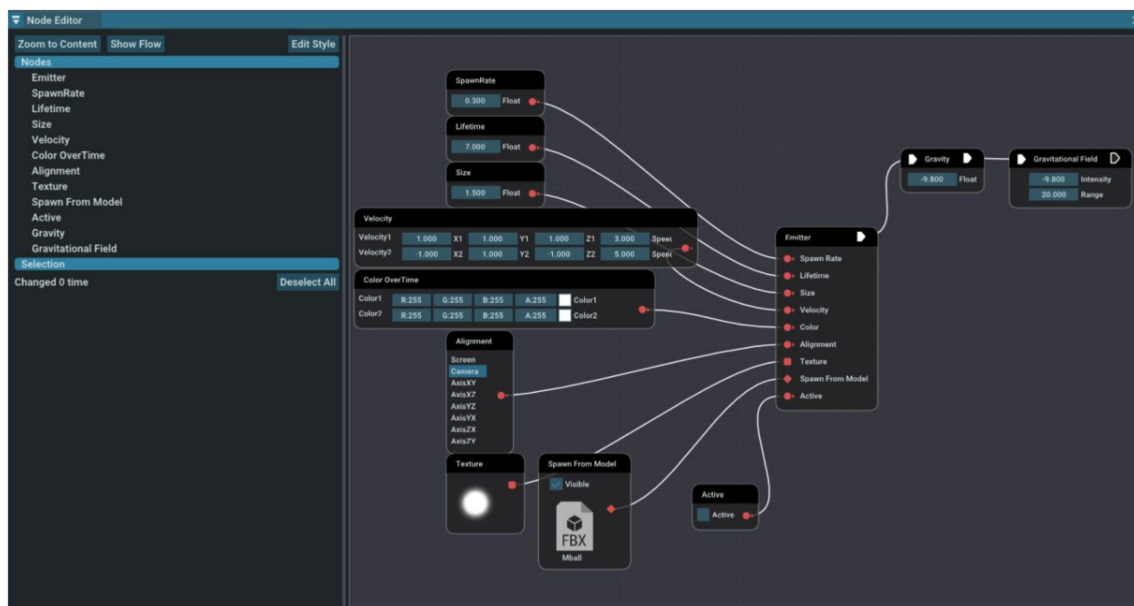

Figure 28 - Example of a complete Node Based Particle System window.

5.5.1 Pin (Base Class)

A pin it's the structure used to connect two nodes, it must be between an Output pin from one node and an Input pin from another one, creating a link between them represented as a curved line from the pins position. As we can see in *Figure 29* both pins represent by an spheric shape and circled in red are connected by an straight white line, thus being the link between them, framed in green.

In this project the Pins will be responsible for transmitting all the necessary information, variables or functionality from one node to another.

The Pin parent class will be defined by an ID (Identifier Number) a Name, a pointer to the parent Node and two defining categories:

- **PinKind:** It splits all pins into Inputs and Outputs. Inputs for all of those positioned at the beginning of the node receiving information when linked to an Output Pin of another node. The Outputs will carry the desired variables to pass along, they will be modified and updated at the parent Node request, and if linked to anther pin input both pins values will be updated.
- **PinType:** It defines the type of information or variables the pin will hold for modification or to transmit to another pins. In order to connect two Pins they also need to share the same PinType.

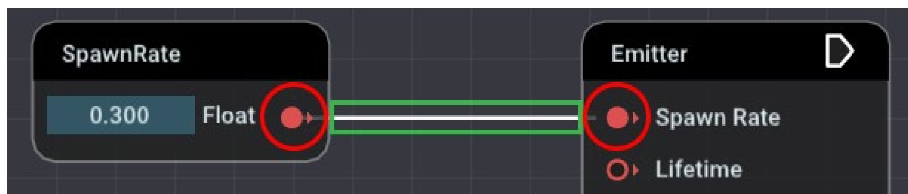

Figure 29 - Core Modular Structure Diagram.

5.5.2 Node (Base Class)

The Node object is a visual representation of the modifiable variables used to fill the necessary modules for an Emitter to work.

It acts as the parent class for all the custom nodes necessary to build a Particle System with the necessary modules. Each node has a vector containing the input and output Pins, a name, an identifier number and a type definition for each derived node.

It also contains a pure virtual function, Draw(), meant to be inherited and overrided. It has all ImGui necessary functions to create the Nodal structure mixing functions from the Node library and some of our own modifications to build the necessary custom nodes, all the node functionality is thought to be compatible and operate with the original library functions.

Each node can hold multiple pins, as we've seen in the previous section, objects with similar structure but totally different functionality between each other.

The link between nodes or more specifically its pins, are simple structures in charge of holding the connection between them. Two linked pins can share the information stored, however if there is no link stablished, they can't hold any relation whatsoever.

5.5.3 Emitter Node

The emitter Node is the one assigned to each Emitter Instance, in every particle system variant the emitter node will always be at the center, it's a unique node used in every case. Duplicating the emitter node will add another emitter to the current Particle System Component, thus generating two sources for particles to spawn from. It functions as an intermediary between all the information received by the connected nodes to its inputs Pins and the system modules.

In *Figure 30* we can appreciate several input pins, each referring to a key aspect to create and shape a Particle Emitter, and a single output Pin Flow which will hold a pointer to the Emitter Instance referred by the parent node. By connecting the PinFlow to another node input PinFlow it grants that node, access to the Emitter Instance without having any direct relation with it. This feature is used by nodes such as Gravity Node and Gravitational Field Node. In further sections we will dive in depth into all those pins logic.

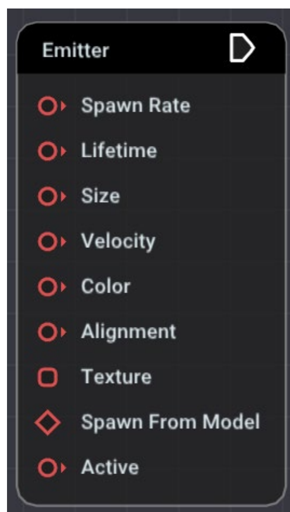

Figure 30 – Emitter node with all the necessary Input Pins

5.5.4 Velocity Node

The velocity node takes as input two float4, each of those will use the XYZ component to define the direction of the velocity and the W component for the speed, trick inspired in Unreal, used to save space and unnecessary calculations.

The Velocity node uses as an output a PinFloat4Array; which holds an array of 2 float4, which once linked to the Emitter node will be passed to the Velocity module for further calculations.

Using a custom Randomize function, each float composing the vectors (X,Y,Z,W) will be used as a minimum and maximum range to calculate that random value in between, having as a result an almost unique direction and speed for each particle spawned.

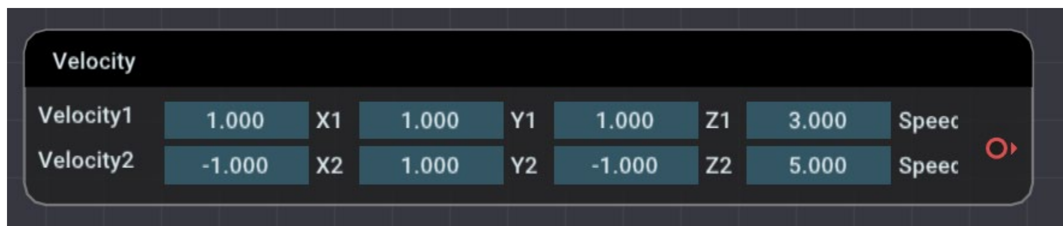

Figure 31 - Velocity and Color Nodes with two float4 as inputs.

5.5.5 Color Node / Color Overtime Node

The Color and Color Overtime Node follows the same structure as the Velocity Node. However each component will be assigned to a RGBA value, thus limiting the value range between 0 and 255, covering the whole color scheme.

Both nodes are meant to be linked to the Color input pin at the Emitter node, each requiring different logic and results. The Color Node will provide each newly spawned particle with a randomized color between the two inputs. On the other hand the Color Overtime Node will interpolate between the two input colors throughout the particle lifetime, hence beginning with the first color and dying with the second color

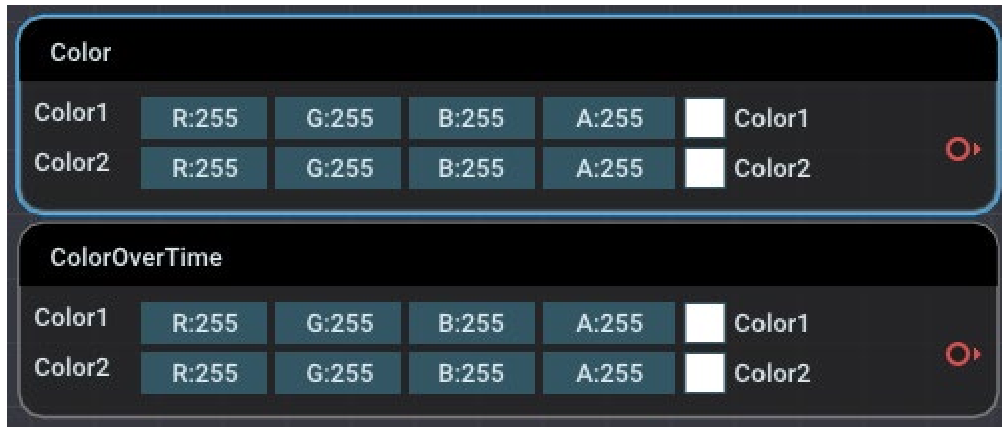

Figure 32 - Color and Color Overtime Nodes with two float4 as inputs.

5.5.6 Alignment Node

This Node works as a simple selectable from an array of strings, each one determines a different alignment of the particles. The alignment calculations will be held by the particle module, the node only specifies the one selected.

It uses a Float Pin to pass along the selected number of the enumerator, the module is the one in charge of interpreting it and apply the transform with the correct rotation to each particle in screen.

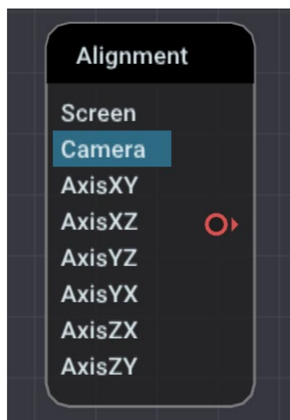

Figure 33 - Alignment Node with all the possible alignments.

5.5.7 Texture Node

The texture Node holds a pointer to a Texture resource which is being used by the ImGui::Image() function to display the currently assigned texture. By using the ImGui drag and drop functions we can change the particles texture just dragging a texture from the Assets Explorer Window to the image inside the node. (As we can appreciate in *Figure 34 t*he Assets Explorer Window and the Node Editor Window are positioned by default one next to the other to facilitate the usage of this feature).

The Texture Node in order to transmit the information, uses a PinTexture, once linked to the corresponding input pin in the Emitter Node, it assigns updates the assigned texture to the material stored at the Emitter Reference with the new one. The material is shared by all particles, even though color is an individual modification for each particle.

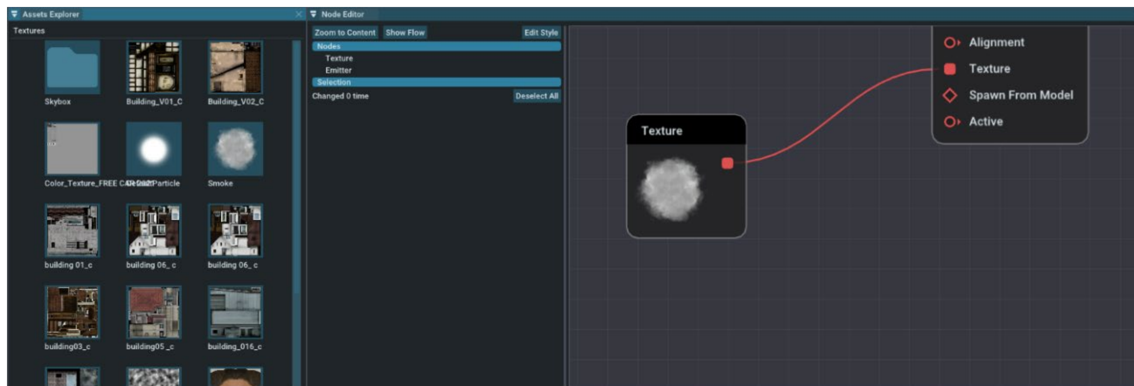

Figure 34 - Positioning of the Assets Explorer Window and the Node Editor Window next to each other.

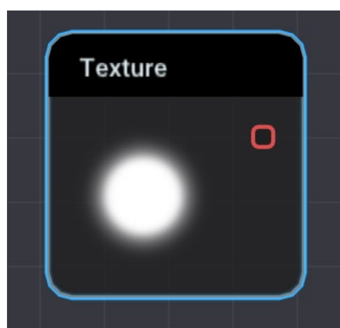

Figure 35 - Texture Node.

5.5.8 Spawn From Model Node

The main idea behind this Node is to fulfill one of the specific goals of this research project, integrate and use external geometry as spawn origin for the particles. Inspirated by many games using particles to vanish models or perform effects with very specific shapes, here it is an attempt to recreate some of those effects.

For example (Renard, 2021) in this GDC talk Rupert Renard exposes how Santa Monica Studio used several VFX and rendering techniques to achieve such result (The technique referenced is shown in the first 3 min).

The SpawnFromModel Node shares some structural logic with the Texture Node, using similar functions to display the currently used model and the drag and drop from ImGui to modify it. I also features a Boolean on top of the image display, in charge of activating or deactivating the visibility of the imported model in the scene.

The Spawn From Model Node imports any model assigned from the Assets Explorer Widow as a new Game Object into the scene, giving the user full control over it to relocate and edit it at compliance.

To communicate with the Emitter Node it uses a GameObject Pin, passing a pointer to the object created in the scene and holding the model information in the Mesh Component. The Module Position will be the one assigned to retrieve all the vertex information used to render the mesh in the scene, interpret it and use it as unique spawn points for new particles.

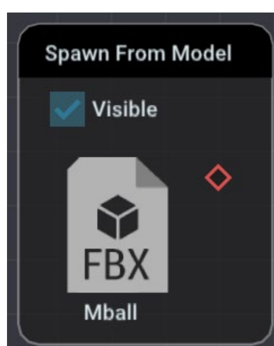

Figure 36 - Spawn From Model Node.

5.5.9 Boolean Node

The Boolean node is thought to be a generic and reusable node in multiple situations. Using an ImGui checkbox the user can modify the Boolean value transmitted through the Bool Pin.

With the current state of the development the main purpose of this node is to alternate the active state of the emitter.

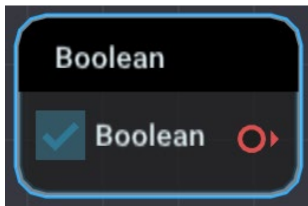

Figure 37 - Boolean Node.

5.5.10 Float Node

The Float Node it's a multipurpose node used for several necessary variables to build the particle system, such as the particle Spawn Rate, Lifetime or Size.

Uses a slider to modify the float and pass it through the Float Pin.

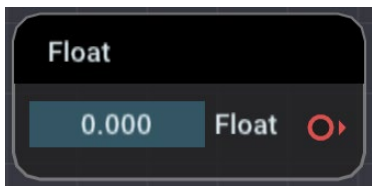

Figure 38 - Float Node.

5.5.11 Vector3 Node

This node called Vector3 was the original velocity node meant to work on multiple situations. However as the project continued its development and the already reviewed Velocity Node was implemented, the Vector3 became useless in the current environment, it does not have any purpose currently.

In the *Figure 39* right below we can see how through three sliders it modifies each component of a float3, if any modification takes place in runtime the PinFloat3 will transfer the modified variable to the linked node.

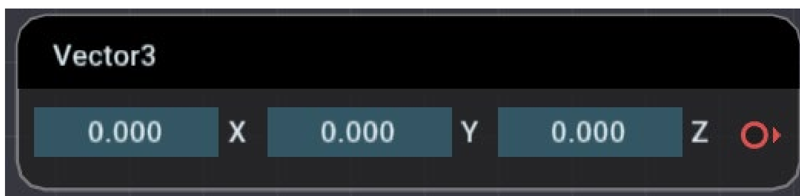

Figure 39 - Vector3 Node

5.5.12 Gravity Node

The Gravity Node along the Gravitational Field Node are the only two nodes relying on PinFlows to remain active. These nodes logic is mostly inspired by how Unreal Engine makes use the nodes flow in their Blueprints. Based on the nodes connected by the current flow they determine which nodes are being executed.

For this project the implementation of the flow was adapted to the needs of it, the PinFlow carries a pointer to the EmitterReference stored in the EmitterNode, being the first node with only an output PinFlow therefore marking the beginning of the flow.

Their structure and behaviour differ from all the other nodes described, even though their logic will indeed influence all the particles in the scene they don’t relate directly to any Particle Module.

By modifying the float slider in the Gravity Node, we alter the Y component of the Force vector stored in the EmitterReference, this vector is always taken into consideration while performing the proper velocity calculations of the particles. So, the value altered in the node will directly influence how the particles behave, thus dragging them downwards if the floating number is negative or upwards if it is positive.

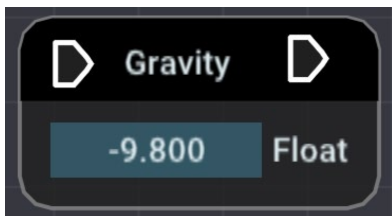

Figure 40 - Gravity Node.

5.5.13 Gravitational Field Node

The Gravitational Field Node shares most of the structure and logic with the Gravity Node, it needs to be linked to an output FlowPin holding the current flow in order to be active and influence the current particles.

Nonetheless the Gravitational Field only impact the particles inside a defined range, in order to do so, upon creation inside the node constructor, it creates an empty GameObject and saves a pointer to it. Having an empty GameObject allow the user to reposition it at will, creating any desired effect bound by the two modifiable floats:

The range works around the object position, whenever a particle finds itself within that range and the object it becomes affected by the intensity of the gravitational field. Like with the Gravity Node the Intensity float value will determine if the particle is repelled or attracted towards the object position.

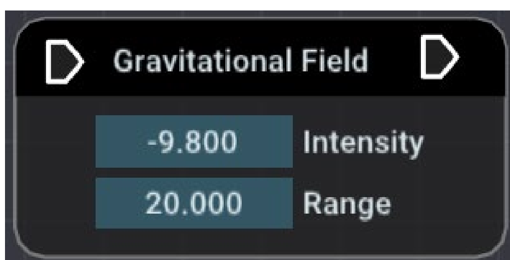

Figure 41 - Gravitational Field Node

5.6 Node Editor Window

In this section we will revisit more in depth the window in charge of rendering all the nodal structure we just reviewed, it creates, draws and cleans up all the nodes, pins and links among other necessary features. We will focus on the implementation made specifically for this project however we will briefly cover the functionality provided by the library.

The main objective of this window is to give the user full control over the Particle System without having to leave the window bounds, all the modifications and input variables from the nodes are already implemented to function and update upon modification.

5.6.1 Library Usage

The grid and node interaction are mainly handled via functions provided by the library. It allows the navigation through the grid, the reposition of any node, the connection from pin to pin with links, the group selection and the deletion of either nodes or links.

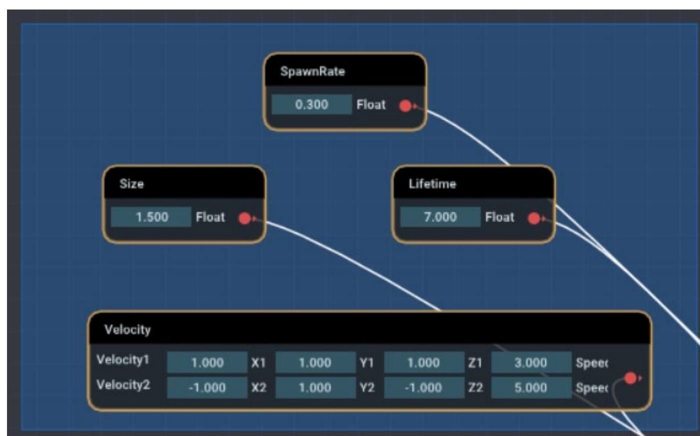

Figure 42 - Group selection inside the Grid panel.

For example the group selection as shown in *Figure 42*, the library making use of ImGui functions and other custom logic gathers every node within the boundaries of the selection area traced by the user, thus by moving either node all of them will reposition following the same motion.

Another example of the usage of the library in this project is the linkage between pins, by using the pins position inside the grid and inside the window itself, the custom function, ImRect_ClosestLine traces the most efficient path connection between the two pins while keeping a smooth delineate using Bezier curves. We can appreciate the result in *Figure 43.*

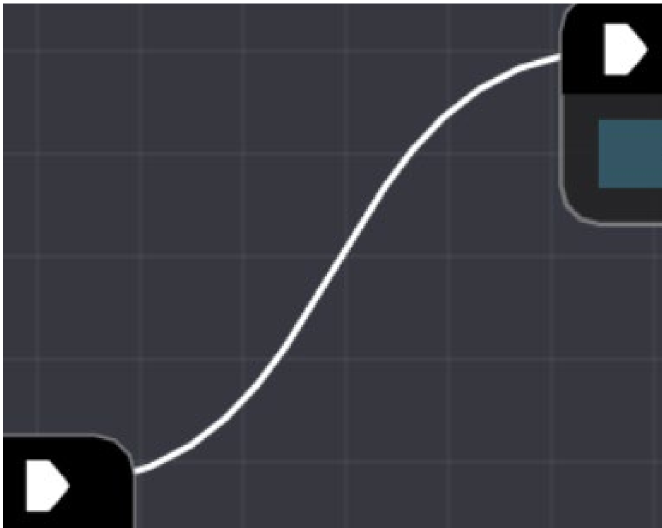

Figure 43 - Example of the resulting Line rendered when two Pins are Linked.

5.6.2 Draw Call

The class Window has a pure virtual function called Draw meant to be overridden by every child window, it focus on managing all the render processes.

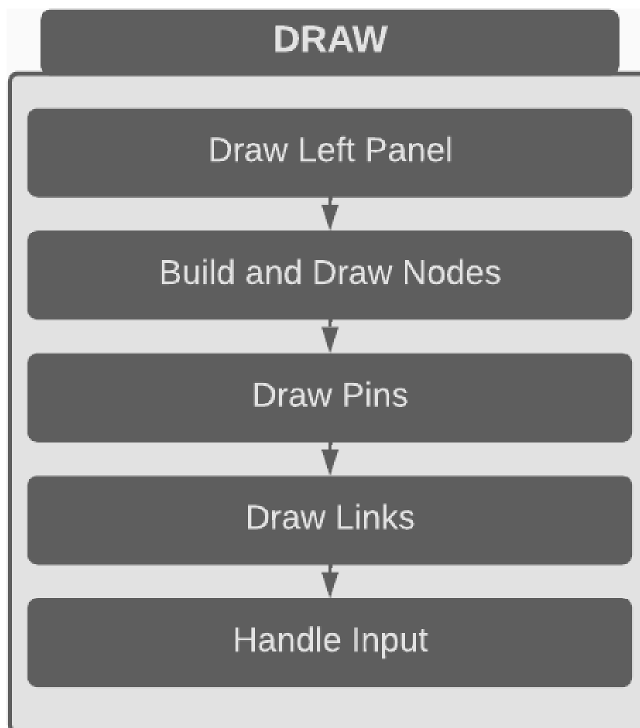

Figure 44 - UML of the Draw function of the Window Node Editor.

As we can observe in *Figure 44* the Window Node Editor Draw call definition can be split into 5 logic groups. First of all we render the left panel which shows a list of all the nodes in screen, in the next section we will dive further into that topic.

Right after comes the building and drawing of each node. To properly explain this block we need to review the Node Builder. The Node Builder is a custom class expected to function as a constructor or renderer of any raw node, meaning that given a few parameters with a single Draw function definition could render any given node, it ended up being discarded due to the complexity and differences between each custom node. Now each node holds their own creation and drawing logic. Thus, the Node Builder is currently used to draw the nodes headers, backgrounds and outlines, summing up, only the common elements.

Returning to the main topic, the functions Begin, and End from the Node Builder will handle the basic structure of each node, and in between the explicit Draw call of each node renders the custom structure to either modify the information or receive it.

The next block is contained within the first one even though it is necessary to mention it as a whole. The Pin drawing itself is done through the function DrawPinIcon from the Node Editor Window, it classifies each pin based on the preestablished PinKind and assign an specific shape to each, right afterwards the DrawIcon performs the actual render using library functions. This draw call needs to be surrounded by two functions indicating the begging of the pin drawing and the ending, telling the ImGui current context what needs to be drawn right afterwards, and where.

The link drawing is rather straightforward, for every successfully link between two pins, following the shortest path it draws a line delineating a smooth curve. This block is mostly taken care of by the library.

In the last place we've got the input handling, in this section with only the mouse position inside the window and the user input as parameter we can take care of; the movement through the grid, the group selection, the popup creation menu and the deleting of any link or node.

5.6.3 Window Usage – Left Panel

In this section we will focus on all the functionality and usage this window has to offer.The window splits in two panels, the right one is the grid where all the nodes are rendered and the user can navigate freely and the left one is a list of all the nodes added to that grid along some buttons for further personalization and action shortcuts for the user.

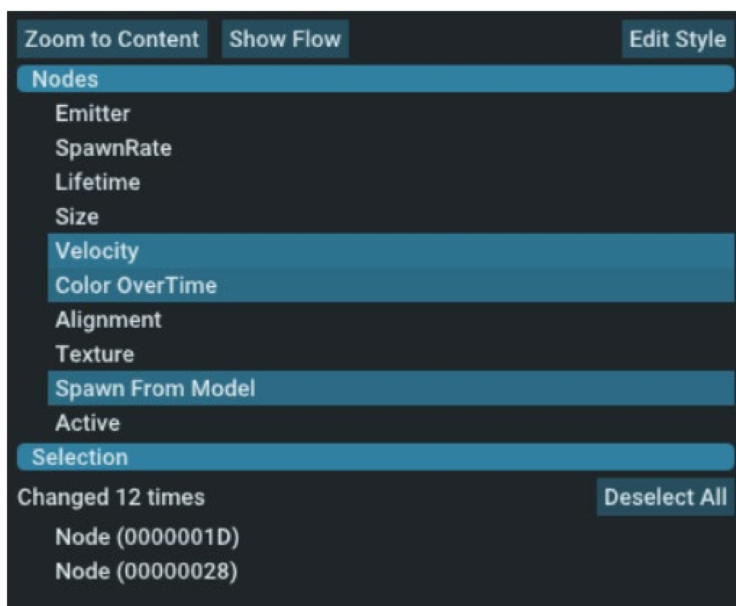

Figure 45 - Left Panel

In the *Figure 45* we can observe a screenshot of the left panel. With this visual support in mind we will review each feature.

On the upper section, somewhat resembling of a header, we find three buttons:

- **Zoom to Content:** Given the current size of the grid, this button gathers all the nodes added to the grid and adjust the zoom of the viewport to fit all of them, allowing the user to visualize the whole nodal structure with a single glance.

- **Show Flow:** It provides visual aid to the user in order to visualize the current flow of the linked nodes as shown in *Figure 46*, the core logic is handled by the library.

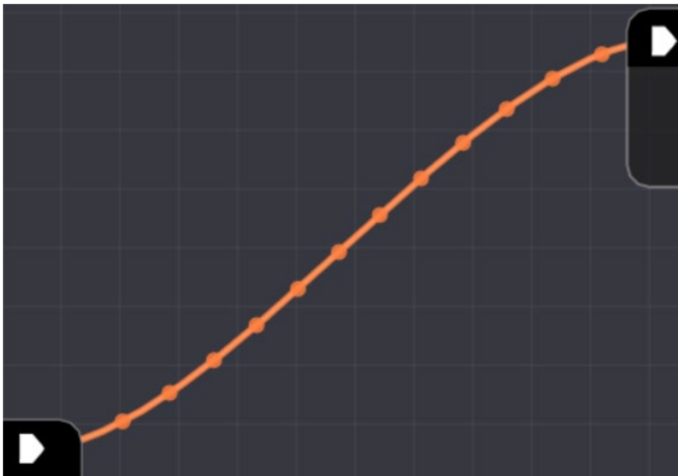

Figure 46 - Direction of the Flow between nodes.

- **Edit Style:** This button opens up an auxiliary window dedicated exclusively to style modifications, as we can discern in the *Figure 47* the window contains a series of sliders to simplify the modifications and adjustment of these indicators. The implementation is quite direct, each slider modify a variable used by le library to shape and structure the grid and all nodes inside.

Figure 47 - Style Editor of the Window Node Editor.

After reviewing the buttons in the upper area, following the *Figure 45* in a downwards direction we stumble upon a list of all the nodes added to the grid, with those selected being highlighted with the characteristic ASE blue tone, used in most of the GUI. With a glimpse the user can identify the present nodes, select them, focusing on a single one or even a group selection by holding Ctrl in the keyboard and clicking on the nodes requested to join the selection.

Finally the last region of the left panel shows the nodes IDs of the selected nodes, serving as merely information for the curious users, it doesn't have any use outside the internal management of the nodes.

5.3.3 Window Usage – Right Panel

Over the last few sections, we have been mentioning and referencing the grid for multiple purposes in several occasions, even though we can also refer to it as the right panel of the Node Editor Window, which essentially is a very large extension where the user can organize and space their nodes at will.

The grid has a few possible inputs in order to navigate through the available space and interact with anything in it:

- Mouse Wheel: Zoom in or out based on the direction spinning the wheel.
- Mouse Left Click: Open up the creation menu, see *Figure 48.*
- Hold Mouse Left Click and move: Navigate through the grid, at the mouse motion speed.
- Hold Mouse Right Click and move: Delineate a blue square with the origin at the first click extending all the way through the mouse motion until releasing the right mouse button, selecting everything contained in the defined bounds.

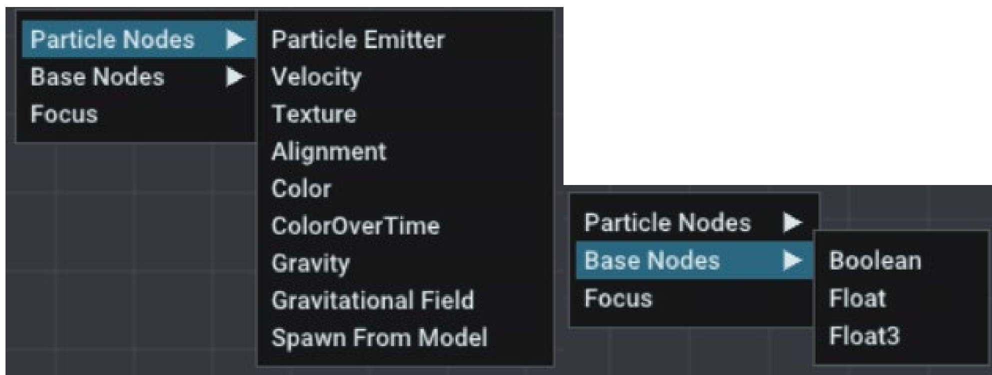

Figure 48 - Node creation Node (Right Click inside the Grid)

Figure 48 shows the Node creation menu which splits into the particle nodes, the basic nodes, and the focus. The focus works exactly as the button Zoom to content from the left panel.

6. Conclusions

This project attempted to build from scratch an accessible user-friendly tool, capable of recreating complex particle simulations imitating certain features present in professional VFX software.

The objectives set in the initial planning phase, were expected to be fulfilled given the initial time schedule and tasks distribution. Once reached the closing day of the last milestone we can assert that most of the objectives were satisfied, even though some of the expected features were left out, the final version brought a functional node-based editor capable of creating multiple particle simulation effects.

The main objective that was left out due to time restrictions and personal priorities, was the serialization of the Particle System and the Node placement in the grid. The idea was to save both in a json file so whenever the user builds a simulation effect of their liking the can save and recover it at any further session.

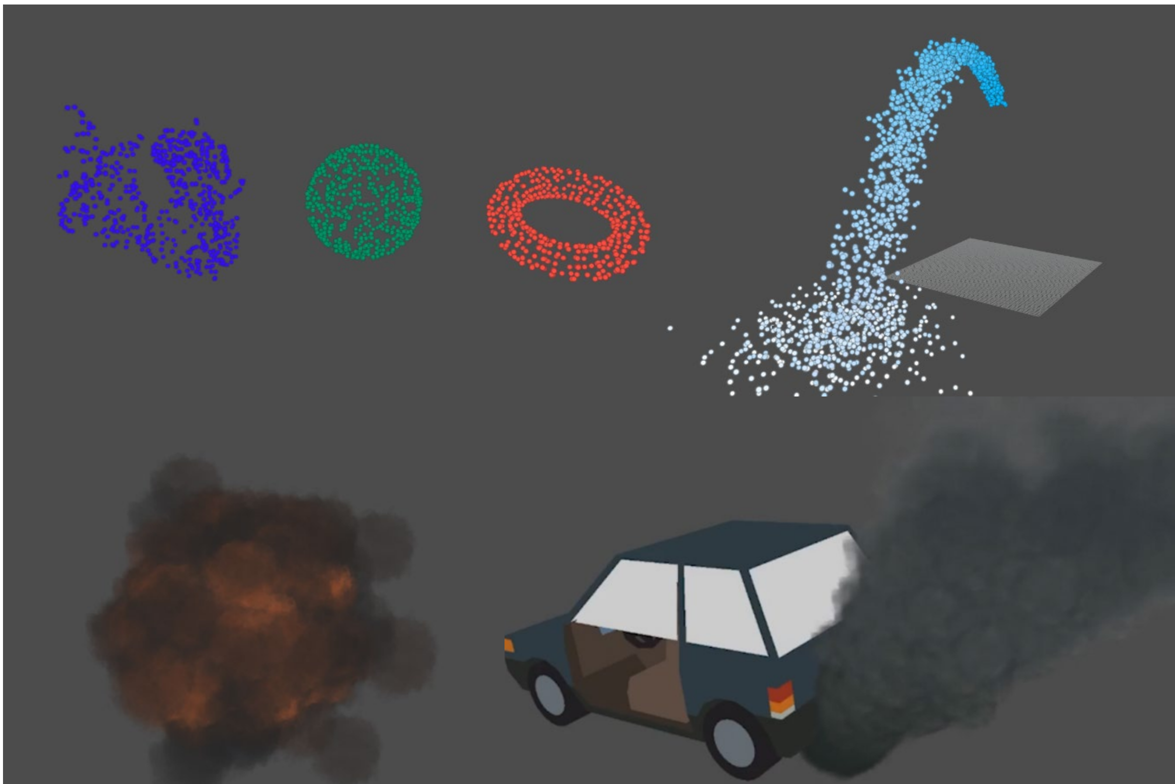

Figure 49 - Results achieved using the final version of the Node-Based Particle System.

- ✓ Develop a solid base engine to build the tool with.
- ✓ Adapt the node library and expand it with the necessary features to support the particles interactions.
- ✓ Build the Node-Editor panel.
- ✓ Create the particle nodes with the necessary attributes.
- ✓ Implement a basic Particle System.
- ✓ Expand the Particle System and adapt it to the node graph workflow.
- ✗ Implement the serialization of the Particle System variables or attributes stored inside the nodes.
- ✓ Allow multiple particles spawner and a high number of particles in scene.
- ✓ Implement the integration and use of external geometry inside the particles simulations.
- ✓ Optimize the performance to ensure stable FPS.

Even though we added instancing to improve performance there are other methods much more effective, such as GPU particle rendering, as we have explained before GPUs are an extremely powerful tool when it comes to perform multiple processes simultaneously. although this time, we attempted to adapt the planification to implement this feature in the engine, it presented several obstacles made us discard the feature completely.

Despite the objectives left out, the tool is capable of producing simulation such as the ones in *Figure 49* proving most of the objectives fulfilled and even couple additional from the first ones stated.

We made wide research about Particle Systems, across multiple software's, their potential and implementation, the possible optimizations and how to make all of this manageable from a Node-based graph editor, we reached that goal. The tool was set to be an only display set up where to portray all the knowledge gathered along the

development given the starting point was almost null experience in the area, outcome provides a visual aid of the current understanding of the author about the topic.

Even though the objectives were almost accomplished, the bar and expectations weren't set high due to personal time schedule issues. With more time investment, the resulting tool could have been much superior with a lot of features left out and new ones that came along the development (as ideas, never actually researched and evaluated). The following section is dedicated to those features that could be implemented with further work on the tool.

7. Next Steps

As recently mentioned in this last chapter of the research we will introduce discarded features that were expected to be in the tool, and others that came as merely and idea. All set aside for future development.

The current system spawns particles on the surface on a mesh with a starting velocity and stores a copy of each particle on CPU, update them sequentially, and lastly upload them to GPU for rendering each frame. The GPU system, with the help of compute shaders to perform calculations that previously were done through CPU, reduce the amount of data moving between the system and the GPU (render calls). As we already talked about the capability of the GPU to perform multiple processes at once, it would allow for a huge number of particles on screen and improve the performance of the engine obtaining a more stable framerate.

Figure 50 - WickedEngine GPU Particles simulation

Another future feature, is the Particle System Window, meaning the editor of the particle system would open a full window which only purpose is to contain, the Node Editor, the Viewport Window, the particle system information and the engine performance. Giving the user more space and mobility to edit and visualize the simulation. The *Figure 51* makes reference to the Unreal VFX editor: Niagara, with approximately the windows previously mentioned.

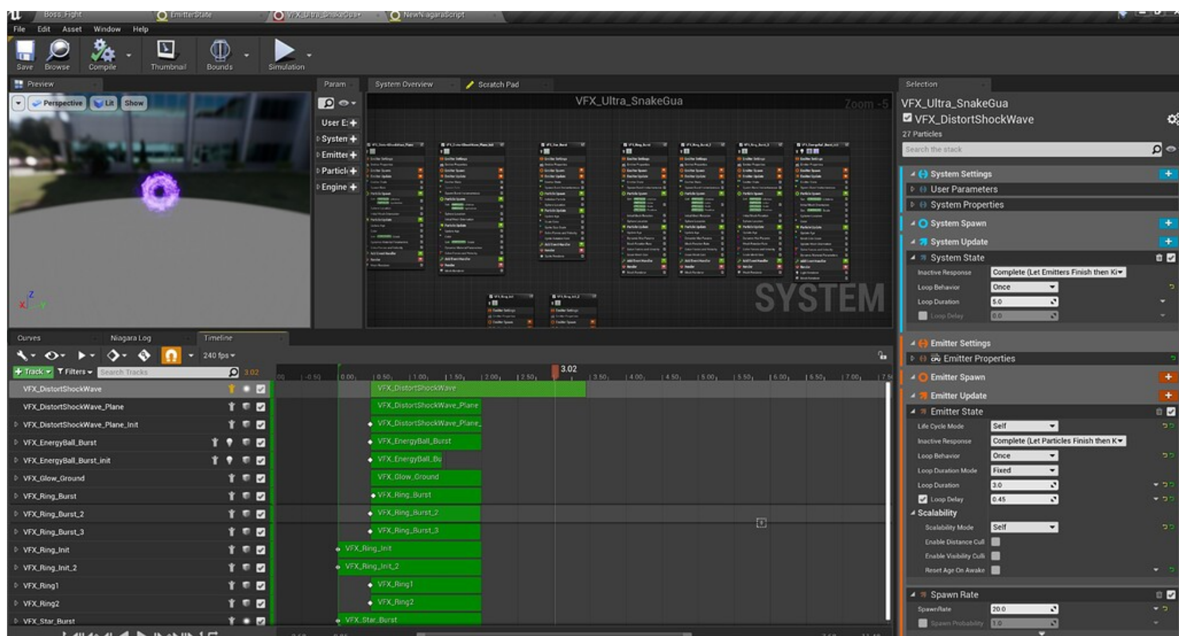

Figure 51 - Unreal Engine: Niagara editor window

As addition to the possible features to develop in the upcoming future, there are some nodes with extra functionality:

- Particle Destination node: Such node would help creatin flux simulations, from the spawn point travelling all the way up to the destination position while taking into consideration the initial speed and added forces.
- Size Overtime or Random between two sizes: It wouldn't be hard to implement such nodes given that most of the functionality is already included. It would give another degree of freedom and personalization while creating the simulations.
- Collision Handler/Obstacle: Node in charge of defining an area, within that area particle collision are taken into consideration when colliding into other meshes or empty colliders defined by the node.

8. Bibliography

Academy, K. (2022, April 5). *Intro to particle systems*. Retrieved from https://www.khanacademy.org/computing/computer-programming/programming-natural-simulations/programming-particle-systems/a/intro-to-particle-systems

Blender. (2022, March 5). *The Freedom to Create*. Retrieved from https://www.blender.org/about/

Burg, J. v. (2022, April 6). *Building an Advanced Particle System*. Retrieved from https://www.gamedeveloper.com/programming/building-an-advanced-particle-system

DigitalRune. (2022, April 04). *Billboards and Particles*. Retrieved from https://digitalrune.github.io/DigitalRune-Documentation/html/4518dd2c-21ea-4cf8-8dec-8b3a32584743.htm

Epic Games. (2022, March 07). *Cascade Particle Systems*. Retrieved from https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/ParticleSystems/

Epic Games. (2022, March 7). *Niagara Overview*. Retrieved from https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/Niagara/Overview/

Flipcode. (2022, April 04). *Types of Billboards*. Retrieved from https://www.flipcode.com/archives/Billboarding-Excerpt_From_iReal-Time_Renderingi_2E.shtml

Gantt. (2022, February 05). *Gantt Chart tool*. Retrieved from https://app.teamgantt.com/

Glassdoor. (2022, March 5). Retrieved from https://www.glassdoor.es/Salaries/spain-game-developer-salary-SRCH_IL.0,5_IN219_KO6,20.htm

Gross, A. (2022, February 26). *ChilliSource Game Engine Particle System Study*. Retrieved from https://scholarworks.umt.edu/cgi/viewcontent.cgi?article=11859&context=etd

Hacknplan. (2022, February 15). *What is HacknPlan?* Retrieved from https://hacknplan.com/knowledge-base/what-is-hacknplan/

Ignac, M. (2022, March 04). *Nodes*. Retrieved from https://nodes.io/story/

Instancing. (2022, May 15). *Rendering particles with Instancing*. Retrieved from http://www.opengl-tutorial.org/intermediate-tutorials/billboards-particles/particles-instancing/

Lucid. (2022, March 20). *UML tool*. Retrieved from https://lucid.app/

Luten, E. (2022, February 26). *OpenGLBook*. Retrieved from https://openglbook.com/chapter-0-preface-what-is-opengl.html

NatrureOfCode. (2022, April 09). Retrieved from https://natureofcode.com/book/chapter-4-particle-systems/

PayScale. (2022, March 05). Retrieved from https://www.payscale.com/research/ES/Job=Video_Game_Designer/Salary

PayScale. (2024, November). Retrieved from https://www.payscale.com/research/ES/Job=Video_Game_Designer/Salary

Petty, J. (2022, March 03). *What is Houdini & What Does It Do?* Retrieved from https://conceptartempire.com/what-is-houdini-software/

Quilez, I. (2022, March 10). Retrieved from https://iquilezles.org/articles/

Renard, R. (2022, February 09). *Youtube*. Retrieved from GDC Santa Monica Studio Talk: https://www.youtube.com/watch?v=ajNSrTprWsg&t=170s

SideFx. (2022, February 27). *Houdini*. Retrieved from https://www.sidefx.com/products/houdini/

Studiobinder. (2022, March 13). *What is VFX?* Retrieved from https://www.studiobinder.com/blog/what-is-vfx/

Subject, C. -E. (2022, April 02). Particle Systems.

Turanszkij. (2022, June 01). *WickedEngine*. Retrieved from https://github.com/turanszkij/WickedEngine

Unity. (2022, March 04). *Visual Effect Graph*. Retrieved from https://www.youtube.com/watch?v=SUZzJcBIK80

Valve. (2022, March 05). *Particle System Overview*. Retrieved from https://developer.valvesoftware.com/wiki/Particle_System_Overview

WIckedEngine. (2022, June 01). *GPU particle simulation*. Retrieved from https://wickedengine.net/2017/11/07/gpu-based-particle-simulation/

8.1 Libraries

SDL:. https://www.libsdl.org/

Glew (OpenGL): http://glew.sourceforge.net/

ImGui: https://github.com/ocornut/imgui

ImGui Node Editor: https://github.com/thedmd/imgui-node-editor

MathGeoLib: https://github.com/juj/MathGeoLib

Assimp: https://github.com/assimp/assimp

Devil: https://github.com/DentonW/DevIL