INVESTIGATION REPORT

Establishing a Framework for Interactive and Dynamic Vector Graphics Content in Real-Time Video Games

By

Cheng Yi Heng

TP058994

APU3F2408CGD

A report submitted in partial fulfillment of the requirements for the degree of

BSc (Hons) Computer Games Development

at Asia Pacific University of Technology and Innovation.

Supervised by Mr. Jacob Sow Tian You

2nd Marker: Assoc. Prof. Ts. Dr. Tan Chin Ike

27th November 2024

Acknowledgement

I would like to express my heartfelt gratitude to everyone who supported me throughout the development of the project. First, I am deeply thankful to my teammates, whose dedication, collaboration, and creativity which were instrumental in bringing this project to life. Their ability to work together, solve problems, and push boundaries made all the difference.

I also wish to extend my sincerest thanks to my supervisors, Mr. Jacob Sow Tian You and Assoc. Prof. Ts. Dr. Tan Chin Ike, for their invaluable guidance, insights, and encouragement throughout this journey.

I would also like to express my deepest gratitude to the open-source communities behind Bevy, Typst, and Linebender (Vello). Their incredible work and commitment to open collaboration provided the tools, inspiration, and support that made this project possible. The countless contributions, whether through libraries, documentation, or community discussions, were invaluable to overcoming technical challenges and advancing the development of the project. I am profoundly grateful to have had the opportunity to learn from and contribute to them.

Finally, I would like to thank my family and friends for their unwavering support and belief in me, which provided the motivation I needed to persevere through challenges. This project would not have been possible without each of you.

This document is written in Typst.
Source file is on GitHub: https://github.com/nixon-voxell/apu_investigation_report.

Abstract

Despite the widespread use of raster graphics in games, vector graphics remain underutilized and largely absent, with little integration and no established framework for their effective use in modern game development. This project introduces Velyst, a streamlined framework for integrating interactive and dynamic vector graphics into real-time video games. It leverages Typst for vector content creation and Vello for real-time rendering of dynamic vector graphics. By simplifying the process, this framework enables developers to produce high-quality and engaging content without needing to delve deeply into technical complexities. This study employs purposive sampling to collect valuable insights from developers in the game development and interactive application sectors. A semi-structured interview utilizing eight pillar questions was conducted on seven industry professionals with experience in vector graphics and user interface development in order to gain expert insights into current practices, challenges, and opportunities within the field. Additionally, structured online survey questionnaires were distributed to a total of 33 developers to capture a wider range of opinions on user experiences in current state-of-the-art game engines. This uncovers the challenges and opportunities of integrating vector graphics, with insights that contributes to the framework on reducing technical barriers, enhancing interactivity, and highlighting areas for further innovation in the field. This research aims to demonstrate the untapped potential of vector graphics in modern gaming and provide a practical solution for their seamless integration. This approach contributes to advancing infrastructure and fostering innovation, aligning with the goals of *Sustainable Development Goal (SDG) 9*.

Keywords: *Typesetting, Markdown, Workflow, Dynamic content, Typst*

SDG Goal 9: Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation

Contents

Figures

Tables

CHAPTER 1: Introduction

1.1. Research Background

The rapid evolution of game development and interactive media demands tools that can deliver high-quality visuals, dynamic interactivity, and optimal performance. While raster graphics have been the dominant standard for visual content, they often lack scalability and flexibility, particularly for modern applications that require diverse screen resolutions and fluid adaptability. This has led to an increasing interest in vector graphics, known for their scalability, precision, and lightweight nature.

Despite their advantages, vector graphics remain underutilized in real-time game engines, with limited support and optimization compared to rasterized approaches. Challenges such as high computational requirements, technical integration complexities, and performance overhead have deterred developers from fully exploring their potential. Additionally, creating a seamless workflow for integrating dynamic vector content into real-time environments poses further hurdles.

This research seeks to address these challenges by examining the role of vector graphics in game engines, identifying technical limitations, and proposing a framework that enhances their use in real-time interactive environments. By bridging the gap between technological capabilities and developer needs, this study aims to unlock new possibilities for high-quality, dynamic content in games and interactive media.

1.2. Problem Statement

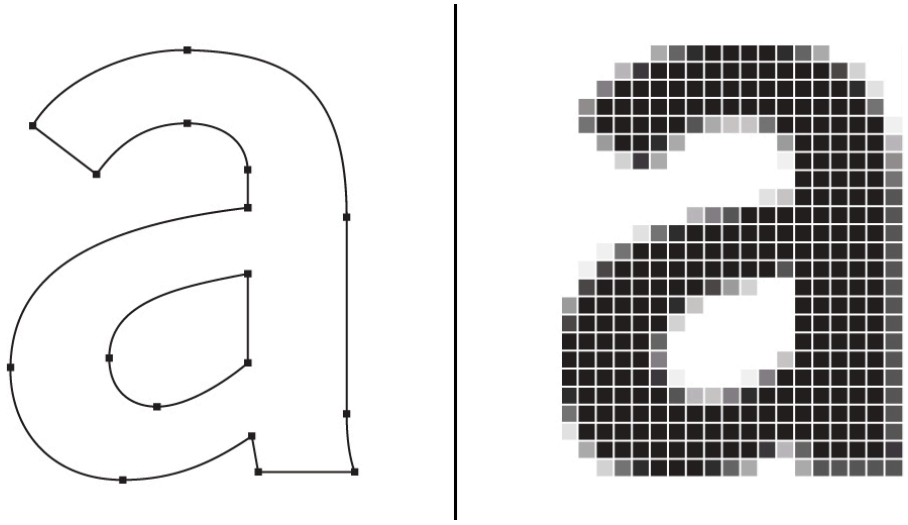


Figure 1: Vector vs Bitmap graphics (Ratermanis, 2017)

Traditional methods of rendering 2D graphics has always relied on bitmap-based texture mapping (Ray et al., 2005). While this approach is ubiquitous, it suffers a major drawback — the *pixelation* effect when being scaled beyond the original resolution (Nehab & Hoppe, 2008). Furthermore, creating animations using bitmap-based textures can be extremely limited and complex because of the rigid grid-like data structure used to store the data. Animating raster graphics are commonly done through the use of fragment shaders which directly manipulates individual pixels, or relying on flipbooks (image sequences) to simulate motion (Iché, 2016; Jeremias & Quilez, 2014).

In contrast, vector graphics offer precision and resolution independence, enabling infinite scalability without visual degradation (shown in Figure 1). A vector illustration is composed of multiple *paths* that define *shapes* to be painted in a given order (Ganacim et al., 2014). Each of these individual paths can be traced, altered, or even morphed into a completely different shape which allows for a huge variety of animation techniques in real-time (Carlier et al., 2020; Dalstein et al., 2015; Mateja et al., 2023). Vector graphics animations can also react to dynamic runtime changes as the very definition of their shapes are extremely flexible. This property allows vector animations to be generated procedurally instead of relying on pre-recorded data (Hoyer, n.d.).

However, despite these advantages, support for authoring complex vector-based interactions and animations within standard user interface creation workflows remains limited. Most game engines in the market including Unity, Unreal Engine, and Godot uses a WYSIWYG (What You See Is What You Get) editor for creating user interfaces (UI) and user experiences (UX)

(Johnson, 2024; Krogh-Jacobsen, 2023; Santos, 2023). WYSIWYG editors are visual centric tools that let users work directly within the presentation form of the content (Mädje, 2022). In WYSIWYG editors, users normally layout their contents using a drag and drop method and then style them using a style-sheet (HeySERP-Team, 2023). To bind interactions or animations towards a content, users would need to label it with a unique tag and query them through code.

Complex content and logic would not be possible through a typical WYSIWYG editor. For instance, it is virtually impossible to author a custom polygon shape in the editor with custom math based animation which depends on a time value. This can only be achieved through code, and is often limited by the API layer provided by the WYSIWYG editor. This creates a huge distinction between the game/UI logic and the visual representation that is needed to convey the messages.

While hot-reloading is applicable for the layout and styling (and simple logic to some extent) of a content. In an Unreal Engine UI tutorial titled “UMG Best Practices” by Gagnon (2019), the author concluded that logic should be kept in the C++ language (code) and Unreal Blueprints (visual scripting). It showcases that a WYSIWYG editor would not be capable of hot-reloading complex logic as these can only be achieved using code, which in most cases, requires a re-compilation. This could lead to frustration and lost of creativity due to the slow feedback loop.

In summary, raster graphics poses two major issues in interactive content creation — the scalability of the graphical content and the handling of dynamic animation state changes. Additonally, modern game engine UI editors lack the capability to support complex interaction, often offloading that responsibility to the game developers. This results in a slow feedback loop and potential miscommunication between design and development teams.

1.3. Project Aim

This project aims to establish a framework that enables the integration of interactive and dynamic vector graphics into real-time video games, examining its potential to enhance visual quality, scalability, and interactivity within modern game environments.

1.4. Research Objectives

1. To assess the impact of vector graphics on visual quality in real-time games.
2. To design a framework that integrates vector graphics into real-time game environments, focusing on dynamic interactivity.
3. To identify and address the technical challenges needed for efficient real-time rendering of dynamic vector graphics.

1.5. Research Questions

1. What is the impact of using vector graphics on the visual quality in real-time games?
2. How can vector graphics be efficiently integrated into real-time game environments to enhance dynamic interactivity?
3. What are the technical challenges associated with rendering dynamic vector graphics in real-time?

1.6. Scope

The scope of this project invovles making a game — **Lumina** that utilizes **Velyst**. The creation of the game will help ensure that Velyst is production ready by the end of this project.

Velyst	An open sourced, interactive Typst content creator using Vello and Bevy.
Lumina	A 2D top down fast-paced objective based PvPvE game.

Table 1: Scope

1.6.1. Tasks to be executed

1. Develop the open sourced Velyst crate.
    a. Develop an integrated compiler for Typst content in Bevy.
    b. Support hot-reloading of Typst content.
    c. Support interactivity between Bevy and Typst.
    d. Develop an easy-to-use framework for UI creation using Typst.
    e. Support in-game textures using vector graphics content (e.g. using vector graphics texture for a crate in the game).
    f. Write up a getting started documentation to make on-boarding easier for new developers.

2. Develop the Lumina game.
    a. Create a game design document (GDD) for the game.
    b. Integrate Velyst into the game.
    c. Develop all the required game mechanics for the game.
    d. Playtest the game and gather player feedbacks on the game.

1.6.2. Constraints

Constraint	Reason
Compatibility	This project uses multiple cutting-edge and innovative technologies. This means that some of the technologies we depend on might be experimental or have yet to stabilize. This makes it difficult to ensure cross-platform / device compatibility for the systems we built. For example, Vello requires compute shaders to render vector graphics, which means it can only run on newer versions of browsers that support WebGPU.
Limited Documentation	Because some of the technologies we use are experimental or less widely adopted, available documentation and community support may be limited. This slows down the development process as issues can be difficult to troubleshoot without established resources.
Limited Resource	The project is subject to limitations in terms of budget, personnel, and time. Allocating sufficient resources to develop, test, and refine the workflow for real-time vector graphics is crucial. Any constraints in these areas can impact the project's scope and delivery timeline.
Vector Graphics Constraint	It is crucial to recognize that while vector graphics offer numerous benefits, it is only suitable for representing precise shapes — such as fonts, logos, and icons. In contrast, complex images with intricate details, like a photograph of a cat are far better represented using bitmap formats.

Table 2: Constraints

1.6.3. What will be done in this project

1. **Typst compiler**: A custom implementation of Typst compiler will be developed to fit the purpose of real-time Typst content rendering. This compiler should be able to re-compile Typst content on demand, allowing developers to view and reflect their saved changes immediately.

2. **Vector graphics renderer**: The Velyst crate will allow users to generate dynamic vector graphics content which is then rendered through the Vello renderer.

3. **Game demo prototype**: Lumina will be developed to showcase all of the above in a compact game format. Players will experience realistic and beautiful 2D lighting, as well as interactive vector graphics UI elements.

1.6.4. What will not be done in this project

1. **Not creating an animation library**: An animation library involves preparing a huge variety of common animation effects. This takes a huge amount of time which does not fit the goal and scope of this project. To understand more about animation libraries, we strongly encourage you to look into the *Bevy MotionGfx* project.

2. **No visual editors**: It is possible to blend the Typst language with some form of visual editors, where the output would be a Typst function that takes in input parameters and produces a dynamic output content. However, our main focus for this project is to address the shortcomings of modern WYSIWYG editors. A feature as described would only be applicable after Velyst becomes a viable solution.

3. **Not a commercial game**: The goal with Lumina is to create a game that demonstrates Velyst. It is not meant to be on par with a full on commercial game.

1.6.5. Open Source License

Velyst is dual-licensed under either:

- MIT License (http://opensource.org/licenses/MIT)
- Apache License, Version 2.0 (LICENSE-APACHE or http://www.apache.org/licenses/LICENSE-2.0)

This means you can select the license you prefer! This dual-licensing approach is the de-facto standard in the Rust ecosystem and there are very good reasons to include both. Head over to the Velyst GitHub repository (https://github.com/voxell-tech/velyst) to learn more!

1.7. Potential Benefit

1.7.1. Tangible Benefit

1. **Reduced asset size and load times**: The framework enables the use of vector graphics, which often require less storage space compared to traditional raster assets. This can result in smaller game asset sizes and improved load times, which can be quantitatively measured, providing concrete evidence of improved performance.
2. **Enhanced visual adaptability and resolution independence**: Vector graphics allow for high-quality visuals at any resolution, ensuring graphics remain crisp on various screen sizes and devices. This adaptability can be measured through resolution testing, demonstrating consistent visual quality across platforms.

3. **Improved development efficiency**: By streamlining the workflow, the framework reduces the time and resources needed for asset creation and rendering. Features like programmable content and hot-reloading can lead to faster development cycles and lower costs, measurable through development time tracking and cost analysis.

1.7.2. Intangible Benefit

1. **Increased creative freedom for developers**: The framework encourages developers to experiment with dynamic and interactive visuals, promoting creativity and innovation in game design. Although challenging to quantify, this freedom can result in more unique and engaging gameplay experiences, contributing to industry innovation.
2. **Enhanced player immersion and engagement**: By enabling smooth, responsive graphics, the framework enhances the overall visual appeal and interactive feel of games, potentially increasing player engagement and immersion. This contributes to player satisfaction, though it is difficult to measure directly.
3. **Long-term industry shift toward versatile graphics approaches**: The adoption of this framework may drive a gradual shift toward broader vector graphics usage in game development, promoting a more adaptable approach to graphics. Although this industry impact may not be immediately measurable, it contributes to evolving standards in visual and interactive design within games.

1.7.3. Target User

Velyst will particularly be targeted towards UI/UX developers, motion graphics creators, and vector graphics enthusiasts. Because it is built on top of Rust and Bevy, the general users will come from the Rust and Bevy community.

As for Lumina, the target audience are gamers who love fast-paced multiplayer games like *Apex Legends* and *Astro Duel 2*. It will particularly appeal to gamers who love the mix of competitive PvP and PvE like *Destiny 2*'s Gambit game mode and *World War Z*.

1.8. Overview of the IR

This investigation report explores the establishment of a framework for integrating interactive and dynamic vector graphics in real-time video games. The project begins by providing a comprehensive introduction, outlining the context and background of the problem, where vector graphics, despite their scalability and precision, remain underutilized in modern game environments. This is followed by an articulation of the research aim, objectives, and scope, along with the potential benefits of the study. The introductory section concludes with an

overview of the research design and project plan, setting the stage for a systematic investigation into the viability and advantages of vector graphics in real-time gaming contexts.

The literature review, presented in Chapter 2, delves into domain-specific research and technical explorations relevant to vector graphics and their application in game development. This section examines similar systems and related technologies, allowing for a comparison of existing approaches and identifying gaps within the current state of the art. Additionally, technical research focuses on the computational and algorithmic challenges posed by real-time vector graphics rendering, which informs the subsequent methodology and framework design. This chapter concludes with a summary that synthesizes the key insights from the literature, shaping the foundation for the methodological approach.

In Chapter 3, the methodology outlines the structured approach taken to develop and validate the proposed framework. Beginning with an introduction to the selected software development methodology (SDM) and its phases, this section describes the data gathering methods used to collect feedback from game developers and interactive application creators. This includes the design and execution of questionnaires and interviews, aimed at gathering insights on the practical demands and potential impacts of vector graphics within the industry. The methodology further explains the criteria for analysis, establishing a rigorous basis for interpreting the collected data. The next section presents the findings from the questionnaire and interview analyses, examining trends, challenges, and developer preferences for integrating vector graphics in real-time environments. These results inform the conclusions drawn in the final section, where the potential of vector graphics in game development is assessed alongside the strengths and limitations of the proposed framework.

Finally, the report closes in Chapter 4 with reflections on the study's contributions, limitations, and possible directions for future research, suggesting a pathway toward broader adoption and innovation in vector-based graphics in gaming.

1.9. Project Plan

Task Name	Duration (Day)	Start Date	End Date
Project Proposal Form	14	9th of September 2024	22nd of September 2024
Chapter 1: Introduction			
Introduction	2	23rd of September 2024	24th of September 2024
Problem Background	2	25th of September 2024	26th of September 2024
Project Aim	1	27th of September 2024	27th of September 2024
Objectives	1	28th of September 2024	28th of September 2024
Scope	1	29th of September 2024	29th of September 2024
Potential Benefits	1	30th of September 2024	30th of September 2024
Overview of IR	1	1st of October 2024	1st of October 2024
Project Plan	1	2nd of October 2024	2nd of October 2024
Chapter 2: Literature Review			
Introduction to LR	1	3rd of October 2024	3rd of October 2024
Domain Research	8	4th of October 2024	11th of October 2024
Similar System	3	12th of October 2024	14th of October 2024
Technical Research	5	15th of October 2024	19th of October 2024
Summary	1	20th of October 2024	20th of October 2024
Chapter 3: Methodology			
Introduction of SDM	1	21st of October 2024	21st of October 2024
Methodology Choice and Justification	2	22nd of October 2024	23rd of October 2024
Data Gathering Design	6	24th of October 2024	29th of October 2024
Questionnaire Design	3	30th of October 2024	1st of November 2024
Interview Design	4	2nd of November 2024	5th of November 2024
Questionnaire Analysis	3	6th of November 2024	8th of November 2024
Interview Analysis	3	9th of November 2024	11th of November 2024
Summary	1	12th of November 2024	12th of November 2024
Chapter 4: Conclusion			
Conclusion	2	13th of November 2024	14th of November 2024

Table 3: Project Plan

CHAPTER 2: Literature Review

2.1. Domain Research

2.1.1. Introduction

Vector graphics is a form of computer graphics where visual images are generated using mathematical formulae (Rick et al., 2024). This includes geometric shapes, such as points, lines, curves, and polygons that are defined on a Cartesian plane. The use of vector graphics in games can be tracked way back to when computer games was first developed. One of the earliest examples of video game, Tennis for Two as shown in Figure 2 uses vector graphics to render their game on a repurposed oscilloscope in 1958 (Filimowicz, 2023; Rick et al., 2024). It was not long before video games was first commercialized during the 1970s and 1980s, with the release of vector graphics rendered games like Space Wars (1977), Battlezone (1980), and Tac/ Scan (1982) (Rick et al., 2024). These games showcases the potential of vector-based visuals to achieve fluid and interactive animations.

Figure 2: Tennis for Two (Higinbotham, 1958)

Around this time, graphical processing units (GPU) were also experiencing rapid development and growth. In 1989, Silicon Graphics Inc. (SGI) created one of the earliest graphics application programming interfaces (API) OpenGL, which forms the foundation of today's computer graphics software (Crow, 2004). As GPU advanced, support for raster graphics improved significantly, leading to a decline in the use of vector-based rendering technology in games (Stanford, 2024).

Despite the rise of raster graphics, the unique benefits of vector graphics (scalability and precision) continue to offer significant potential in modern game environments. Today, vector graphics are rendered using high resolution monitors through the process of rasterization (Tian & Günther, 2022). This necessity led to the rise of algorithms specifically designed to convert mathematically defined shapes into pixels, creating a new domain of computational challenges. In addition, there is little to no tool available that effectively integrates vector graphics content into real-time, interactive game environments. The absence of such tools has hindered the widespread adoption of vector graphics in modern game development, limiting their use to methods like triangulation and sign distance field due to technical constraints (Alvin, 2020; DesLauriers, 2015).

2.1.2. Rasterization of vector graphics

Display devices (e.g. monitors and televisions) are the primary medium for displaying dynamic graphical content in modern computing. These devices operate by illuminating small units called pixels, arranged in a grid, to form images (Sheldon, 2022). Each pixel can emit a specific color as shown in Figure 3, and together, millions of these pixels create the visuals we see on screen. The clarity of an image on a monitor is determined by its resolution, which defines the number of pixels displayed horizontally and vertically. Higher resolutions allow for finer details, but this also requires more computational power to manage and render the visuals effectively (*Gaming Resolution and Hardware Choices*, 2023). Over the years, three primary technologies have been used to render images on display screens: Cathode Ray Tube (CRT), Liquid Crystal Display (LCD), and Light-emitting Diode (LED) (Sheldon, 2022).

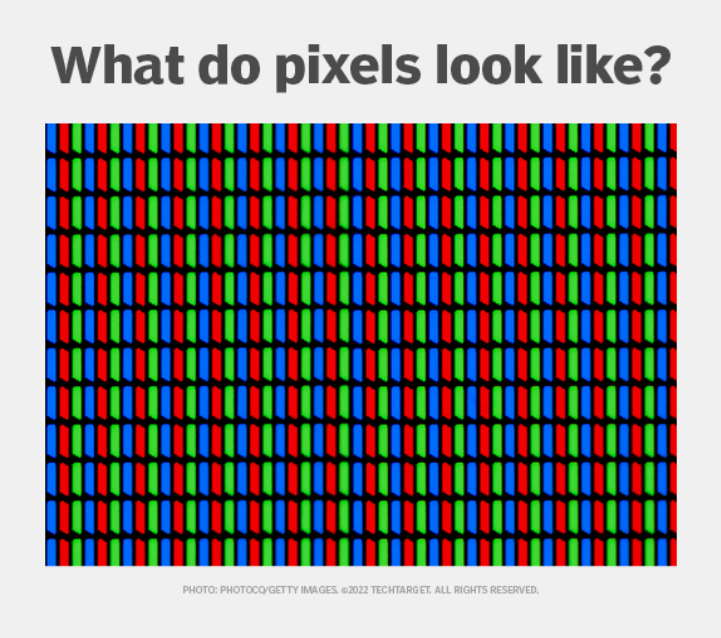


Figure 3: What do pixels look like (Sheldon, 2022)

Displaying vector graphics on these displays requires a process known as rasterization. Rasterizing vector graphics involves converting mathematical paths and shapes into pixel-based images that can be displayed on screens. Solving for this process is non-trivial, as it is often required to compute a partial differential equation (PDE) (Tian & Günther, 2022).

One method for rasterizing vector graphics is scanline rendering. Scanline rendering is the process of shooting rays from one side of the screen to the other while coloring pixels in between based on collision checkings with paths in between. A GPU based scanline rasterization method is proposed by parallelizing over boundary fragments while bulk processing non-boundary

fragments as horizontal spans (Li et al., 2016). This method allows fully animated vector graphics to be rendered in interactive frame rates.

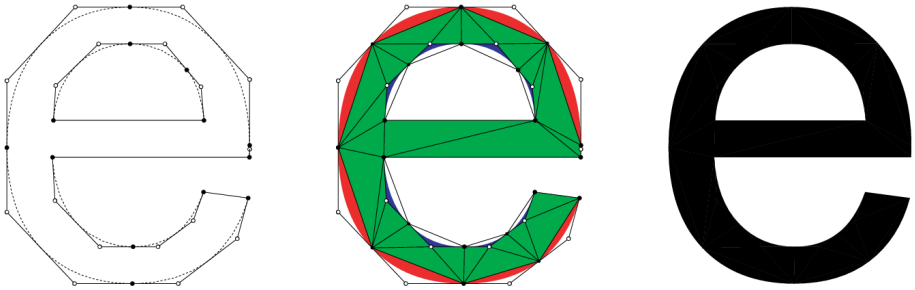


Figure 4: Tessellation with shader evaluated curves (Loop & Blinn, 2005)

Apart from scanline rasterization, tessellation method can also be used to convert vector graphics into triangles and then pushed to the GPU for hardware accelerated rasterization. Loop and Blinn (2005) further improved this method by removing the need of approximating curve segments into lines. Instead, each curve segments is evaluated in a fragment shader which can be calculated on the GPU, as shown in Figure 4. This allows for extreme zoom levels without sacrificing qualities.

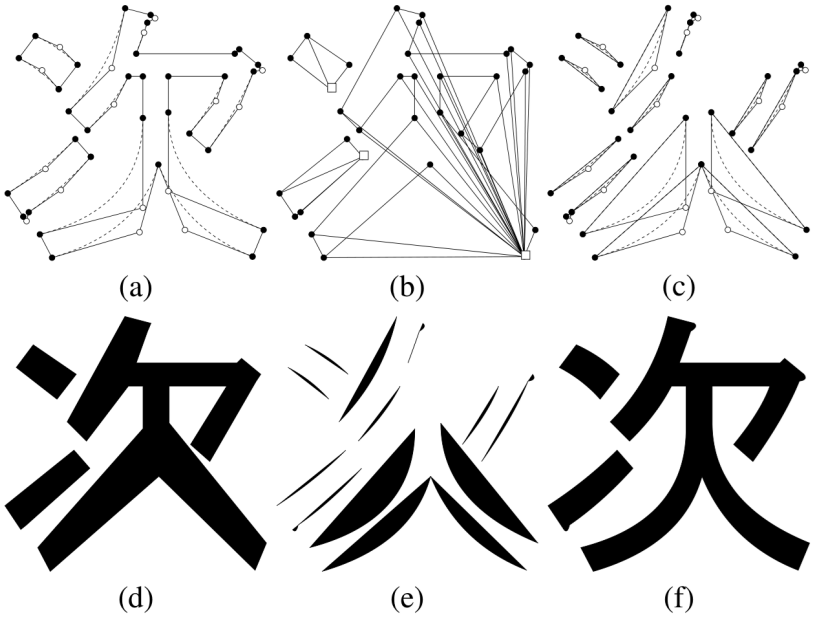


Figure 5: Triangle fans (Kokojima et al., 2006)

Re-tessellation of vector graphics can be computationally expensive, especially when it is inherently a serial algorithm that often needs to be solved on the Central Processing Unit (CPU). Kokojima et al. (2006) combines the work of Loop and Blinn (2005) with the usage of GPU's stencil buffer by using triangle fans to skip the tessellation process as shown in Figure 5. This approach, however, does not extend to cubic Bézier segments as they might not be convex.

Rueda et al. (2008) addressed this issue by implementing a fragment shader that evaluates the implicit equation of the Bézier curve to discard the pixels that fall outside it. The two-step "Stencil then Cover" (StC) method builds upon all of these work and unified path rendering with OpenGL's shading pipeline — `NV_path_rendering` (Kilgard & Bolz, 2012). This library was further improved upon by adding support for transparency groups, patterns, gradients, more color spaces, etc. (Batra et al., 2015). It was eventually integrated into Adobe Illustrator.

2.1.3. Immediate mode and retained mode

Graphical user interfaces (GUI) are integral to modern software, providing users with intuitive ways to interact with applications across a range of devices (Juviler, 2024). GUIs represent software functionality using graphical elements like buttons, sliders, icons, and menus instead of relying on text-based commands as shown in Figure 6. From smartphones and computers to ATMs and gaming consoles, GUIs have become the standard for interacting with modern technology. They are designed to provide a seamless, visually engaging experience that bridges the gap between users and the complex operations running behind the scenes.

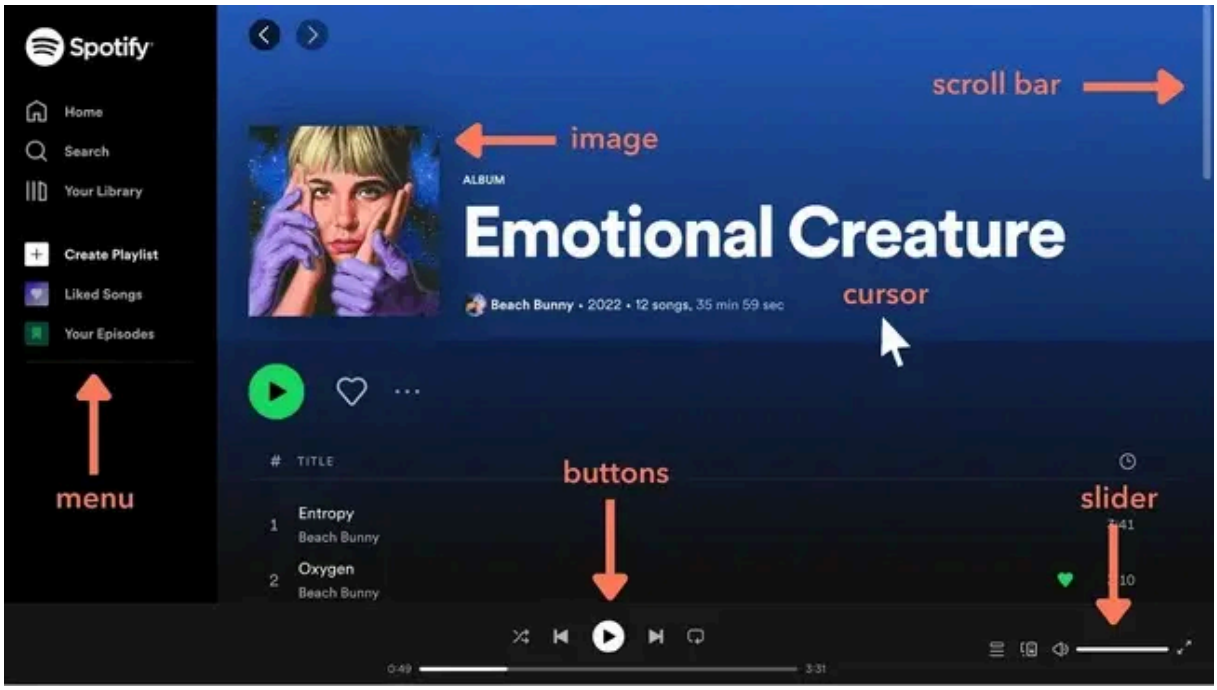


Figure 6: Modern GUIs (Juviler, 2024)

Beneath all graphical interfaces lies the underlying code that structures and renders the visual elements. A GUI has a lot of states to manage as it can be highly dynamic and might contain complex user interactions (Levien, 2018). The two approaches towards creating user interface frameworks are immediate-mode graphical user interface (IMGUI) and retained-mode graphical user interface (RMGUI). Some open sourced IMGUI frameworks includes Dear ImGui and Egui, while open sourced RMGUI frameworks includes Xilem and Qt. Although powerful, these UI frameworks strongly relies on hard-coded programming. In retained-mode, the application attempts to retain its previous state as much as possible and only perform changes when necessary. In contrast, the immediate-mode application reconstructs its frame in every update with no states stored in between frames (Satran & Radich, 2019). This makes retained-mode useful for applications that do not require high dynamic changes or devices that require low

power consumption and vice versa (Glazkov, 2021). In the article by Satran and Radich (2019), the authors also stated that while retained-mode APIs are generally easier to use, they often come with higher memory demands and offer less flexibility than their immediate-mode counterparts.

```
ui.heading("My egui Application");
ui.horizontal(|ui| {
  ui.label("Your name: ");
  ui.text_edit_singleline(&mut name);
});
ui.add(egui::Slider::new(&mut age, 0..=120).text("age"));
if ui.button("Increment").clicked() {
  age += 1;
}
ui.label(format!("Hello '{name}', age {age}"));
ui.image(egui::include_image!("ferris.png"));
```

Figure 7: Egui code example (imperative)

```
struct AppData {
  count: u32,
}

fn count_button(count: u32) -> impl View<u32, (), Element = impl Widget> {
  Button::new(format!("count: {}", count), |data| *data += 1)
}

fn app_logic(data: &mut AppData) -> impl View<AppData, (), Element = impl Widget> {
  Adapt::new(|data: &mut AppData, thunk| thunk.call(&mut data.count),
    count_button(data.count))
}
```

Figure 8: Xilem code example (declarative)

In practice, IMGUI is imperative and RMGUI is declarative as shown in Figure 7 and Figure 8. In imperative programming, developers define each step needed for a program to reach the desired state, specifying precisely how to display UI components. Declarative programming, on the other hand, allows developers to describe what should be displayed without detailing how to achieve it. Declarative frameworks often rely on underlying imperative code to translate high level commands into step-by-step instructions for execution (Ricardo, 2020). While imperative programming allows programmers to have complete control over system resources, it would eventually lead to higher complexity as projects scale, increasing the risk of bugs. In contrast, declarative programming minimizes state mutability by favoring more sophisticated constructs like pipelines and work graphs, which leads to better scalability to larger projects. However,

it is important to understand the memory overhead and higher learning curve that comes with declarative programming (Kashivskyy, 2024).

2.1.4. Markup languages for graphical interface content creation

Enter the web technologies. Modern browsers typically render UI elements using markup languages like Hyper Text Markup Language (HTML) and Scalable Vector Graphics (SVG) for structuring the content and style-sheets like Cascading Style Sheets (CSS) for styling them as shown in Figure 9 and Figure 10. The use of markup structures allows developers to fully separate their UI layout from the codebase, simplifying the identification and management of UI components. With style sheets, developers can create, share, and reuse templates, enhancing consistency and streamlining the design process throughout the application.

```html
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" href="styles.css">
</head>
<body>
  <h1>This is a heading</h1>
  <p>This is a paragraph.</p>
</body>
</html>
```

Figure 9: HTML example

```css
/* styles.css */
body {
  background-color: powderblue;
}
h1 {
  color: blue;
}
p {
  color: red;
}
```

Figure 10: CSS example

Markup languages also give rise to many WYSIWYG editors. These editors let users perform drag and drop actions to layout UI for quick prototyping as each components can now be represented using only markup syntax (no code required). An example of this framework's application within a game engine is the Unity UI Toolkit, which employs it to power its UI editor (*Comparison of UI Systems in Unity*, n.d.; Krogh-Jacobsen, 2023).

A major limitation of simple markup languages like HTML is that structural changes can only be achieved through code (Sigireddyviswesh, 2023). These features are often delegated to the programmer which can lead to unintended miscommunications. For example, if you want a form to disappear after button press, you would need to alter the HTML via code. Typst offers an alternative towards this problem by introducing programming capabilities into markup (Mädje, 2022).

```latex
\documentclass{article}
\begin{document}
First document. This is a simple example, with no
extra parameters or packages included.
\end{document}
```

Figure 11: LaTeX example

Typst is a competitor to LaTeX (Figure 11), designed to simplify the typesetting process with a modern and intuitive approach. Unlike its predecessors, Typst can directly embed logic (Figure 12). Using the previous example, developers would only need to pass in a boolean value and Typst will automatically exclude the form from being in the layout at all. In the Typst ecosystem, developers gain enhanced flexibility by sharing their work as packages. Unlike templates, Typst packages support complex scripting, offering greater adaptability and expanded functionality.

```typst
#let values = (1, 2, 3, 4)
#values.pop() \
#values.len() \

#("a, b, c"
    .split(", ")
    .join[ --- ])

#"abc".len() is the same as
#str.len("abc")
```

Figure 12: Typst example

2.1.5. Summary

The exploration of rasterization techniques for vector graphics underscores the complexity and potential of rendering mathematically defined shapes in real-time applications. Vector graphics are valued for their scalability and precision, but rendering them on modern displays requires the conversion into pixel-based formats through processes like scanline rendering and tessellation. These techniques, while effective in achieving interactive frame rates, often come with significant computational costs. Innovations such as GPU-based scanline rasterization, which parallelizes boundary fragments, and advanced methods like Stencil then Cover (StC), have enabled more efficient rendering. Additionally, the development of fragment shaders for evaluating curves at extreme zoom levels enhances the ability to handle vector graphics without sacrificing quality, illustrating the evolving landscape of vector graphics rendering in computational graphics.

In the domain of graphical interface content creation, the use of markup languages like HTML, SVG, and CSS has revolutionized the way user interfaces are structured and styled. These technologies allow for the separation of content from design, simplifying the creation, management, and customization of user interfaces. However, challenges remain in introducing dynamic interactivity and structural changes without relying on code, a limitation addressed by systems like Typst, which introduces programming capabilities directly into the markup. Typst's approach enables greater flexibility by supporting complex scripting and packages, offering a significant improvement over traditional methods. This shift towards more programmable, reusable, and adaptable UI frameworks opens new possibilities for creating responsive and user-driven interfaces in modern applications.

Altogether, advancements in both rasterization and content creation tools continue to drive the development of modern game environments. As these technologies progress, game engines can offer more sophisticated and seamless graphical experiences, enabling developers to balance aesthetic quality with technical efficiency and bringing richer visual and interactive elements into games.

2.2. Similar Systems/Works

2.2.1. Unity UI Toolkit

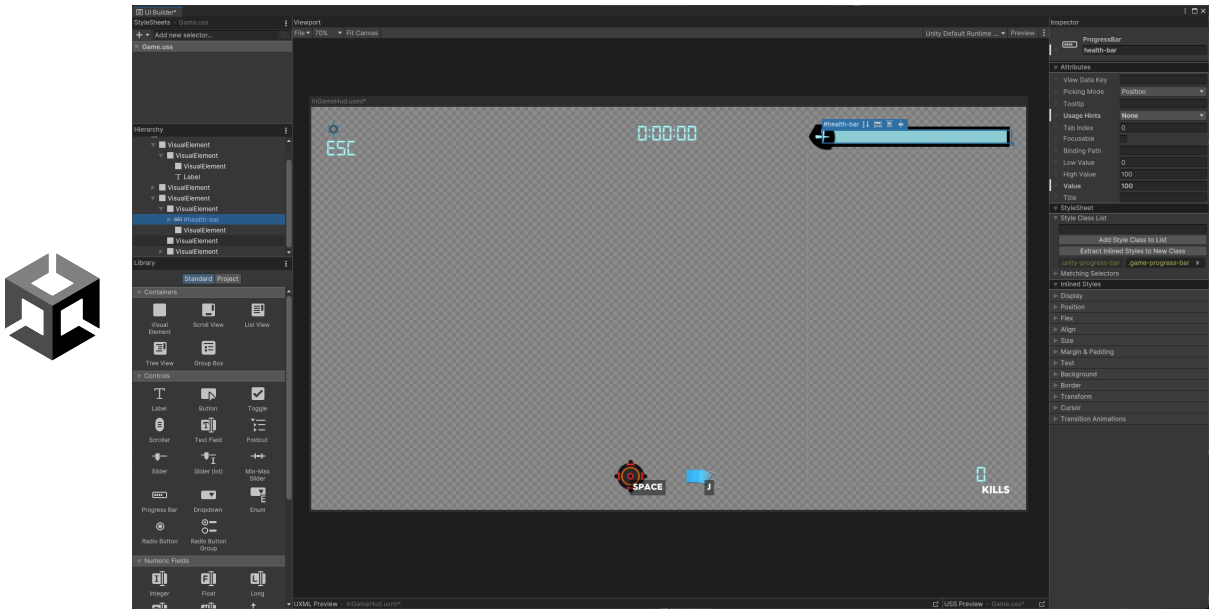


Figure 13: Unity UI Toolkit Screenshot

The Unity UI Toolkit shown in Figure 13 is a retained mode WYSIWYG editor, developed for the Unity3D Engine for editor GUI and runtime UI creation (Kok, 2021). It uses a markup language called Unity Extensible Markup Language (UXML) to define the UI structure (*Introduction to UXML*, n.d.). UXML is highly inspired by languages like HTML, Extensible Application Markup Language (XAML), and Extensible Markup Language (XML). The UXML is tailored in such a way that is efficient to work with the Unity game engine.

```
<?xml version="1.0" encoding="utf-8"?>
<ui:UXML ...>
  <Style src="<path-to-file>/styles.uss" />
  <ui:VisualElement name="root" />
</ui:UXML>
```

Figure 14: UXML example

```
/* styles.uss */
#root {
  width: 200px;
  height: 200px;
  background-color: red;
}
```

Figure 15: USS example

The Unity3D team also developed a modified CSS language called Unity Style Sheet (USS) and Theme Style Sheet (TSS) to style UIs that are structured via UXML (*Add Styles to UXML*, n.d.), (*Theme Style Sheet (TSS)*, n.d.). TSS is just a slight variant of USS in that it is solely responsible for the overall theme that the application falls back to when there is no USS or inline styling. An example of UXML and USS is shown in Figure 14 and Figure 15. Users can create custom

visual elements in C# and use it in UXML. The USS can be reused across multiple UXML and can even be imported into other USS files. This prevents duplicate work and helps keep the user interface consistent across the application.

The Unity UI Toolkit also improves collaboration between artists and developers. With UI Toolkit, artists focus on UXML and USS files, handling design elements like colors and materials, while developers add behaviors and interactions exclusively through code, without modifying the design files. This separation of logic and style streamlines merging and makes style adjustments more efficient. For example, changing project-wide fonts only requires editing Panel Settings instead of individual assets. UI Builder further assists by providing a visual interface that allows artists and designers to create and edit UI without coding, enhancing teamwork and organization for larger projects.

2.2.2. Egui

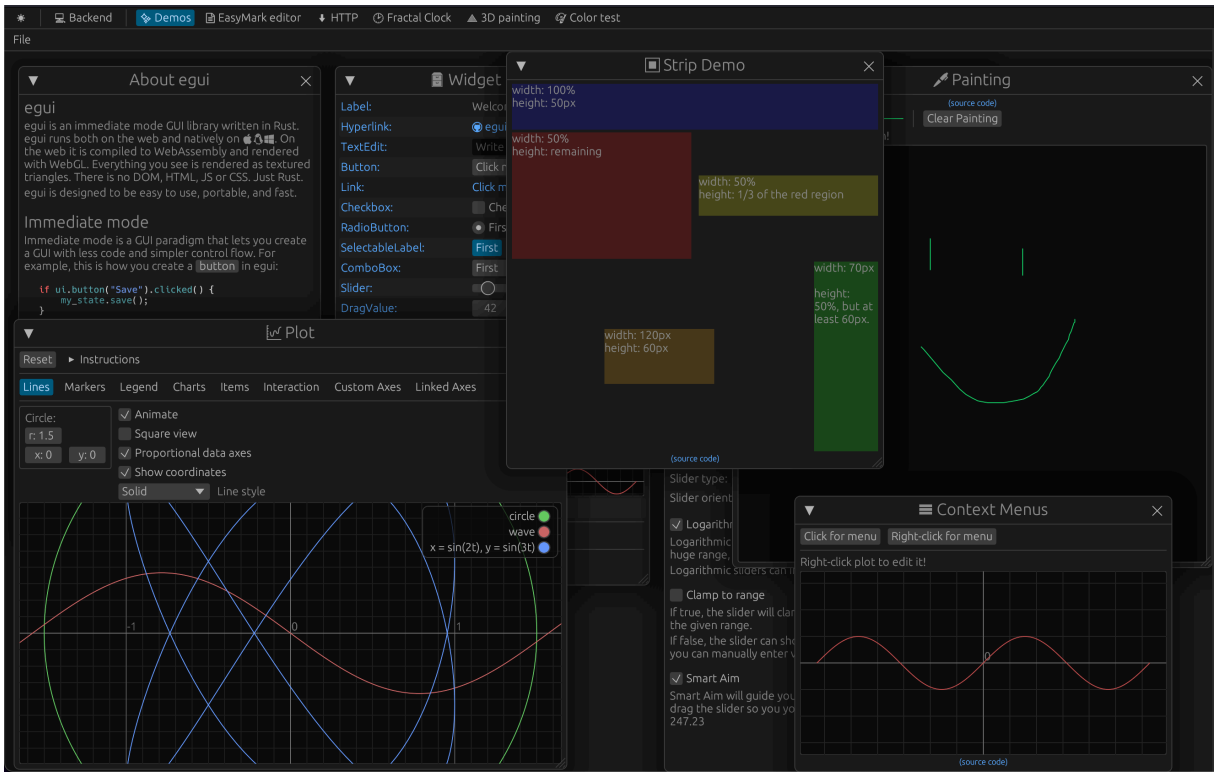


Figure 16: Egui Demo (Siwiec, 2023)

Egui is an easy-to-use immediate mode GUI in Rust that runs on both web and native. It is extremely lightweight and offers numerous graphics tools for the creation of GUI with minimal effort as shown in Figure 16. Some examples include graph plotting, text input, sliders, scroll view, painting, etc. Egui is a cross-platform UI library (Hana, 2023). Hana (2023) noted in the article that Egui is built on top of WebGPU technology. This enables it to run on both native platforms like Windows, Linux, and Mac, as well as in the browser, in the form of Web Assembly (WASM). Users can also integrate it into custom or existing game engines, custom native applications, web applications, etc. It is also highly inspired by Dear ImGui from the C++ world (Rodney, 2024).

Moreover, Egui supports extensibility by allowing users to create and integrate custom components, providing additional flexibility and customization options. When it comes to drawing capabilities, Egui offers built-in support for rendering vector graphics, including complex elements like Bézier curves, basic shapes, and geometric primitives, as noted by Hana (2023). This feature set enables developers to design visually engaging and highly interactive interfaces, suitable for both simple and complex applications, while maintaining Egui's lightweight footprint and ease of use.

2.2.3. Strengths and Weaknesses

The Unity UI Toolkit and Egui are two distinctive GUI frameworks with specific strengths and weaknesses, tailored to different development environments. The Unity UI Toolkit is a retained mode, WYSIWYG editor designed for Unity3D, whereas Egui is an immediate mode GUI built in Rust, emphasizing simplicity and flexibility across platforms.

Unity UI Toolkit

One of the primary strengths of the Unity UI Toolkit lies in its structured, markup-based approach, which leverages the UXML and USS languages. UXML's structure-based syntax is inspired by web languages like HTML and XAML, making it relatively accessible for developers familiar with web-based frameworks. This setup allows for clear separation between structure and styling, which is further enhanced by the USS and TSS. Such separation enables efficient style reuse and consistent design application across multiple UI components, avoiding redundant work and maintaining visual coherence. Moreover, *UI Toolkit*'s integration with Unity's *UI Builder* makes it a practical choice for artist-developer collaboration; it allows designers to construct interfaces visually, without needing to edit the underlying code. This, along with its retained mode nature, means developers can focus more on event-driven logic and less on rendering details, which can simplify UI maintenance and readability.

However, the Unity UI Toolkit has notable limitations. The framework's retained mode design, though powerful, can be more memory-intensive than immediate mode approaches, impacting performance in highly dynamic or resource-constrained applications. Additionally, since the UI Toolkit is specialized for Unity, it lacks portability outside the Unity ecosystem, restricting its utility for developers working across multiple engines or platforms. Although it offers an accessible toolset for users within Unity, it may require more complex workflows to integrate into non-Unity environments. Customization options are available through C# for creating visual elements, yet the reliance on Unity-specific markup and styles can create a learning curve for developers unfamiliar with the Unity ecosystem.

Egui

Egui, in contrast, is an immediate mode GUI that stands out for its simplicity, light memory footprint, and versatility. Written in Rust, Egui is designed to be cross-platform, operating seamlessly on both native environments (Windows, Linux, and macOS) and the web, thanks to its WebAssembly (WASM) compatibility. The immediacy of Egui's approach allows it to be incredibly lightweight; every frame is actively rendered, which can be ideal for

applications requiring high responsiveness or frequent UI updates. Its Rust-based design further enhances portability, enabling easy integration with various custom engines or standalone applications, making it suitable for projects not tied to a specific ecosystem. Additionally, Egui supports vector graphic rendering, including shapes and Bézier curves, which, combined with customizable components, provides significant flexibility for developers creating interactive and visually dynamic UI.

However, Egui's immediate mode nature can also present some challenges. Immediate mode UIs often involve redrawing the entire UI each frame, which can become computationally intensive in complex interfaces, potentially leading to higher CPU usage. While Egui is excellent for lightweight applications, it may not scale as efficiently for large-scale, interactive applications with many UI elements. Furthermore, Egui lacks a visual editor, which can be a drawback for teams with designers who prefer WYSIWYG tools. This absence means that UI creation and design adjustments must be handled directly in code, potentially complicating collaboration between designers and developers.

2.2.4. Summary

In summary, both frameworks offer valuable advantages within their respective domains. Unity UI Toolkit excels within the Unity ecosystem, providing structured, easily manageable UI elements and a design-friendly interface that streamlines collaboration. However, it is less portable and may consume more memory. Meanwhile, Egui's lightweight, immediate mode design and cross-platform flexibility make it highly suitable for Rust applications and custom engines, albeit with potential scalability and collaboration limitations for larger projects. Both frameworks cater to different development needs, and choosing between them depends on the platform, performance requirements, and team workflow preferences.

2.3. Technical Research

2.3.1. Rust - Programming Language

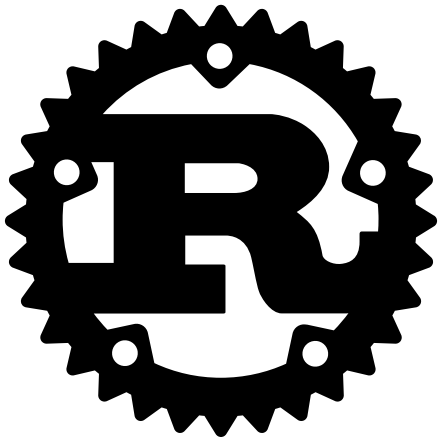


Figure 17: Rust Logo

Rust is a systems programming language designed for performance, reliability, and safety (Verdi, 2023). It is statically typed and compiles to native code, making it ideal for high-performance applications, such as game engines, operating systems, and embedded systems. Rust's primary feature is its ownership system, which ensures memory safety without the need for a garbage collector. This system eliminates common issues like null pointer dereferencing, data races, and memory leaks, all of which are common in languages like C and C++.

Rust's memory safety, concurrency capabilities, and zero-cost abstractions make it a strong choice for developing high-performance game engines. Additionally, Rust's ecosystem provides robust libraries and frameworks like Bevy for game development and Wgpu for graphics programming. The language's modern tooling, including the cargo package manager and rustfmt for code formatting, helps developers maintain efficient and clean codebases.

2.3.2. Bevy - Game Engine

Figure 18: Bevy Logo

According to the founder, Anderson (2020), Bevy is a refreshingly simple data-driven game engine and app framework built in Rust that is free and open-sourced. It is designed to be modular with its robust plug-in system, simple to use with its Entity Component System (ECS), and fast with its multi-core utilization and parallel by default design. In the article, the founder claim that Bevy's ECS is the most ergonomic ECS in existence (Anderson, 2020). As shown in Figure 19, users can create a fully working Bevy application with just a few lines of code:

```rust
use bevy::prelude::*;

fn setup(mut commands: Commands) {
  commands.spawn(Transform::default());
}

fn movement(mut q_transforms: Query<&mut Transform>, time: Res<Time>) {
  for mut transform in q_transforms {
    transform.x += 1.0 * time.delta_seconds();
  }
}

fn main() {
  App::build()
    .add_plugins(DefaultPlugins)
    .add_systems(Startup, setup)
    .add_system(Update, movement)
    .run();
}
```

Figure 19: Bevy example

Bevy is also cross-platform. It runs on Windows, MacOS, Linux, iOS, Android, and the web. Although Bevy is relatively new compared to more established game engines like Unity or Unreal Engine — having only been around for four years as of 2024 (Anderson, 2024) — it has

quickly gained attention for its modern, modular approach to game development. As a result, there has already been many volunteer plug-ins being developed for the game engine.

Some notable plug-ins that will be used of this project are:

Plug-in	**Description**
Velyst	The plug-in that we are going to develop in this project.
Bevy Vello	For renderingn vector graphics using Vello.
Lightyear	For multiplayer and networking.
Bevy Enoki	For particle effects.
Leafwing Input Manager	For managing different input devices (e.g. keyboards, mouse, controllers, etc.)
Blenvy	For level design.
Avian2d	For rigid-body physics and collision detections.

Table 4: Notable Bevy plug-ins in this project

And many more!

2.3.3. Vello - Compute-centric Vector Graphics Renderer

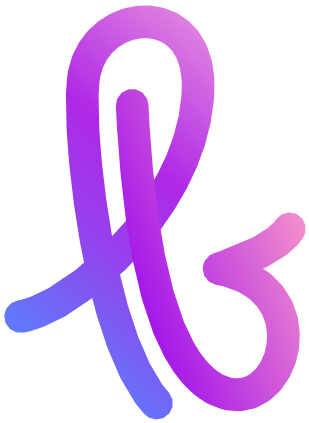


Figure 20: Linebender Logo

Vello is a compute-centric vector graphics renderer developed by the Linebender organization. The Linebender organization is a group of people who share an interest in 2D graphics and UI design (*Linebender*, 2024). Some other projects that they developed include Xilem, Masonry, Vello, Kurbo, Peniko, etc (*Linebender*, 2024).

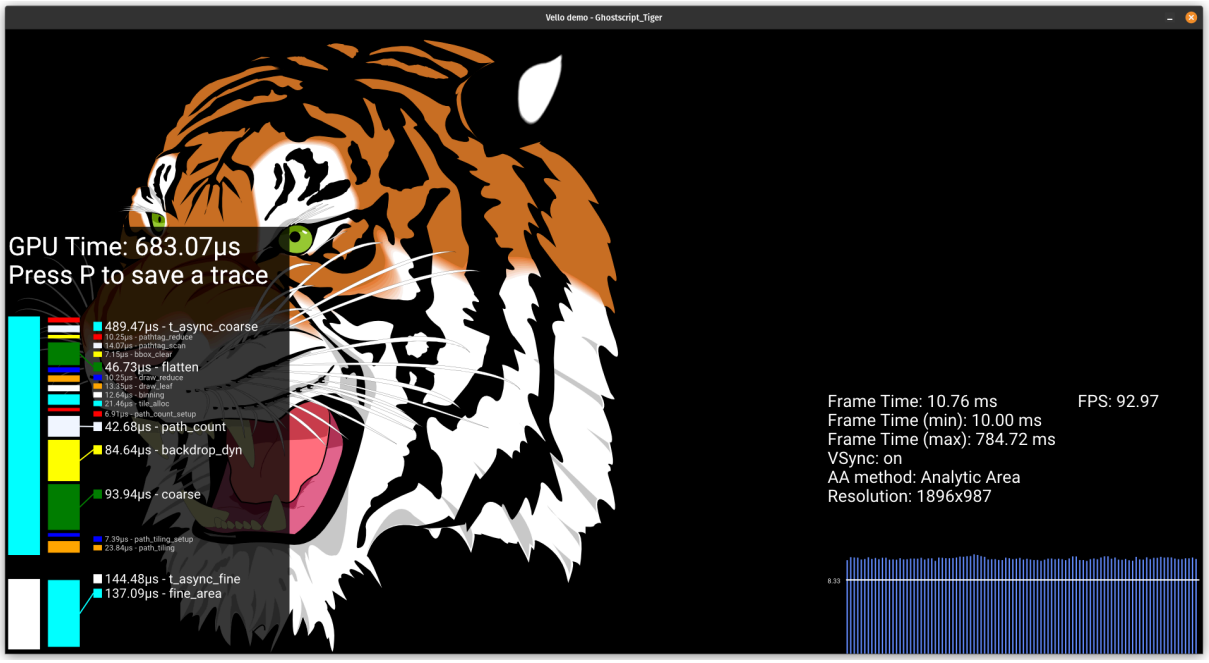


Figure 21: Vello demo

Unlike traditional vector graphics renderer, Vello strives to off-load all of the rendering steps to the GPU via compute shaders. At the moment of writing this article, Levien (2020) stated that there are plans for Vello to support retained mode in the future, but for now, it is mostly an immediate mode API. This is actually only possible because of the performance gain that Vello obtain via parallel computing on the GPU.

Figure 22: Wgpu logo

Part of what makes Vello so appealing is its cross-platform capability. Unlike many other research projects that uses CUDA as their GPU compute platform, Vello achieves portable GPU compute by utilizing the Wgpu library (Levien, 2020; *Roadmap for 2023*, n.d.). Wgpu runs natively on Vulkan, Metal, DirectX 12, and OpenGL ES; and browsers via WebAssembly (WASM) on WebGPU and WebGL2 (*Wgpu*, n.d.).

2.3.4. Typst - Typesetting

Figure 23: Typst logo

Typst has been mentioned multiple times in this paper. It is used as the language for creating graphical content in Velyst. Typst is a programmable markup language for sophisticated typesetting (Mädje, 2022). It supports programming concepts like conditionals, while and for loops, closures, functions, and many more. These concepts are directly embedded into the Typst language without any third party scripting language (Mädje, 2022).

By integrating Typst within a Rust environment through Velyst, this project combines Typst's layouting and scripting capabilities with Bevy's real-time rendering. This enables games to leverage Typst's typesetting strengths for on-the-fly generation of UI elements, text, and other graphical assets. The result is an efficient workflow where content in Typst can be manipulated and rendered dynamically within the game engine, enhancing both the visual and interactive elements of the application. This approach bridges the gap between content creation and real-time execution, offering a powerful tool for developers who seek sophisticated text and graphics rendering in games.

CHAPTER 3: Methodology

3.1. System Development Methodology

3.1.1. Introduction

Selecting a suitable development methodology is crucial for ensuring an efficient and adaptable workflow in any project. Existing methodologies each come with unique strengths and limitations, often tailored to specific types of projects. Among the most popular methodologies are Waterfall and Agile.

Waterfall

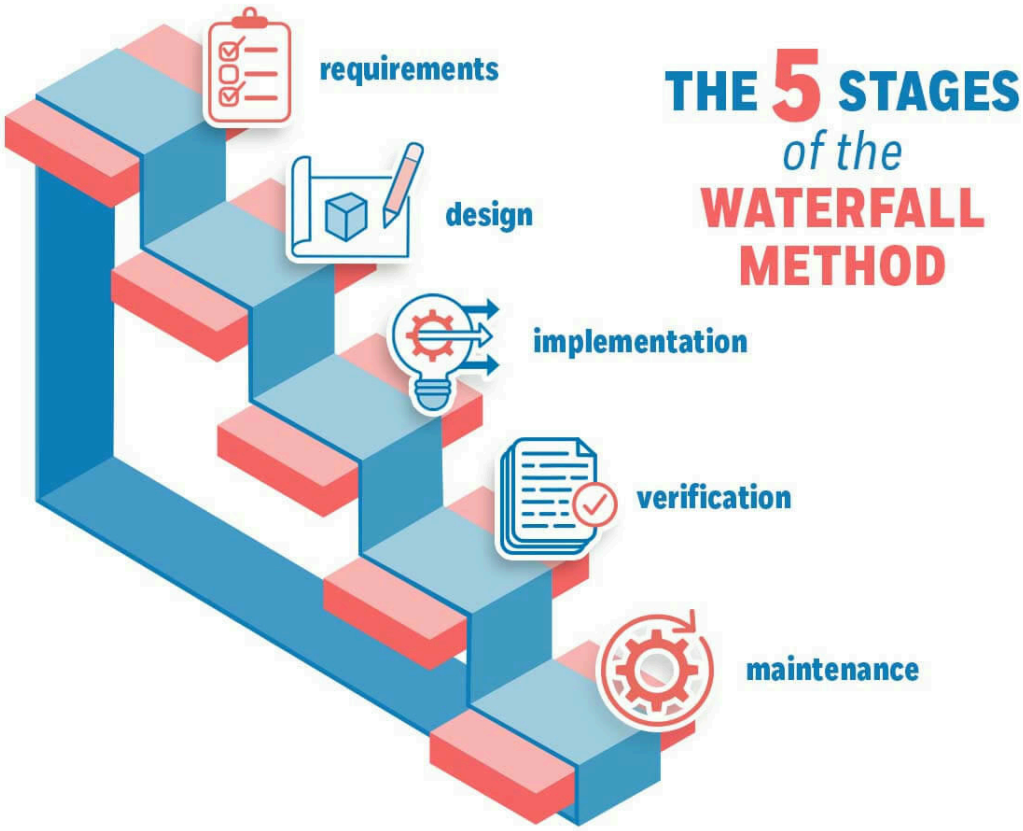


Figure 24: Waterfall methodology (Jayathilaka, 2020)

Waterfall is a linear and sequential model ideal for projects with well-defined requirements and minimal expected changes. Its strength lies in its clear structure and phase-based approach, making it predictable. However, its rigidity can be a disadvantage in projects where requirements may evolve over time, as it lacks flexibility to accommodate ongoing changes.

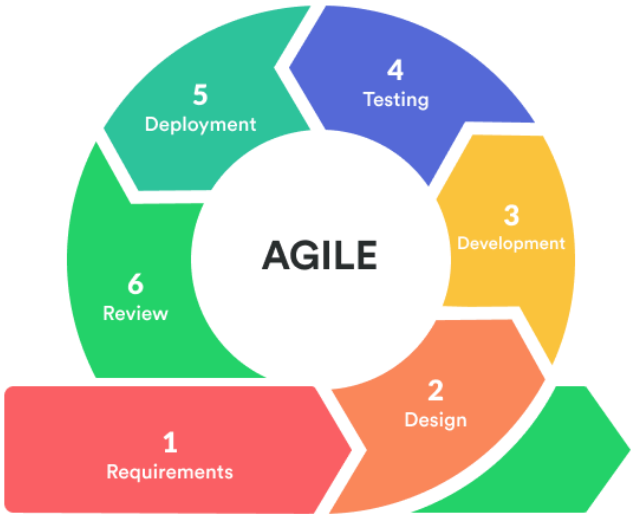


Figure 25: Agile methodology (Waseem & Hart, 2023)

Agile

Agile, on the other hand, offers a highly flexible and iterative approach that allows for continuous feedback and adjustments throughout the development process. This model excels in projects where requirements are expected to change, as it accommodates user input and evolving project goals. However, Agile can sometimes struggle in projects requiring a fixed scope and budget, as its adaptability can lead to scope creep.

Below is a simple comparison of these methodologies:

Methodology	Advantages	Disadvantages
Waterfall	• Clear structure and phase-based, making project planning predictable. • Works well for projects with well-defined requirements and goals. • Clear documentation and deliverables at each stage.	• Inflexible to changing requirements, making it hard to adapt once development starts. • Lack of early testing and feedback can lead to large-scale reworks if requirements change. • Difficult to address issues mid-project as each phase is sequential.
Agile	• Highly adaptable to change, allowing for user feedback at every stage.	• Potential for scope creep, as continuous adaptation can lead to expanding project scope.

Methodology	Advantages	Disadvantages
	• Reduces risk through continuous testing and regular check-ins. • Ideal for fast-paced environments where requirements evolve frequently. • Promotes collaboration, enhancing team responsiveness and communication.	• Requires active stakeholder engagement, which may be time-consuming. • Lacks predictability, which may make budgeting and timeline estimation challenging.

3.1.2. Methodology Choice and Justification

For this project, Agile methodology is the preferred choice due to its iterative and user-centered design, which allows the team to remain adaptable in response to feedback and evolving project requirements. Given the nature of the project, where user feedback and rapid iterations are essential, Agile supports a workflow that promotes ongoing testing, evaluation, and refinement at each stage.

The workflow in Agile will follow a sprint-based cycle, broken down into the following sections:

1. **Planning**

   Each sprint begins with a planning phase where tasks are identified and assigned based on the project goals for that sprint. The team outlines the deliverables, prioritizes tasks, and ensures each team member is clear on their responsibilities. Planning ensures that each sprint has a focused objective, with clear milestones to meet.

   Implementation: Every two weeks, our team will hold a general meetup to discuss on our current progress and next steps. During the planning phase, we also actively brainstorm on new ideas, or better implementations of mechanics, art, and design.

2. **Design and Development**

   During the design and development phase, the team works on implementing features and functionalities outlined in the sprint plan. Development occurs in small, manageable iterations, with regular check-ins and adjustments to ensure alignment with the sprint's objectives. This phase enables quick adaptability and minimizes the risks of large-scale reworks.

Implementation: After planning, each of our team member will head back to working and refining the project on their individual part. If a collaboration between two or more people is needed, they will be able to continue their discussion and work together either on Discord or physically.

3. **Testing and Feedback**

   At the end of each sprint, the team conducts testing to ensure that newly developed features are functional and aligned with project requirements. Feedback is gathered from relevant stakeholders, and any issues or suggested changes are documented. This feedback loop is critical in Agile, as it ensures that adjustments can be made continuously.

   Implementation: Approximately every month, our team will roll out a test build to collect feedbacks from a selection portion of participants. These participants will likely be our friends, family, and initial fans, audience, or followers.

4. **Review and Retrospective**

   After testing, the team performs a sprint review, evaluating both the successes and challenges encountered. The retrospective phase allows the team to reflect on the process, identify areas for improvement, and incorporate lessons learned into the next sprint's planning phase. This continuous improvement approach aligns with Agile's focus on flexibility and adaptability.

   Implementation: After the testing phase is over, we will review and evaluate all the constructive and valuable feedbacks and organize them from critical to subtle. Then, we will open up a discussion on these during the planning phase.

By following Agile, the project benefits from a dynamic and responsive workflow that can easily adapt to any changing needs, helping to ensure timely delivery and a user-centered approach. The flexibility offered by Agile makes it the ideal choice, enabling iterative progress and a focus on evolving project goals and user expectations.

3.2. Data Gathering Design

In the research design, data gathering will be conducted using two complementary methods: a questionnaire and an interview. This dual-method approach aims to balance broad, generalized data from a larger group with in-depth insights from a select group of professionals.

The questionnaire will target a broad audience who are currently in the game, UI/UX, and vector graphics field, serving as the primary method to collect quantitative data on user opinions, experiences, and preferences related to the research topic. By reaching a larger pool of respondents, the questionnaire enables the gathering of statistically relevant insights and general trends across a wide demographic. Structured in a way that includes multiple-choice questions, Likert scales, and a few open-ended items, the questionnaire will facilitate an efficient analysis while also allowing for some qualitative feedback (DEWI, 2021). This design helps to capture both objective data and individual nuances within responses, which can be further analyzed to identify general patterns and correlations. Additionally, the anonymity provided in the questionnaire encourages honest and diverse opinions from a range of backgrounds, helping to avoid biases that might be present in more direct methods of data collection.

Complementing the questionnaire, the interview component will provide a more nuanced, qualitative perspective by engaging directly with subject-matter experts who possess specialized knowledge or expertise in the field of game, UI/UX, and vector graphics. The interviews are designed to gather deeper insights that would not typically be captured in a standardized questionnaire. Conducted in a semi-structured format, the interview will allow for flexibility, enabling the researcher to probe further into responses based on the interviewee's expertise. This format encourages participants to share their informed opinions, elaborate on complex ideas, and provide practical examples, thus enriching the data with professional and real-world insights (DEWI, 2021). In addition, these expert insights will offer a valuable context for interpreting the trends identified in the questionnaire data, ensuring that the analysis incorporates both user-focused feedback and professional validation.

Together, these two methods will provide a balanced and holistic dataset, with the questionnaire delivering broad quantitative data and the interviews offering rich qualitative insights. This mixed-method approach not only enhances the credibility and depth of the research findings but also supports a comprehensive understanding of the topic from both the end-user and expert perspectives. Ultimately, combining these data sources will enable the research to deliver well-rounded conclusions and recommendations that are both broadly applicable and grounded in professional expertise.

3.2.1. Questionnaire Design

Section 1: Introduction and Basic Information

This section gathers general demographics information from the respondents such as name, age, and gender.

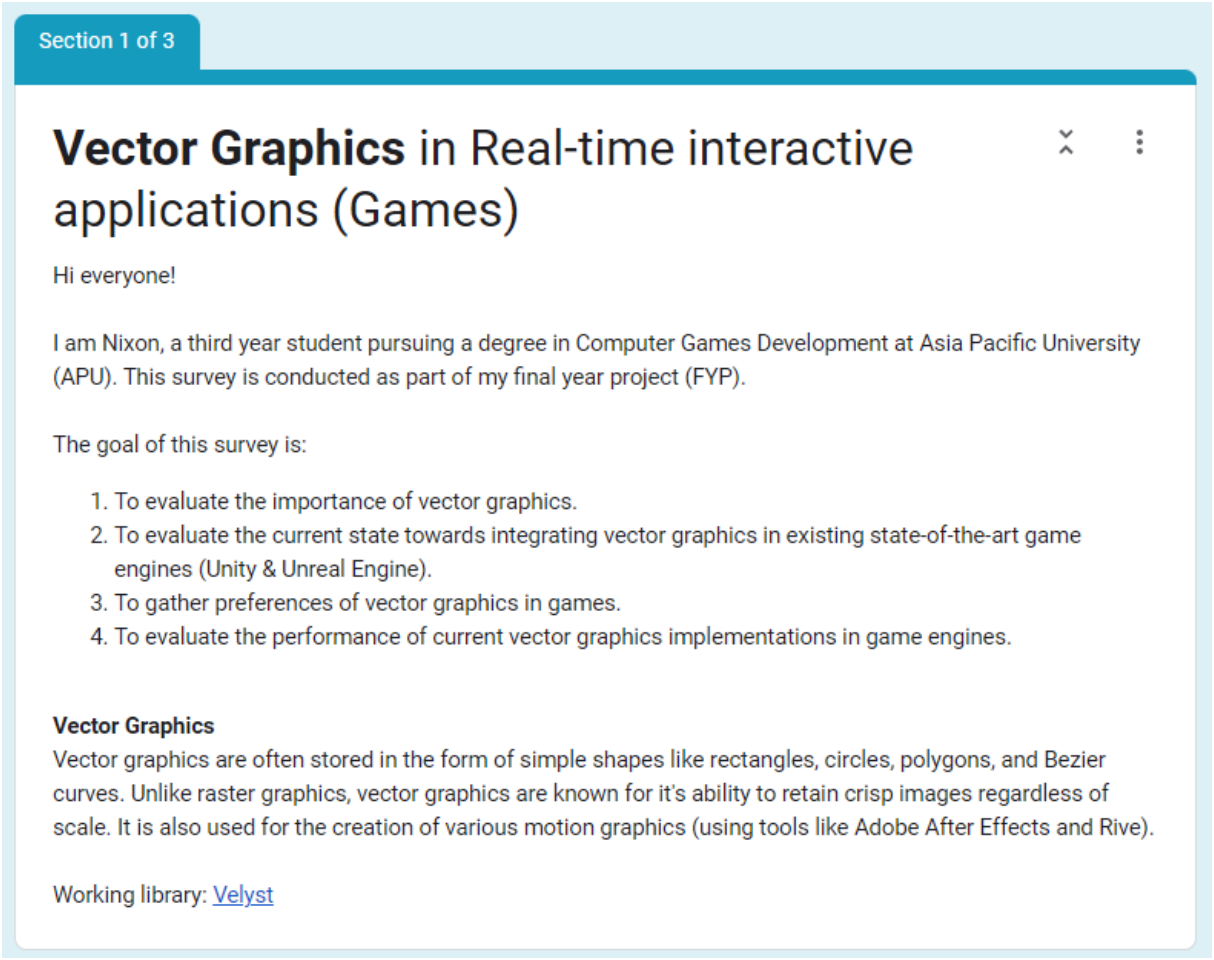


Figure 26: Screenshot of questionnaire description

No.	Question	Answer	Justification
1	Name	Short text answer	Prevent duplicate data as well contact purposes in case if any clarifications are needed.
2	Email	Short text answer	For contact purposes in case if any clarifications are needed.

No.	Question	Answer	Justification
3	Gender	Multiple Choice • Male • Female • Prefer not to say • Other:	Identify preference differences between different genders.
4	Age	Multiple Choice • < 18 • 18 - 24 • 25 - 30 • > 30	Identify preference differences across various age groups.

Table 5: Section 1 questions, answers, and justifications

Section 2: Vector graphics in games

This section provides an overview of the current state of vector graphics in game engines, focusing on feature preferences, implementations, and challenges. The questions are designed with the assumption that the respondent has a basic understanding of vector graphics. This is due to the fact that vector graphics implementations in today's game engines are still relatively limited, which means that many respondents may only be familiar with the basic concepts or have limited experience with more advanced features. Therefore, the questions aim to capture general insights rather than technical expertise, helping to gauge the broader perceptions and experiences of users with vector graphics in real-time game development.

No.	Question	Answer	Justification	Relevant Research Objective
1	How important is the use of vector graphics for creating crisp and scalable visuals in games?	Linear scale from 1-5 1 = Not important 5 = Very important	To assess the perceived significance of vector graphics for enhancing visual quality and scalability in games, which relates to their impact on player experience.	Research objective 1
2	What assets would you prefer vector graphics over raster graphics.	Checkboxes • Icons • User Interface • Particle Effects • In Game Textures • Other:	Understanding which game assets are favored for vector graphics helps identify areas where vector graphics improve visual quality and performance.	Research objective 1 and 2

No.	Question	Answer	Justification	Relevant Research Objective
3	Are you familiar with vector graphics concepts, such as scalability and the use of shapes like Bezier curves and polygons?	Multiple Choice • Yes • No	Gauges familiarity with vector graphics concepts, which informs on potential challenges in implementation.	Research Objective 3
4	How are the performance of vector graphics in said game engines?	Multiple Choice Grid Rows: • Unity • Unreal Engine Columns: • Very bad • Bad • Moderate • Well • Very well	Assess impact of vector graphics on game engine performance and identify performant implementations.	Research Objective 3
5	How challenging do you find it to implement vector graphics in said game engines?	Multiple Choice Grid Rows: • Unity • Unreal Engine Columns: • Very bad • Bad • Moderate • Well • Very well	To understand technical barriers and identifying user friendly design patterns.	Research Objective 2 and 3

Table 6: Section 2 questions, answers, and justifications

3.2.2. Interview Design

Interviews provide qualitative data that yields valuable insights into the research objectives of this project. The selected interviewees for this section are experienced developers who have been working professionally in the industry sector related to games, UI/UX, and vector graphics, bringing real-world expertise to the discussion. This targeted selection allows the interviews to be more in-depth and technical, yielding concrete evidence and examples that can enhance the understanding of practical challenges and benefits associated with vector graphics in real-time game development. These insights will help inform the design of an effective framework and guide future advancements in integrating vector graphics into game engines.

No.	Question	Justification	Relevant Research Objective
1	Could you briefly introduce yourself and your background in UI/UX or game development?	Establish context for their experience level and focus areas.	-
2	How would you describe your general workflow when designing and implementing vector graphics in applications or games?	Gather insights into workflows to identify typical processes in implementing vector graphics or similar elements.	Research Objective 2
3	What are some challenges or frustrations you encounter with current graphics tools or technologies during development?	Understand pain points that could reveal technical limitations or unmet needs in vector and motion graphics workflows.	Research Objective 3
4	In terms of visual quality and performance, what impact do you think vector graphics have on the user experience?	Gauge perceived benefits and challenges of vector graphics on the visual and performance aspects of applications.	Research Objective 1
5	Do you currently use vector graphics for interactive or dynamic content in your projects? If so, how effective have they been, and what are the main limitations?	Learn about specific use cases and barriers in vector graphics implementation within real-time environments.	Research Objective 2 and 3

No.	Question	Justification	Relevant Research Objective
6	What tools or techniques do you typically use for integrating motion graphics, such as animations from platforms like Lottie, Rive, or After Effects?	Explore how developers are bringing in external motion graphics and the role they play within interactive applications.	Research Objective 2
7	What would you like to see improved in existing tools or new tools for integrating vector or motion graphics in real-time applications or games?	Identify specific feature or functionality gaps in current technologies.	Research Objective 3
8	Do you have any feedbacks or suggestions that could help streamline the workflow of incorporating dynamic vector or motion graphics in game engines or real-time applications?	Solicit practical suggestions to improve workflows, targeting a user-centered design approach.	Research Objective 2 and 3

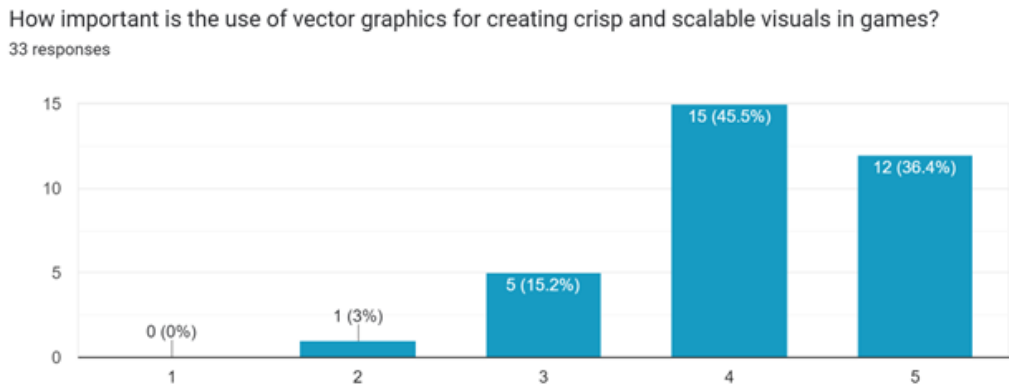
Table 7: Interview questions

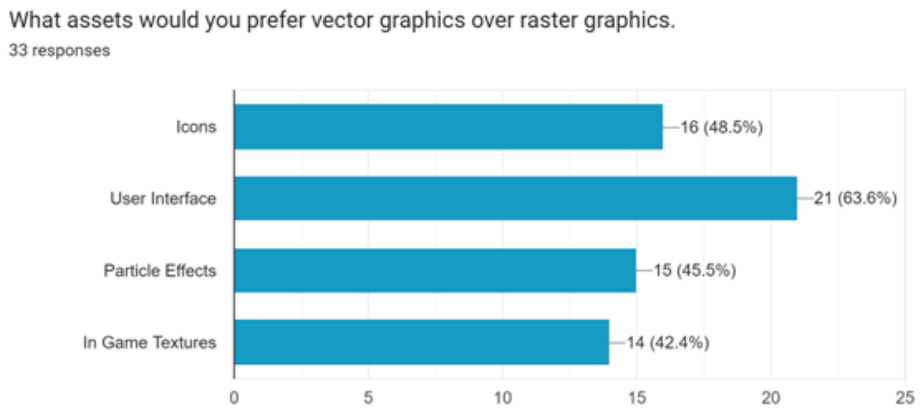
3.3. Analysis

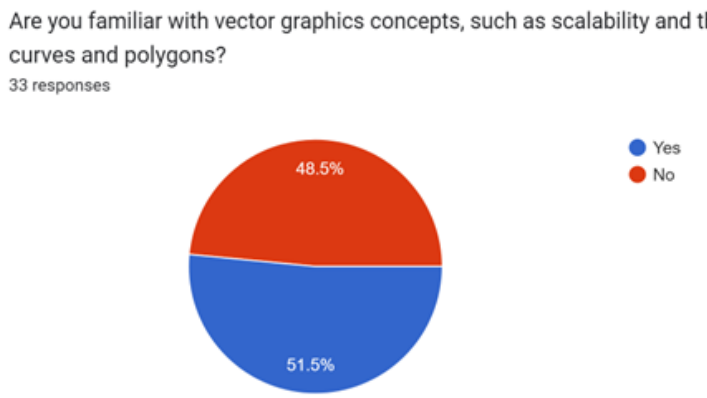
This chapter is dedicated to the analysis of the data gathered from the questionnaire and interviews. The questionnaire has acquired a total of 33 respondants while the interview has been held with seven participants.

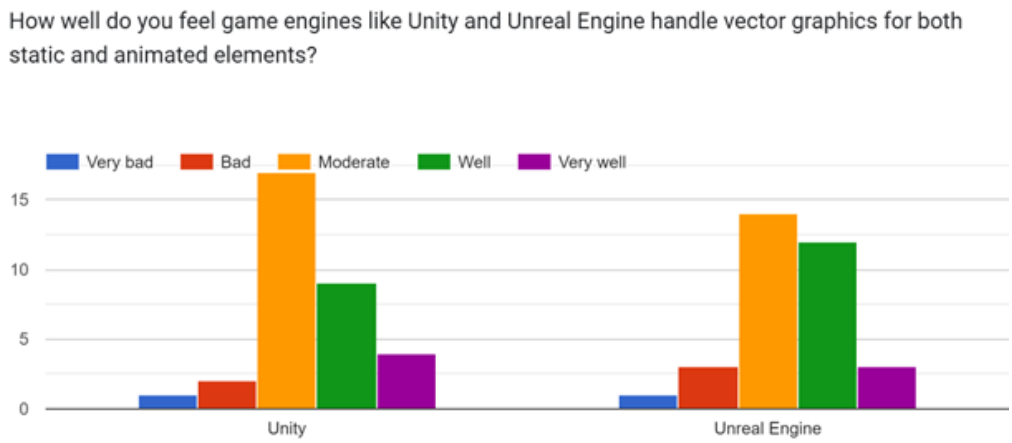
3.3.1. Questionnaire Analysis

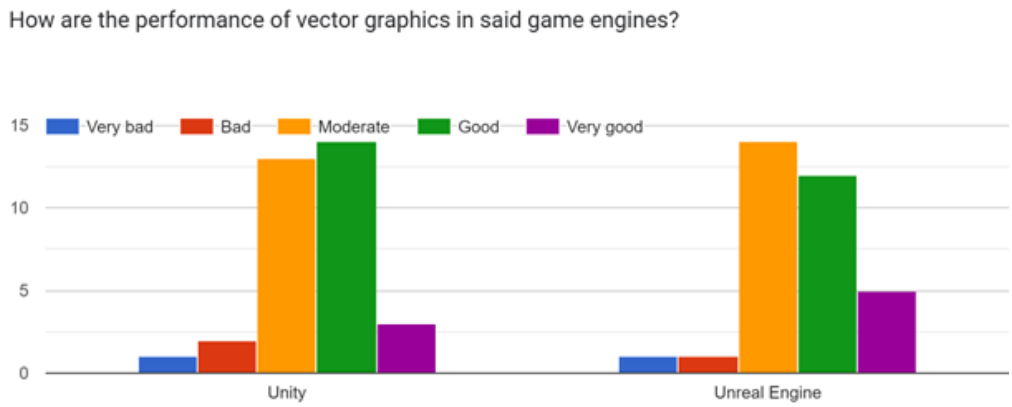
Section 2: Vector graphics in games

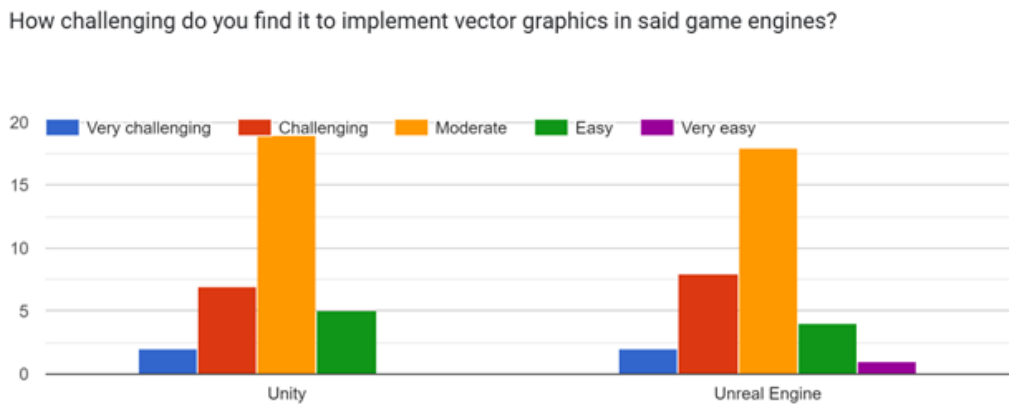
1	How important is the use of vector graphics for creating crisp and scalable visuals in games?
Data	
Analysis	The survey results show that the majority of respondents (~81%) consider vector graphics to be important or very important (ratings of 4 and 5) for creating crisp and scalable visuals in games, highlighting their perceived value in modern game design. A moderate 15.2% rated their importance as neutral (3), while only 3% gave a slightly important rating (2), and none rated them as not important (1). These findings emphasize the widespread recognition of vector graphics as a critical tool for achieving high-quality visuals across different resolutions and devices.

2	What assets would you prefer vector graphics over raster graphics.
Data	
Analysis	The survey results indicate that vector graphics are most preferred for user interfaces, with 63.6% of respondents selecting this category. Icons follow closely at 48.5%, highlighting their importance for scalable and clean visuals. Particle effects (45.5%) and in-game textures (42.4%) are also notable areas where vector graphics are valued, although to a slightly lesser extent. These preferences suggest that vector graphics are particularly suited for assets requiring scalability and clarity, such as UI elements and icons, while their adoption for dynamic and texture-heavy assets like particle effects and in-game textures is comparatively lower.

3	Are you familiar with vector graphics concepts, such as scalability and the use of shapes like Bezier curves and polygons?
Data	
Analysis	The survey reveals that familiarity with vector graphics concepts, such as scalability and the use of shapes like Bézier curves and polygons, is almost evenly split among respondents. Approximately 51.5% indicated familiarity with these concepts, while 48.5% were not familiar. This near-equal divide highlights the varying levels of knowledge about vector graphics within the respondent group and suggests a potential need for greater education or awareness about these technologies, particularly in areas where they can be effectively applied.

4	How well do you feel game engines like Unity and Unreal Engine handle vector graphics for both static and animated elements?
Data	
Analysis	The survey reveals varying perceptions of how well Unity and Unreal Engine handle vector graphics for static and animated elements. In Unity, the majority of respondents rated its handling as *Moderate*, followed by a significant portion rating it as *Well*. A smaller number of respondents rated it *Very well*, while negative ratings (*Very bad* and *Bad*) were minimal. For Unreal Engine, the ratings skew more positively, with *Well* being the most common rating, followed closely by *Moderate*. A noticeable portion rated it as *Very well*, and only a few respondents chose negative ratings. These results indicate that while both engines perform adequately, they still does not provide a consistent great user experience as most respondants only recognize it as a moderate integration.

5	How are the performance of vector graphics in said game engines?
Data	
Analysis	The survey reveals that the performance of vector graphics in Unity and Unreal Engine varies across different rating categories. In Unity, a majority of respondents rated the performance as *Moderate*, followed by *Good*. A smaller portion indicated *Very good*, while negative ratings (*Very bad* and *Bad*) were minimal. In contrast, Unreal Engine received higher positive feedback, with the majority rating the performance as *Good* and a noticeable number choosing *Very good*. However, *Moderate* ratings were slightly fewer compared to Unity, and negative ratings remained minimal for both engines. This distribution highlights that while both engines deliver adequate performance for vector graphics, Unreal Engine appears to have a stronger perception of positive performance among respondents.

6	How challenging do you find it to implement vector graphics in said game engines?
Data	
Analysis	The survey reveals differing levels of difficulty in implementing vector graphics in Unity and Unreal Engine. For Unity, the majority of respondents found it *Challenging*, with a smaller number rating it as *Moderate* or *Easy*. Only a few considered it *Very challenging* or *Very easy*. In Unreal Engine, the response distribution is almost the same, with most respondents rating it as *Moderate* or *Challenging*, while fewer found it *Easy* or *Very challenging*. These results indicate that both engines present challenges in implementing vector graphics, suggesting the need for better tools or workflows to ease the implementation process.

3.3.2. Interview Analysis

Question #1: *Could you briefly introduce yourself and your background in UI/UX or game development?*

Spencer	My background is in traditional computer science but, like many, I wanted to build games. I wanted to do so in a big way. I have built 3 game engines for fun, but a first vector-native game engine captivated my interests and is what I am working towards.
David	I have a background in mathematics and hold a mathematics degree. I had done some work on visualizations of scientific models before using BASIC and some general web work in the past. I have also used LaTeX before. I moved out to rust around 2020/2021, and worked on some tile based UI. Later on, I also worked on Bevy UI and contributed quite a lot to it. And then I got recruited by Hedra to make a reactive UI using Bevy.
Raph	My current role is being a research software engineer at Google on the Google Fonts team. The main thing that I worked on there is GPU rendering of 2D graphics and fonts. I was working on Ghostscript for almost 10 years. Before that I also worked on GIMP a litle bit while I was in high school. Around 2007, I went from Ghostscript to Google. I helped co-found and launch the Google Fonts team back in 2010, which is hugely successfully, powering a significant fraction of all of the websites in the internet now. And for almost five years, I was the tech lead on Android, working on the Android UI Toolkit. After that, I took a two and a half year break and intended to make a game, but got interested in GPU. Then Google reached back and I ended up going back as a researcher.
Bruce	I have been programming for 30 years and have spent about 10 years in game development. After that I have also worked on back-end systems. I have written and maintained serveral UI toolkits previously with different level of qualities and usefulness. Now I work on a CAD related product about space planning. I am currently in a small RnD team of four people in a company of 300 to create the next generation of their technology. Because it is in CAD, a lot of it is in vector graphics.

Cheryl	I am a product designer that is currently working in the client fintech industry. I mainly focus on designing for enterprise products and I have been in UI/UX for almost three years, doing freelancing and also some parti-time jobs. I have worked with many clients in various projects ranging from websites to software. In terms of development, I do have a limited practical knowledge as I come from a data science background.
Elise	I am a full-time UI/UX designer for a social media app. I am also freelancing as a UI/UX designer, mostly focused on designing landing pages for different types of companies.
Jon	I am a game developer currently in Metronomik and I am a senior game programmer in a lot of areas and I would like to think that I specialize in UI/UX. My thesis is also specifically in UI/UX and over the years in my development, I am known as the UI programmer.

Analysis

Each participant talks about their background and current work. Our participants have a background on UI/UX development (Cheryl, Elise, and Jon), game development (Spencer, David, Bruce, and Jon), software research and development (Bruce), and 2D graphics research (Raph).

Question #2: *How would you describe your general workflow when designing and implementing vector graphics in applications or games?*

Spencer	I have looked into many approaches for artists as a technology lead and decided to adopt SVG and Lottie. The general flow is to produce still images in a vector program, which could be Illustrator or Figma. Then, animate it in either AfterEffects (with bodymovin). I have also looked into all-in-one tools such as LottieLab, Phase, and Rive, in addition to meeting with their respective CEOs. LottieLab is currently my recommendation to adopt.
David	I made an SDF renderer to render vector graphics for Hedra and use that to rasterize the UI. I also used fragment shaders to draw vector graphics such as gradients and lines, though they are quite low level. Most of them time when integrating SVGs in games, you might want to use an intermediate format that would be more performant.

Raph	I do not really spend a lot of my time building applications or games. But I can describe how I make the interactive demo for research visualizations. The workflow of that is quite personal and idiosyncratic, and most of that is writing Rust code. Sometimes I would start with drawing it on paper, which is then converted into SVG using Inkscape.
Bruce	Unfortunately, we are not artist driven at all as it is difficult especially in a company that does not currently have a large art team (mainly only programmers and industry designers). Right now we tend to look at what we need and code up with something that does it. But with vector, it is also more flexible.
Cheryl	Before you start anything, it is important to understand the project requirements and goals, especially from your clients or from your managers. After everything is set, the design team should sit down and determine the art direction or maybe the UI direction. We rely on third party graphic packs such as unDraw and Figma.
Elise	It really depends on the context. Normally, I include vector graphics as a supporting item instead of just relying on plain texts itself. I use Freepik to download SVGs and Adobe Illustrator files. It is customizable, for example, I could just change the color if I wanted to.
Jon	In game development, developers mostly just try to make things look good, as players are not going to zoom in and look into every pixels. You just need to make sure it looks nice and concise. Implementing user interface is more or less a transferable skill from web development. Developers have to take into account of DPI, scalability in terms of screen sizes, making sure it is dynamic, and most importantly intuitive for the users. This is especially important in mobile games.

Analysis

The responses reveal a broad spectrum of workflows for designing and implementing vector graphics, ranging from traditional design tools to deeply technical, programming-driven approaches. Spencer outlined a structured process involving Adobe Illustrator or Figma for static graphics and After Effects with the Bodymovin plug-in for creating Lottie-compatible animations. He also explored all-in-one tools like LottieLab, Phase, and Rive, recommending LottieLab for its streamlined workflow, though gaps in fully comprehensive solutions remain. Bruce and Raph, operating in resource-constrained environments, rely heavily on programming to create flexible, tailored solutions, with Bruce highlighting the

challenges posed by the lack of a dedicated art team and Raph describing a code-driven workflow in Rust that supports his research visualizations.

David emphasized technical approaches, using SDF renderers and fragment shaders for performance optimization when integrating vector graphics into games, often resorting to intermediate formats for improved efficiency. Smaller teams like Cheryl's and Elise's lean on third-party resources such as Freepik, unDraw, and Figma to simplify their workflows and meet project requirements. Meanwhile, Jon's perspective highlights the importance of scalability and practicality in game development, where UI design requires adaptability to varying DPI and screen sizes, especially in mobile games.

Despite the diversity of workflows and tools, a common challenge emerges: there is no single, all-in-one solution capable of handling the entire workload seamlessly. Artists, designers, and developers must often rely on a combination of tools and techniques, from manual coding to third-party resources, to meet their goals. This fragmentation highlights a significant gap in current workflows, underscoring the need for a comprehensive solution that can streamline the process and reduce the reliance on multiple disconnected tools.

Question #3: *What are some common challenges or frustrations you encounter with current graphics tools or technologies during development?*

Spencer	There is many challenges. I am using WebGPU which is not stable on browsers, Lottie is an ever-changing spec with poor documentation and no versioning standard until the last few months, SVG is an extremely large spec and difficult to reduce to the functions my renderer supports, and my renderer (vello) also has many limitations. However the largest challenge is likely tooling; there is no good workflow for producing 2D lottie assets complete with bones and sophisticated features.
David	Basically cross-platform, always. Had a problem with a bug that only occurs on a specific GPU. Documentation and resources of WGSL/WGPU is not good which poses a barrier.
Raph	One of the fundamentals is just performance, you want to be able to express extreme rich content, which often limits what people can do as well as the general art direction into just static elements instead of animated ones. There is also shortcuts taken to solve visual artefacts instead of solving the root problem.

Bruce	Integrating vector art in 2D space with games in a 3D space is really hard in a lot of systems. Often it is easy to render it into a texture and billboard into the scene but sometimes it is more expensive than rendering just a texture. For Vello, it is a purely 2D solution which does not know anything about the 3D space. It would be cool to have something like a color buffer, a normal buffer, and a depth buffer, because it allows you to do things like shadows, edge detection, screen space ambient occlusion, and many other interesting stuffs. Another frustration is that when you go into the field of vector, you are dealing with people who think only in 2D. Another thing is also about platform supports, where we have to consider whether if technologies will catch up to these new cutting edge technologies like Vello.
Cheryl	I will take Photoshop as an example. It has a lot of features, but in the same time it has a super steep learning curve especially for beginners that have limited knowledge about graphics. Besides that, sometimes it is also quite difficult to communicate your graphical expectations with your developers because they might not know what you are actually thinking.
Elise	There are no main challenges at the moment. However, I wish that the tools provided can allow for more customizability on the spot.
Jon	In game UI development, the tooling is not as robust as web development where you can just drag and drop UI elements. It often involves a lot of coding. It is also much less convenient as there is a lack of tooling available.

Analysis

The responses reflect a wide range of challenges encountered in graphics development, emphasizing both technical and workflow-related obstacles. A common theme among participants, such as Spencer and Jon, is the lack of robust tooling tailored to specific needs. Spencer highlighted the absence of a streamlined workflow for creating sophisticated 2D Lottie assets, while Jon pointed out the inefficiencies in game UI development compared to the drag-and-drop ease of web development tools.

Performance and cross-platform compatibility are recurring issues as well. David discussed the challenges of addressing bugs that arise on specific GPUs, while Raph emphasized the limitations in rendering rich, dynamic content due to performance constraints, often necessitating compromises in art direction. Similarly, Bruce pointed out the difficulties in

integrating 2D vector art into 3D spaces, noting the lack of features like color, depth, and normal buffers in current tools, which could unlock advanced techniques such as shadows, ambient occlusion, and edge detection.

Additionally, the complexity of specifications and documentation poses significant barriers. Spencer noted the challenges of working with large, evolving specifications like SVG and Lottie, which are poorly documented and lack stable versioning. David also mentioned the inadequate resources for emerging technologies like WGSL/WGPU, further complicating development.

The responses also shed light on learning curves and communication barriers in graphic design. Cheryl highlighted the steep learning curve of tools like Photoshop for beginners and the difficulty in bridging the gap between designers and developers. Lastly, Bruce expressed concerns about whether platforms and tools will evolve to support cutting-edge technologies like Vello, emphasizing the uncertainty of adopting innovative solutions in a rapidly changing field.

Question #4: *In terms of visual quality and performance, what impact do you think vector graphics have on the user experience?*

Spencer	Overall, vector graphics will impact UX through resolution-independent UI that is identical on all devices. In terms of enjoyment, I also believe there could be many gameplay innovations made by someone mathematically talented and willing to manipulate the image primitives at runtime. Some other obvious areas of impact would be accessibility with a toolset of rich styling and scaling at your disposal.
David	In terms of performance, it does not matter much, because game UI rarely displays a lot of UI elements. Most of the time, it is text that eats up the performance (though most of the time you are just drawing rectangles). Vector graphics can also be used in diegetic UI concepts which can make it more immersive. You can also perform animations using vector graphics, like animating text.
Raph	The vocabulary of vector graphics is very rich. For text and documents, you need the font to be very high quality. It can also be used for animating text's weight and width dynamically.

Bruce	One big place we see it is in text especially when dealing with stuff outside of Latin text. With vector graphics we do not need to worry about rasterizing bitmaps for different types of texts as it is just as easy to rasterize English texts as any other languages. We also appreciate the fact that everything flows pretty well when you are able to animate stuff around. Some constraints for art pipelines are also gone as you do not need to have massive bitmap graphics for everything. It also allows to be playful, for examples, we can use vector graphics to simulate rope physics when connecting two nodes in a node graph. This provides that extra interactivity and interest, as well as a sense of tangibility and realism. And because it is vector graphics, we also do not need to rely on artist to do everything while being able to provide interesting experience and not worried about the complexity.
Cheryl	I think that vector graphics are really scalable because it can maintain great quality in any form of resolution. In UI/UX we really look into responsiveness as we often design for different screen sizes. It also has smaller file size which is a plus. It is also a great tool to help to guide users' vision and also increase their awareness in certain items.
Elise	Animated vector graphics elements can help improve user attractions. It can also help ease users in understanding the contents. I can also use animated graphics to make users focus on certain areas.
Jon	Vector graphics have a big impact on user experience, but it really depends on the context. For instance, they're excellent for scalability and maintaining quality on different screen sizes, which is great for responsive designs. However, when it comes to more complex animations or detailed visuals, performance can become a challenge. It's always about finding the right balance between technical goals, like resolution independence, and ensuring the user experience is smooth and efficient.

Analysis

The responses emphasize the significant role of vector graphics in enhancing both visual quality and performance for user experiences across various applications. A key advantage highlighted by multiple respondents, including Spencer, Cheryl, and Jon, is the resolution independence of vector graphics, ensuring consistent visual quality across devices and screen sizes. Cheryl and Jon also underscore the importance of scalability and

responsiveness, particularly in UI/UX design, where adaptable graphics are essential for maintaining usability on diverse platforms.

Spencer and Bruce emphasize the potential for innovation with vector graphics, from runtime manipulation of primitives to creative uses like simulating rope physics in node-based systems. Bruce notes that vector graphics reduce dependency on bitmap assets, lightening the art pipeline and offering flexibility for playful and interactive designs without requiring extensive artist input.

From a technical standpoint, David and Raph highlight the impact of vector graphics on performance and animation. David points out that vector graphics are well-suited for diegetic UI concepts and text animations, while Raph elaborates on their rich vocabulary, particularly for text rendering and dynamic adjustments to font weight and width. Bruce and David both touch on how vector graphics simplify handling diverse text styles and languages, enhancing accessibility and global usability.

In terms of user interaction, Cheryl and Elise explore how vector graphics can guide user attention and improve content comprehension. Elise highlights the use of animations to focus user attention, while Cheryl mentions their potential to guide users' vision and enhance awareness of key elements. Jon, however, tempers these benefits with a cautionary note, explaining that complex animations and detailed visuals can introduce performance challenges, necessitating a balance between resolution independence and system efficiency.

Overall, the analysis reveals that vector graphics bring a wide range of benefits to visual quality, scalability, interactivity, and accessibility. However, there is no one-size-fits-all solution; the impact of vector graphics heavily depends on the context of use and the balance between technical constraints and user experience goals.

Question #5: *Do you currently use vector graphics for interactive or dynamic content in your projects? If so, how effective have they been, and what are the main limitations?*

Spencer	I am employing vectors in some unique ways. For example, instead of a minimap, I allow the user to zoom out to see the entire map, which then shows map labels. The main barrier to more enticing visual features, such as generated vector fire, is a high ceiling for jazz-mathematics, which I am not currently focused on but interested in.

David	I have used vector graphics for rendering music visualization before using shaders. It is not really flexible because you always have to pass the data in. As for SDFs, they have complex transformations, and morphing between shapes is not very easy and is actually quite limited.
Raph	*Skipped as Raph does not really create or use vector graphics but rather more on the research side of rendering it.*
Bruce	Yes, they are effective, and their main and biggest limiataions are 2D - 3D integration. Little support for color scales like CubeHelix and Viridis which can maintain the luminance of the color in a grayscale gradient, which would also benefit a lot in terms of printing and accessibility, especially for people with color-blind.
Cheryl	It is not really suitable for most serious scenarios. We need to be careful of the placement as it is a constant trade-off between catching attention and formalism. Sometimes, it is also a technical hurdle as it might not be trivial to implement.
Elise	Yes, I use it as a supporting element. It is quite effective in a sense that I do not need to design from the scratch itself with vector graphics found online such as Freepik.
Jon	They are not effective in games which is why games do not use it at all. We actually do see vector graphics in games, in the form of 3D models. This is achieved through something called the render pipeline. The render pipeline is much more efficient in rendering raster images than calculating mathematical equations for rendering vector graphics. However, there is still a market for it, but you will have to find the niche.

Analysis

The responses reveal mixed experiences and perspectives on using vector graphics for interactive or dynamic content in projects. While vector graphics offer certain advantages, their effectiveness and applicability are often context-dependent, with limitations that can hinder broader adoption.

Spencer and Bruce highlight creative use cases where vector graphics shine. Spencer describes an innovative approach, replacing a traditional minimap with a zoomable map featuring vector-based labels, showcasing their flexibility and scalability. Bruce, on the other hand, emphasizes their potential for accessibility, particularly in implementing

advanced color scales like CubeHelix or Viridis, which could improve usability for colorblind users and support print media.

Despite these benefits, limitations remain a significant concern. David points out the rigidity of using vector graphics in interactive applications, particularly in scenarios requiring morphing or complex transformations, as seen with signed distance fields (SDFs). Similarly, Jon notes that vector graphics are often overlooked in games due to the efficiency and suitability of raster-based render pipelines for performance-critical applications. While vector graphics have niche applications in game development, they face challenges in competing with established 3D workflows.

Cheryl and Elise provide additional insights into practical applications and constraints. Cheryl highlights the trade-offs between visual appeal and formalism, noting that vector graphics can sometimes be unsuitable for serious or technical contexts due to implementation challenges. Elise, meanwhile, values vector graphics as supplementary elements, often relying on pre-designed assets from online resources to save time and effort.

The overarching theme across the responses is the contextual nature of vector graphics' effectiveness. While they excel in scalability, accessibility, and niche use cases, significant barriers remain. These include technical hurdles like complex transformations, limited integration with 3D workflows, and a steep learning curve for certain advanced applications. As a result, vector graphics are most effective in specialized scenarios, but their adoption for broader interactive or dynamic content remains constrained by these challenges.

Question #6: *What tools or techniques do you typically use for integrating motion graphics, such as animations from platforms like Lottie, Rive, or After Effects?*

Spencer	I wrote a custom integration (Bevy Vello) which loads lottie and SVG files as assets into the bevy game framework to render them.
David	Obviously figma, that has been about it. Though Lottie and Rive are quite impressive as well.
Raph	We have done some integration for Lottie (Velato) and Rive. I think that generally when people use these tools they tend to be simple animations, and I think that there is potential to scale up.

Bruce	Right now none, as we are all programmers. But we are looking into creating tools like Interpoli that can help in that sense. There is also stuff like Manim that allows people to create animations via code.
Cheryl	I only tried out Lottie. It is very user-friendly and is able to cater to Figma quite well as there is a plug-in for it. It is also kind of free to use. Another alternative is the Figma prototyping tool, but it is mostly for simpler interactions.
Elise	I use Lottie which has an integration with Figma as a plug-in. The Figma plug-in also allow users to export it into a GIF file, allowing more flexibility. Sometimes I also use After Effects or Figma to create animations from scratch, though Figma is only better for simpler items. This often involves using keyframes to animate vectors from one shape or position to another.
Jon	Almost no one uses those applications in game development. There are three ways motion graphics or animations are performed in games. The first way is to code it out in the engine if you want dynamism. The second way is to use something like GIF (flipbooks) where you rasterize the animations and just perform a playback in the game. The final way is using shaders.

Analysis

The responses highlight a variety of tools and techniques for integrating motion graphics, reflecting the diverse approaches employed across different domains and workflows. While platforms like Lottie, Rive, and After Effects are widely acknowledged for their capabilities, their adoption and application depend heavily on context, team expertise, and project requirements.

Spencer and Raph demonstrate a strong inclination towards custom and scalable solutions. Spencer has developed Bevy Vello for rendering Lottie and SVG files in the Bevy game engine, emphasizing adaptability for game development. Similarly, Raph's work on integrating tools like Lottie (via Velato) and Rive underscores the potential to expand their use beyond simple animations into more complex scenarios.

Cheryl and Elise focus on accessibility and ease of use. Both highlight Lottie's user-friendly integration with Figma through plug-ins, making it a convenient option for designers. Cheryl praises Lottie for its free-to-use model and its ability to cater to Figma workflows, while Elise underscores its flexibility in exporting animations into formats like GIF.

However, both note that while Figma is suitable for simpler animations, more advanced tasks often require tools like After Effects.

Jon and Bruce represent the developer-centric perspective, emphasizing programming-driven approaches. Jon argues that in game development, motion graphics are rarely sourced from platforms like Lottie or Rive. Instead, developers rely on three primary methods: engine-level coding for dynamic animations, flipbook-style GIF playback for rasterized animations, and shader-based techniques for flexibility and performance. Bruce, on the other hand, points to the lack of tools in programmer-driven environments but sees potential in solutions like Manim, which allows animation creation via code.

David acknowledges the appeal of tools like Lottie and Rive but indicates a reliance on Figma for animation-related tasks. His mention of these platforms' impressive capabilities aligns with their growing reputation for bridging the gap between design and development.

Overall, the responses underline that while platforms like Lottie, Rive, and After Effects are powerful tools for integrating motion graphics, their use in game development and interactive applications is often supplemented—or even replaced—by custom or code-driven solutions. The challenges of scalability, flexibility, and real-time performance often necessitate custom solutions, especially in technical contexts like game engines. This indicates that there is no one-size-fits-all solution, and teams must balance ease of use with the technical demands of their projects.

Question #7: *What would you like to see improved in existing tools or new tools for integrating vector or motion graphics in real-time applications or games?*

Spencer	For 2d editors, bones and rigging. But I think I would benefit most from a good tool to transform 3d animations into 2d lottie files, which was done once before by a program called Swift3D back in the days of Shockwave Flash.
David	Improvement in documentations would be great. Also, current tools either have too high level interface with very less control or too low level which requires a lot of technical knowledge. I would also like to see better platform support, well as Rust and Bevy support. I also hope that tools can help reduce the friction, for example, like reducing the intermediate types needed. I also think that dynamic reload is also an important feature.

Raph	I feel like Béziers are not that intuitive and is difficult for people to get really good results in Béziers. And I think that it is more true when you are making animations. I think there are curves that are better than Béziers that we should be using and one of them is Euler spirals (though it is only good for smooth curves). I feel that we can make a curve that has both the smoothness in spirals and the flexbility of Béziers. That is something I would want to see, a better primitives that is more intuitive for designers to use.
Bruce	For the programmer side, better APIs especially in the Rust world. This is also an area where the whole industry are going through big shifts. For example, moving from raster tiles to vector tiles, e.g. Open Street Maps.
Cheryl	I think existing tools can have pre-built components on common patterns and some vector manipulations for developers. Some sort of templating tools will also help in reducing time taken to integrate motion graphics or vectors.
Elise	I would like to see tools that are customizable on the spot.
Jon	I hope that there are more robust animation tools. An example would be to have an animation tool that is inside the game engine itself. There could also be a market for vector graphics tools if there are more basis of proof that current techniques are capable of rendering it with similar or better performance than existing techniques. This could enable a much more creative usage on vector graphics and therefore ignite a better tool set and a section of game development.

Analysis

The responses reflect a variety of improvements desired in existing tools and highlight gaps in current workflows for integrating vector and motion graphics in real-time applications or games. Spencer emphasized the need for 2D editor features such as bones and rigging He also expresses interest in a tool that could transform 3D animations into 2D Lottie files. He referenced Swift3D as an example of such functionality. David pointed out the challenges of tools being either overly simplistic or too technically demanding. He stressed the need for better documentation, platform support (especially for Rust and Bevy), and features like dynamic reloading, which would streamline workflows and reduce friction in integration.

Raph highlighted the difficulty in achieving intuitive results with Bézier curves, proposing alternative curve primitives, such as Euler spirals or new hybrids, to improve usability for designers. He envisioned smoother, more flexible curve tools that would be easier to

manipulate in animation workflows. Bruce echoed the need for better APIs, particularly within the Rust ecosystem, while also noting broader industry trends like the shift from raster to vector tiles in mapping applications. This shift indicates the growing importance of vector-based technologies and the need for more efficient ways to integrate them into real-time environments.

Cheryl suggested that pre-built components and templating tools would significantly reduce development time and improve the integration of motion graphics and vectors. Elise proposed tools with greater customizability on-the-spot, allowing developers and designers to adapt as needed during the creative process. Finally, Jon highlighted the potential for more robust animation tools directly integrated into game engines. He also proposed that if vector graphics could be proven to perform as efficiently as traditional methods, it could drive a creative revolution in their use within game development.

Collectively, these insights suggest a demand for tools that are more intuitive, feature-rich, and accessible, bridging the gap between technical complexity and user-friendliness.

Question #8: *Do you have any feedbacks or suggestions that could help streamline the workflow of incorporating dynamic vector or motion graphics in game engines or real-time applications?*

Spencer	There needs to be as few roadblocks in the way as possible for an artist that wants to hand over an SVG or Lottie to a vector game developer. Rather than asking an artist to “get me 4 sizes of this texture, starting from 32x32 and working up to 1024x1024,” or adjusting a single wrong color, we should be able to make changes to fit our technology without the aide of the original designer.
David	I hope there is much better API to work with GPUs. Reduce the overhead of rendering vector graphics or even custom shaders in general. I also think that performance should be increased as well as flexbility.
Raph	I think we can utilize WASM so that the animation player can hot-reload the WASM so you do not need to restart the application everytime. It is a very hard problem: you need a way of scripting, you need a way of using tools to create and manage assets.
Bruce	Better tooling, better API supports, and better hardware support.

Cheryl	We could create tools to gather feedback on how the animations and actions impact user experience such as checking attention and interaction patterns. This can help designers understand the most effective designs for users. Impact assessment can also be implement in the form of heat maps. A cool example is Hot Jar.
Elise	Mostly just about the customization of tools and the ability to do everything all in one place.
Jon	Currently rendering vector graphics in games is almost always done using custom shaders. This takes a lot of time and effort to setup which is often time seen as too much of a cost. People will want to know if it is worth it compared to how it is normally done using existing tools.

Analysis

The feedback regarding streamlining workflows for incorporating dynamic vector or motion graphics in game engines or real-time applications emphasizes several key areas for improvement. Spencer advocates for reducing roadblocks between artists and developers, suggesting that artists should be able to provide assets like SVGs or Lottie files without needing to accommodate multiple sizes or color adjustments. This would allow developers to make necessary adjustments independently, streamlining the process.

David proposes that better APIs for GPU interactions could reduce overhead in rendering vector graphics and custom shaders, leading to both improved performance and flexibility. Raph suggests utilizing WebAssembly (WASM) to enable hot-reloading of animations, which would eliminate the need to restart applications when updating assets. However, he acknowledges that this is a complex challenge requiring better scripting, asset management, and tool integration.

Bruce calls for better tooling, API support, and hardware compatibility to enhance workflows, while Cheryl suggests creating tools to assess the impact of animations on user experience. For example, using heat maps to measure attention and interaction patterns could provide valuable insights to designers. Elise emphasizes the importance of tool customization and the ability to manage everything in a unified platform, simplifying the workflow for developers and designers. Finally, Jon points out that the process of rendering vector graphics in games often requires custom shaders, which can be time-consuming and might be seen as an inefficient cost compared to using more traditional rendering methods.

These feedbacks underscores the need for more efficient, integrated tools that balance technical complexity with user-friendly features.

3.3.3. Conclusion of Interview Analysis

The analysis of all eight questions highlights the diverse perspectives and experiences of participants, offering valuable insights into the current state and future potential of integrating vector and motion graphics into real-time applications and games.

Participants' backgrounds demonstrate a blend of UI/UX design, game development, and technical research, which informs their varied workflows. While some rely on traditional design tools like Figma and After Effects, others prefer programming-centric approaches tailored to specific project needs. A recurring theme is the lack of a single, comprehensive solution, requiring teams to combine multiple tools to meet their objectives.

Challenges are multifaceted, ranging from limited tooling and poor documentation to performance constraints and integration hurdles. Participants noted gaps in flexibility, scalability, and ease of use, particularly in cross-platform environments. Vector graphics' scalability and visual quality are universally acknowledged, but they remain underutilized in game development due to performance trade-offs and integration complexities.

Motion graphics tools like Lottie and Rive are valued for their user-friendly interfaces, yet their applicability in technical contexts often requires custom solutions. The feedback emphasizes the need for better APIs, dynamic reloading capabilities, and intuitive interfaces for animation workflows. Additionally, participants see a growing demand for advanced features like 2D rigging, robust curve manipulation, and pre-built components to simplify development.

Key suggestions for improvement include bridging the gap between designers and developers, enabling hot-reloading of assets, and providing better tools for evaluating user interaction with animations. Enhanced APIs for GPU utilization and tools for streamlined asset integration are also critical for reducing workflow friction.

In conclusion, the findings underscore a clear demand for innovative tools that combine intuitive usability with technical flexibility. Addressing these gaps could unlock new possibilities for vector and motion graphics, making them integral to real-time applications and game engines.

CHAPTER 4: Design and Implementation

4.1. Introduction

Figure 27: Velyst logo

Figure 28: Lumina logo

This project is split into two parts: **Velyst** and the integration proof-of-concept — **Lumina**. Velyst is created as a separated Rust crate that is completely independent of Lumina. Lumina utilizes the Velyst crate[1] for all (or at least most) of its UI/UX creation. This includes titles, buttons, labels, icons, in-game overlay, and many more!

Velyst is an interactive Typst content creator using Vello as the renderer and Bevy as the game engine for driving logic and user interactions. At it's core, Velyst aims to simplify the integration of Typst into the Bevy game engine. The goal is to allow developers to create dynamic vector graphics content using Velyst. Velyst can also be used in combination with other technologies such as SVG and Lottie files. Developers and artists will be able to leverage the full capabilities of Typst for authoring logic and animation directly inside the language. Below is a table of developer productivity tools developed inside Velyst to streamline the workflow:

Hot-reloading	Whenever the Typst source file changes, it will automatically be re-evaluated on the fly. The evaluated source file will produce an accelerated data structure for compilation and layouting at a later phase.
Function Macros	Velyst expects developers to create functions in the Typst source file and run them within Rust. This is particularly repetitive as developers will need to constantly define the function name and arguments. Velyst makes this simple by introducing a procedural macro for handling all these on a normal Rust `struct`.
Auto Compilation	This ties hand-in-hand with the function macros, all `struct` that derive the function macro will automatically be eligible for the auto compilation process.

[1] A crate is just a Rust library.

Auto Layout	Once the compilation is complete Velyst will automatically layout the compiled content. The layouted content will result in a Typst frame which can then be converted into a Vello scene.
Rendering	The converted Vello scene is automatically rendered onto the screen with the help of the Bevy Vello crate. It utilizes the GPU for accelerated vector graphics rendering.

Concurrent with the development of Velyst, the Lumina game is being built to investigate Velyst's potential use cases in a real world production scenario. Lumina is a fast paced, top down, 2D, objective based, player versus player (PvP) game. In the game, players pilot spaceships to complete objectives by depositing lumina into the tesseract to push the tesseract meter. Players of different teams can also eliminate each other during the match in an attempt to obtain an upper hand. To win the game, one team will need to push the tesseract meter completely towards the opposing side to obtain total dominance.

The game is heavily physics-driven, with each spaceship consisting of a physics body that interacts dynamically with the environment. It employs client-side prediction to ensure seamless and responsive physics interactions, maintaining a smooth gameplay experience. To uphold fair play, all predictions are validated against the server, enabling a fully server-authoritative system that prevents cheating.

Lumina will also employ a custom implementation of global illumination solution using the Radiance Cascades method (Osborne & Sannikov, 2024). Unlike most 2D games, this game does not use any point or spot lights to illuminate the scene. Instead, it relies on emissive materials in the scene. The algorithm will be developed using compute shaders and runs entirely on the GPU.

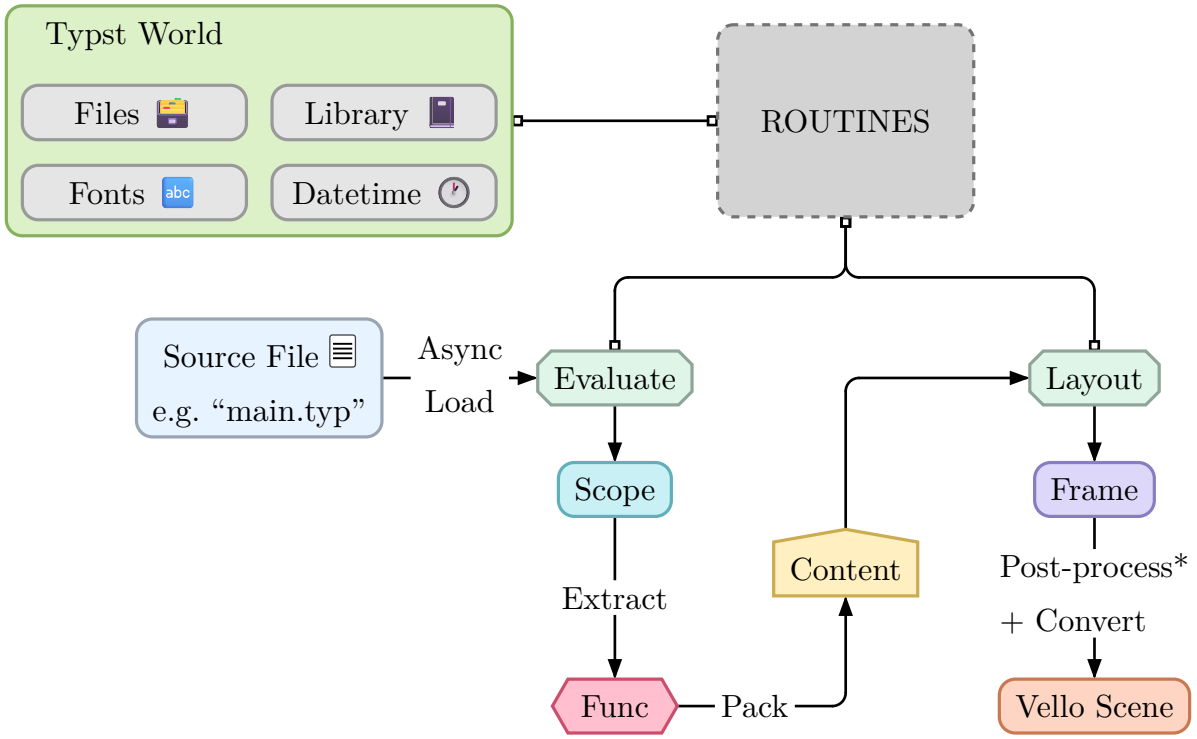
Lumina features four different levels:

Local Lobby	Loaded when the player first enters the game.
Multiplayer Lobby	The waiting area when player enters matchmaking.
Sandbox	Playground and tutorial level where player can get to know the game better.
Abandoned Factory	The in-game map of the game where the deathmatch happens.

The following sections will dive into the design architecture of Velyst and how it is being integrated into the Lumina game. We will also briefly dive into how Lumina works to provide a better overall understanding of the underlying systems.

4.2. Design

4.2.1. Velyst Architecture



* *indicates optional steps.*

Figure 29: Velyst architecture

Figure 29 illustrates the architecture of Velyst, detailing the process of transforming a Typst source file into a Vello scene.

Typst World

Velyst depends strongly on the Typst world for providing the four foundation pillars:

1. **Files**: Responsible for loading and caching the *.typ source files and other assets, for instance, images, vector graphics, tables, etc.
2. **Library**: The Typst standard library and user defined variables and functions.
3. **Fonts**: Keeps track of all the fonts available to the compiler.
4. **Datetime**: Provide the current date and time, down to the second!

```rust
/// World for compiling Typst's [`Content`].
pub struct TypstWorld {
    /// The root relative to which absolute paths are resolved.
    root: PathBuf,
    /// Typst's standard library.
    pub library: LazyHash<Library>,
    /// Metadata about discovered fonts.
    book: LazyHash<FontBook>,
    /// Locations of and storage for lazily loaded fonts.
    fonts: Vec<FontSlot>,
    /// Maps file ids to source files and buffers.
    slots: Mutex<HashMap<FileId, FileSlot>>,
    /// The current datetime if requested. This is stored here to ensure it is
    /// always the same within one compilation. Reset between compilations.
    now: OnceLock<DateTime<Local>>,
}
```

Figure 30: Typst world code snippet

ROUTINES

The Typst compiler also stores a number of "routinely" used methods inside a static global variable — `ROUNTINES`. Amongst them, the evaluate — `eval(..)` and layout — `layout_frame(..)` method is used extensively inside the Velyst pipeline.

Pipeline

```rust
pub enum VelystSet {
    /// Loading and reloading of [`TypstAsset`].
    AssetLoading,
    /// Compile [`TypstFunc`] into a [`TypstContent`].
    Compile,
    /// Layout [`Content`] into a [`TypstScene`].
    Layout,
    /// Render [`TypstScene`] into a [`VelloScene`].
    Render,
}
```

Figure 31: The Velyst render pipeline system sets code snippet

The Velyst pipeline might vary slightly depending on the use case. However, in most scenarios, it begins with creating a native Typst source file within the assets folder. This source file will be loaded asynchronously into Bevy at runtime through the Bevy asset pipeline. Once loaded,

the Typst compiler *evaluates* the source file into a `Scope`, which contains everything defined within the source file.

Velyst allow developers to run any functions defined in the Typst source file. During this phase the chosen Typst function will be extracted from the `Scope` and then packed into a `Content` data structure. The `Content` is later being used to *layout* into a `Frame`. Lastly, developers can apply any final post-process needed to the `Frame` which will then be converted into a `VelloScene`.

What is a `Content`? *Info*

A `Content` is a universal "type" in Typst that packs the information needed to create a renderable `Frame` view. It can be created completely from scratch in Rust or compiled from markdown. On rare occasions where developers need to create a `Frame` completely in the Rust language, they can do so by creating a `Content`. In addition to that, developers can also modify a `Content` using a utility crate that Velyst provides called ***Typst Element*** (see Section 4.2.2).

What is a `Frame`? *Info*

A `Frame` is a vector graphics format created by *laying out* a `Content`. It primarily consists of the shapes, texts, fills, and strokes needed to draw the vector graphics content. This also means Velyst can use any rendering backend, with Vello being the one currently in use. Velyst also offers a simple post-processing step that developers can apply to a `Frame` for direct vector graphics manipulation using the ***Typst Vello*** utility crate (see Section 4.2.2).

4.2.2. Velyst Utility Crates

Velyst provides two utility crates: ***Typst Element*** and ***Typst Vello***.

4.2.2.1. Typst Element

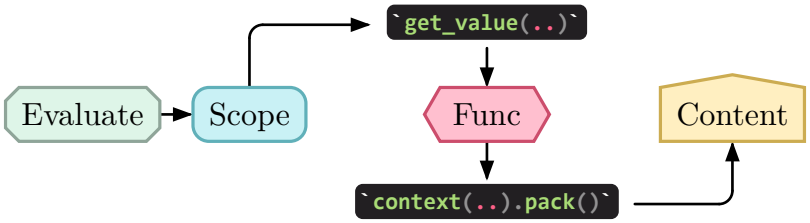


Figure 32: Typst Element utilities

This crate provides utility tools for creating, modifying, and styling a `Content`. The most crucial function from Typst Element is the `context(..)` function (Figure 33). It allows developers to to run any function from the `Scope` and produces a context element which can be packed into a `Content`.

```rust
pub fn context(
    func: foundations::Func,
    apply_args: impl Fn(&mut SpannedArgs),
) -> foundations::ContextElem {
    let mut spanned_args = SpannedArgs::new(func.span());
    apply_args(&mut spanned_args);
    foundations::ContextElem::new(func.with(&mut spanned_args.args))
}
```

Figure 33: Context function code snippet

The `Func` argument required for the `context(func, ..)` function can be obtained from the scope extension trait:

```rust
impl ScopeExt for Scope {
    fn get_value<T: FromValue>(&self, var: &str) -> Result<T, ScopeError> {
        self.get(var)
            .ok_or(ScopeError::VariableNotFound)
            .and_then(|value| {
                value.read().clone().cast::<T>().map_err(ScopeError::ValueCastFailed)
            })
    }
}
// Example usage:
let func = scope.get_value::<Func>("func_name").unwrap();
```

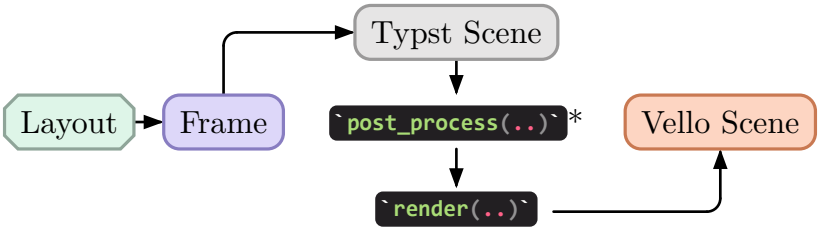
Figure 34: Scope extension code snippet

This crate also provides a huge number of other utility functions using the `fn_elem!(..)` macro as shown in Figure 35.

```
macro_rules! fn_elem {
    ($fn_name:ident, $elem:ty, $($param:ident = $in_elem:ty),+) => {
        #[doc = concat!("[", stringify!($elem), "]")]
        pub fn $fn_name($($param: impl Into<$in_elem>),+) -> $elem {
            <$elem>::new($($param.into()),+)
        }
    };
}
// Example usage:
fn_elem!(sequence, foundations::SequenceElem, Vec<Content>);
// And many more...!
```

Figure 35: `fn_elem!(..)` macro and example use cases

4.2.2.2. Typst Vello



* *indicates optional steps.*

Figure 36: Typst Vello utilities

Typst Vello exposes functionalities that allows developers to apply vector graphics post-processing and convert a `Frame` into a `VelloScene`. The core data structure of Typst Vello is a `TypstScene` (Figure 37). It flattens a vector graphics scene into a vector of groups and allows post-processing to any labeled group content.

```
pub struct TypstScene {
    size: kurbo::Vec2,
    group_scenes: Vec<TypstGroupScene>,
    group_map: AHashMap<Label, SmallVec<[usize; 1]>>,
    pub post_process_map: AHashMap<Label, PostProcess>,
}
```

Figure 37: Typst scene code snippet

Below shows how you can label a piece of content in the Typst markdown structure:

```
#box(fill: blue, inset: 1em)[This is a piece of content.] <my-label>
```

Figure 38: Labeled markdown content.

Finally, ***Typst Vello*** exposes a `render(..)` function to render a `TypstScene` into a `VelloScene`:

```rust
pub fn render(&mut self) -> vello::Scene {
    let mut scene = vello::Scene::new();
    // Setup scene stuff...
    for (i, group_scene) in self.group_scenes.iter_mut().enumerate() {
        // Render the group scenes...
    }
    scene
}
```

Figure 39: Render function code snippet

4.2.3. Integration with Bevy's ECS World

The Bevy's ECS world is responsible for managing the entire game logic in **Lumina**. It is crucial for the **Velyst** architecture to integrate seamlessly with Bevy's ECS world. Velyst uses common Bevy ECS data structures, specifically the *assets*, *resources*, *entities*, and *components* to communicate between the Typst world and the ECS world.

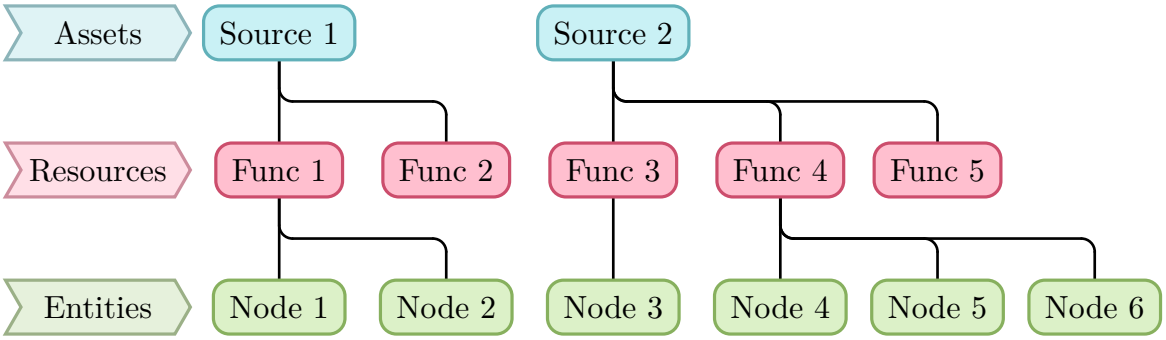


Figure 40: Velyst-ECS communication diagram

When a native Typst source file (*.typ) is loaded into Bevy, it will be stored in Bevy's asset server. The asset server is responsible for loading, and hot-reloading changes from the source file. Each source file may contain several Typst functions which can be used to create renderable `VelloScene` in Bevy. These functions are needed to be defined in the Rust source code too. Fortunately, Velyst provides a simple `TypstFunc` procedural macro for the creation of these definitions (Figure 42).

```
#let main(data) = {..}
```

Figure 41: Native Typst function

```
#[derive(TypstFunc, Resource, Default)]
#[typst_func(name = "main")]
struct MainFunc {
    data: Option<Dict>,
}
```

Figure 42: `TypstFunc` procedural macro

As shown in Figure 40, each function definition is also required to be a Bevy *resource* so that the Velyst pipeline can pick up information from them. Inside a Typst function, developers can label parts of their boxed content which will eventually be turned into an entity as an interactable Bevy UI `Node`. Note that not all functions contain nodes, it should only be added if interaction is required between the content and the user.

```
#let button(label) = [#box[#label] <my-button>]
```

Figure 43: Labeled boxed content in Typst

Finally, Velyst also makes it extremely simple to register these bindings with a simple app extension trait:

```rust
pub trait VelystAppExt {
    /// Load [`TypstAsset`] and detect changes made towards the asset.
    fn register_typst_asset<P: TypstPath>(&mut self) -> &mut Self;
    /// Compile [`TypstFunc`] into [`TypstContent`].
    fn compile_typst_func<P: TypstPath, F: TypstFunc>(&mut self) -> &mut Self;
    /// Layout [`TypstContent`] into [`VelystScene`] and render it into a [`VelloScene`].
    fn render_typst_func<F: TypstFunc>(&mut self) -> &mut Self;
}
// Example usage:
impl Plugin for MainUiPlugin {
    fn build(&self, app: &mut App) {
        app.init_resource::<MainFunc>()
            .register_typst_asset::<MainUi>()
            .compile_typst_func::<MainUi, MainFunc>()
            .render_typst_func::<MainFunc>();
    }
}
```

Figure 44: App extension trait

4.2.4. Lumina

Under the hood, **Lumina** fully utilizes Bevy ECS to handle everything that happens in the game, from user interaction, to visual effects, and rendering.

4.2.4.1. Spaceships

There is a total of two spaceships available in Lumina — **Assassin** and **Defender**. Each spaceship has a common ability — *Dash*, and a unique ability — *Shadow* for **Assassin** and *Heal* for **Defender**.

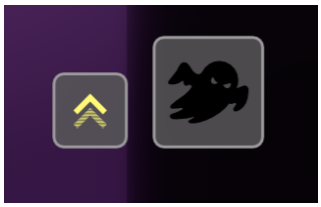
 	**Assassin** A fast and agile spaceship, specialized in stealth and precision strikes. It has lower firing rate but higher damage points. It moves slightly faster than the defender spaceship. The bullets of the assassin also travels faster. The *Shadow* ability allows it to travel unseen (Figure 45).
 	**Defender** A highly durable spaceship, with strong shields for defense-focused gameplay. The defender spaceship has significantly more health than the assassin. The bullets of the defender travels slower, but it has higher firing rate. The *Heal* ability allows the spaceship to regain health of all the spaceship in its team including itself (Figure 46).

Figure 45: Assassin Shadow visual effect

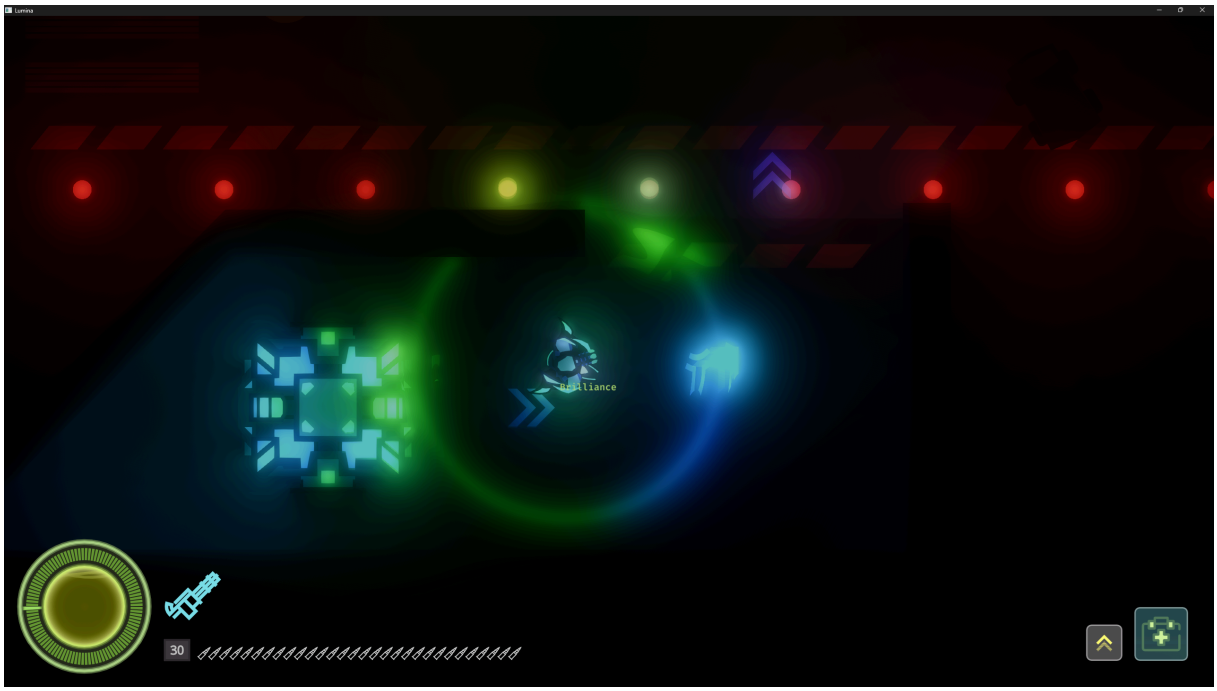


Figure 46: Defender Heal visual effect

4.2.4.2. Maps

Lumina also comes with a total of four maps:

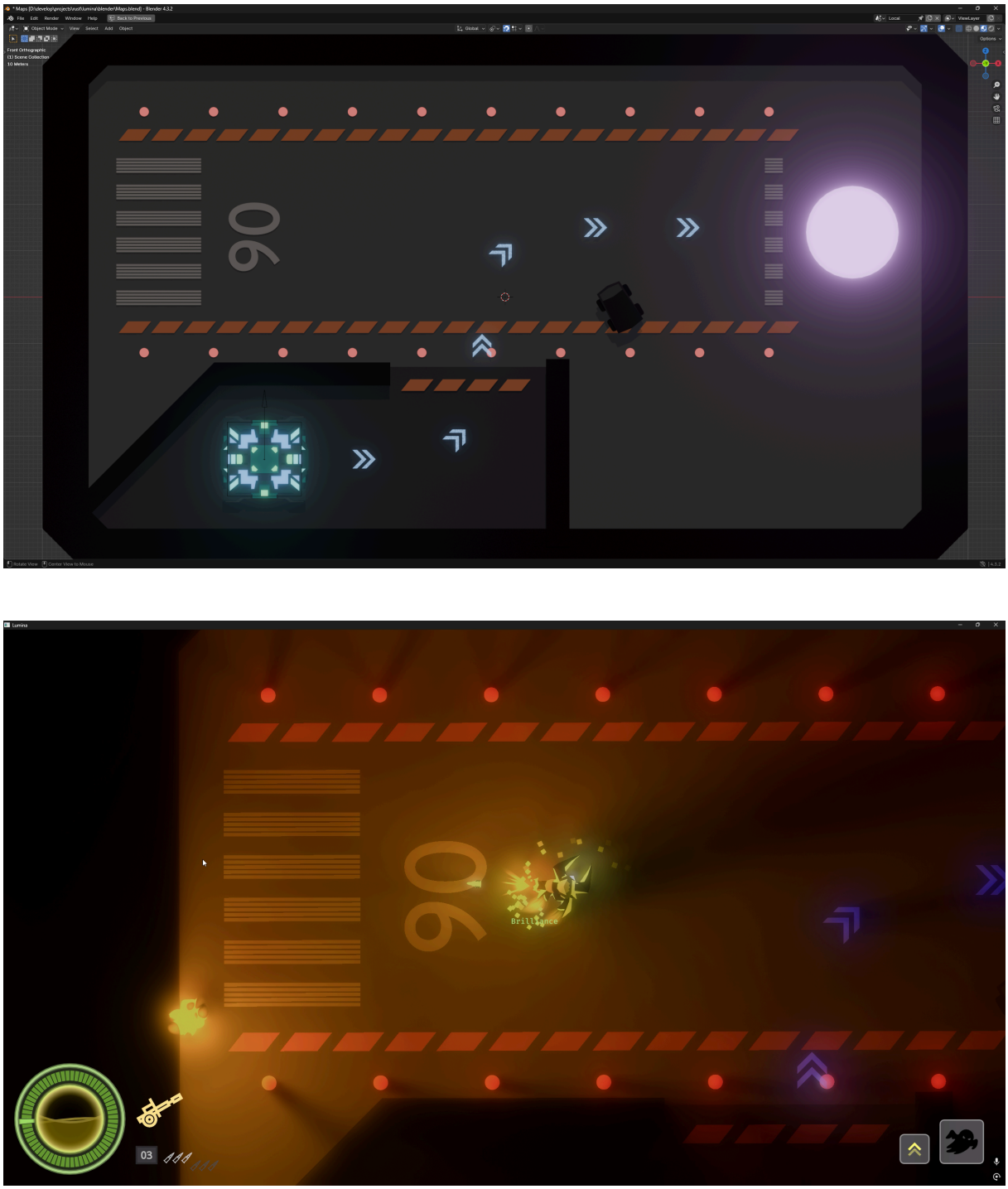


Figure 47: Local lobby (Map 1)

The Local lobby map is where players start when they enter the game.

Figure 48: Sandbox (Map 2)

New players can choose to enter the Sandbox map to test out controls and learn how to play the game.

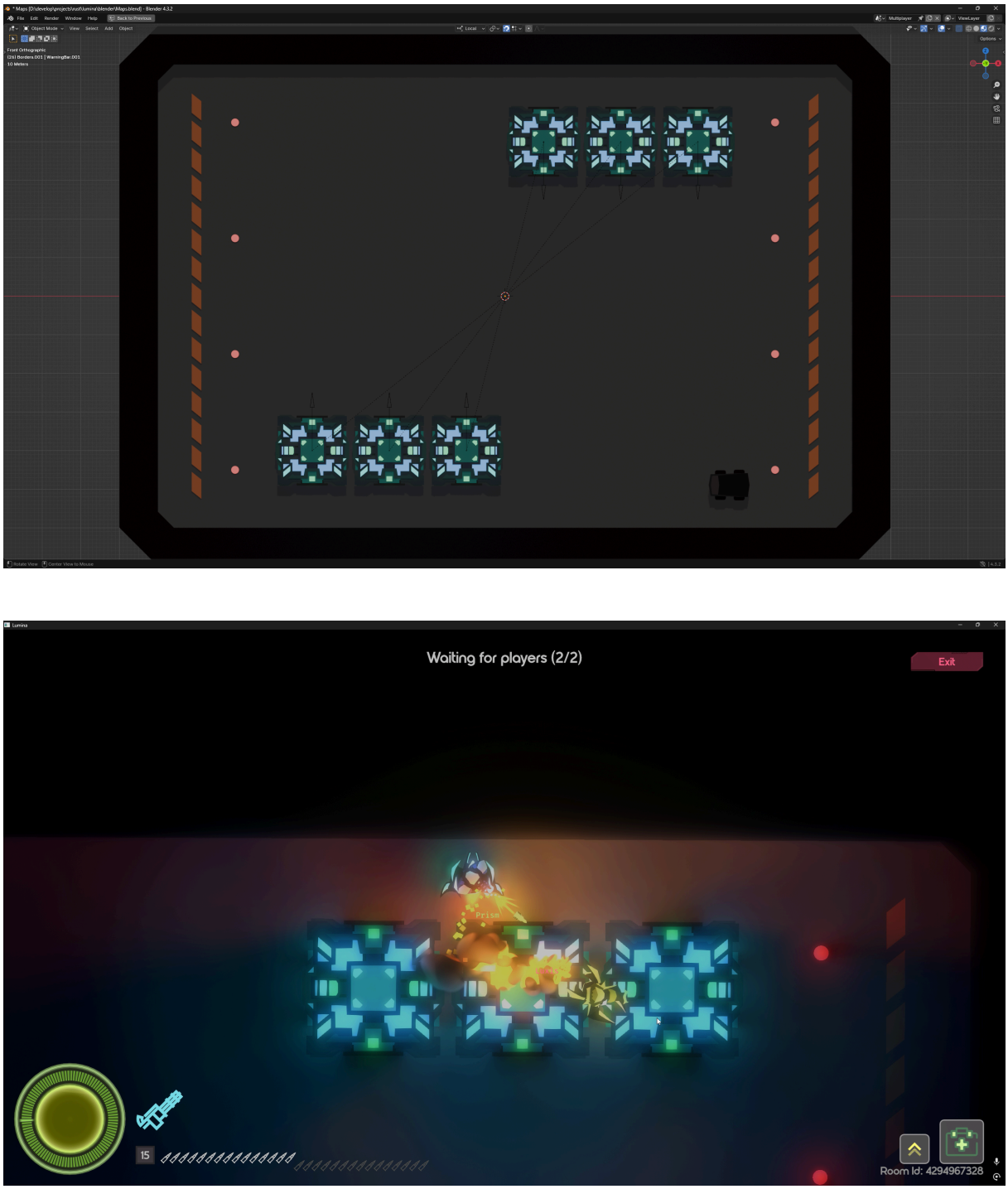


Figure 49: Multiplayer lobby (Map 3)

When players enter the matchmaking mode (1v1/2v2/3v3), they will be teleported into the Multiplayer lobby where they can meet their team mates and opponents in real-time as they join. Once the lobby is full, they will enter the Abandoned Factory.

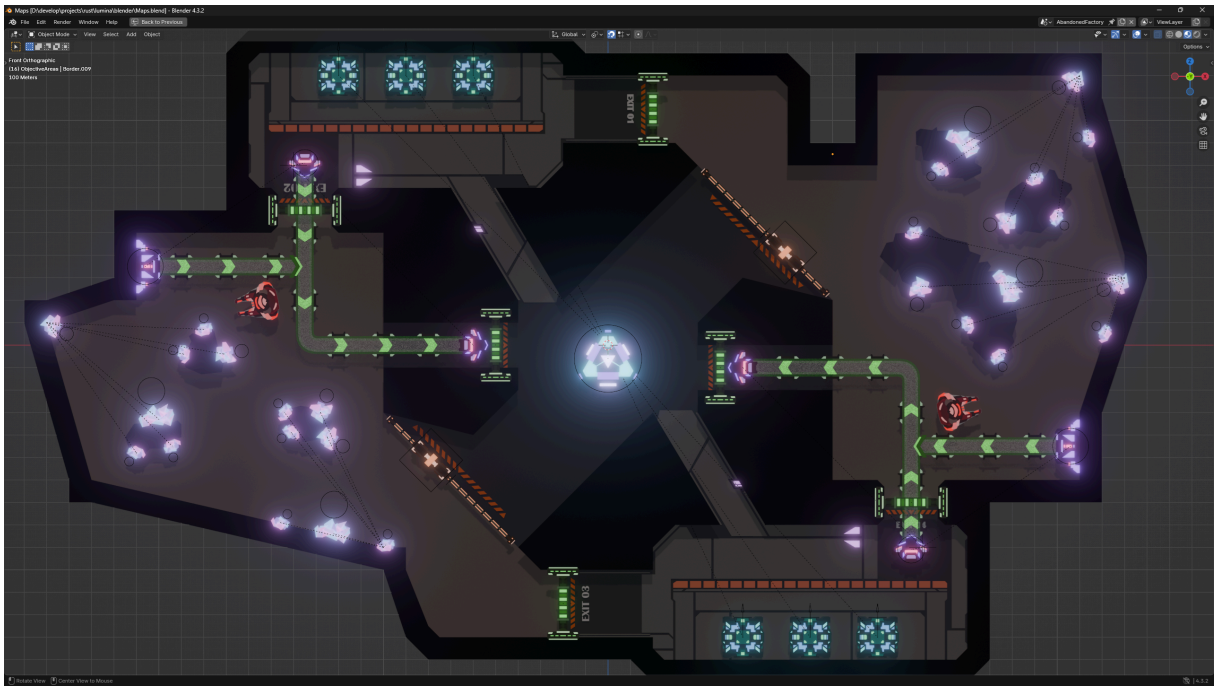


Figure 50: Abandoned factory (Map 4, in-game map)

The Abandoned Factory is the map where opposite teams compete between each other.

4.2.4.3. Objective

The objective of the game is to destroy ores (Figure 51), collect luminas, and deposit it into the tesseract (Figure 52)!

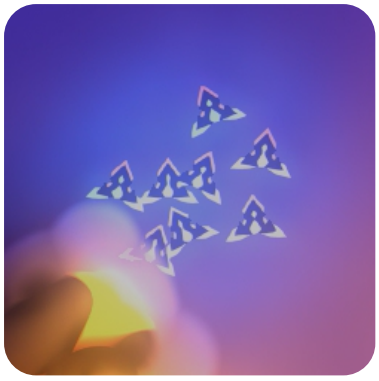
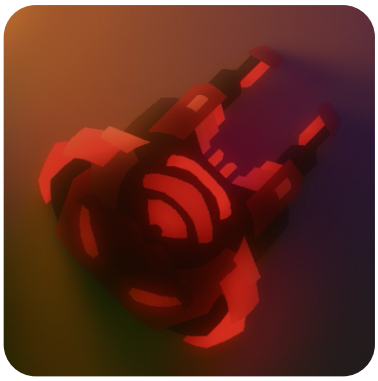
Figure 51: Destroy ores

Figure 52: Deposit Lumina into tesseract

An alternative to obtain Lumina apart from destroying ores is by destroying the opponent's spaceships. When a spaceship is destroyed, all Lumina that it is holding will be dropped onto the map. Each deposited Lumina will push the bar (shown in Figure 53) towards the other end (right in the player's view). The team that pushes the bar towards the end will win the game!

Figure 53: Tesseract effect bar

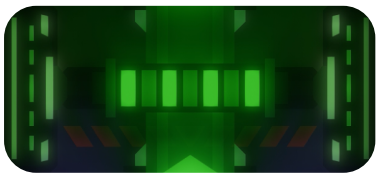
4.2.4.4. Miscellaneous

 *Figure 54: Lumina*	 *Figure 55: Small ore*	 *Figure 56: Medium ore*
The points that the player collects and deposit into the Tesseract.	When destroyed, 1-2 Luminas will drop into the map.	When destroyed, 3-5 Luminas will drop into the map.
 *Figure 57: Large ore*	 *Figure 58: Spawn point*	 *Figure 59: Laser*
When destroyed, 5-8 Luminas will drop into the map.	The location where players spawn.	Will fire occasionally and damage any spaceship within its range.

<table>
<tr>
<td> 
Figure 60: Teleporter

Teleport spaceships to the other end of the teleporter.</td>
<td> 
Figure 61: Teleporter base

Teleport spaceships to the other end of the teleporter.</td>
<td> 
Figure 62: Bullet

A glowing bullet that applies damage on opponents and ores.</td>
</tr>
<tr>
<td> 
Figure 63: Door

Decorative door.</td>
<td> 
Figure 64: Moving door

A door that moves from start to end in a loop. This door will obstruct spaceships and bullets from going through.</td>
<td></td>
</tr>
</table>

4.3. Interface Design

Custom SVG assets are utilized and created for the buttons, cards, and icons.

4.3.1. Menu Designs

Figure 65: Main menu

The start menu consists of a start button, a settings button, and an exit button.

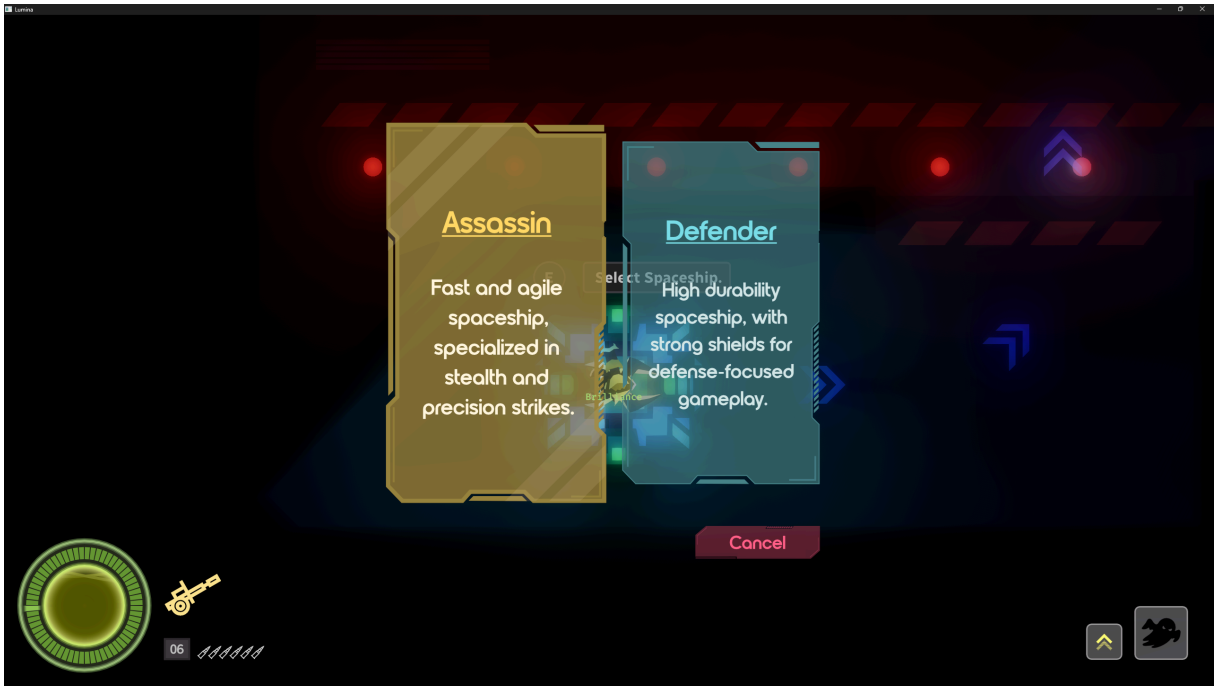


Figure 66: Spaceship select menu

The spaceship select menu allows player to select between the different types of spaceships.

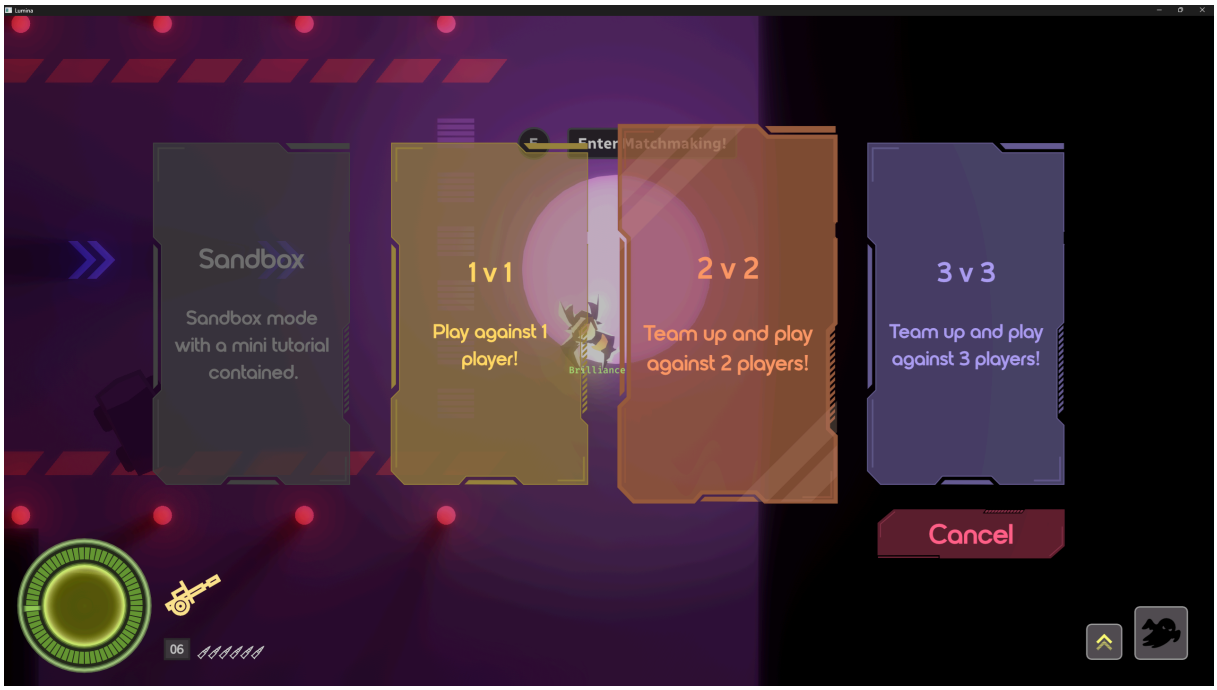


Figure 67: Matchmake select menu

The matchmake select menu allows player to choose either sandbox or a team versus match.

4.3.2. Spaceship Stats

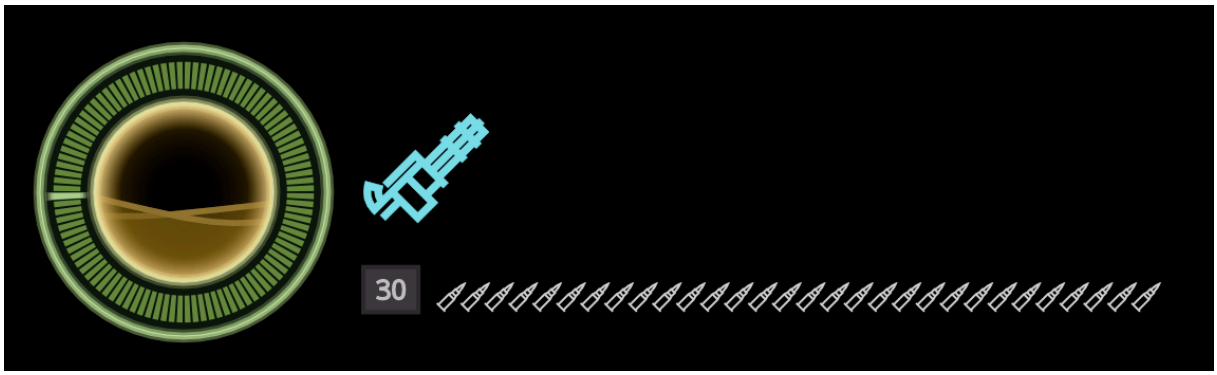
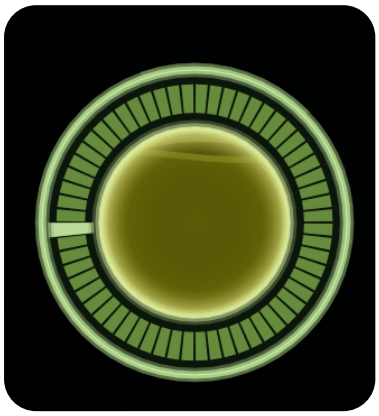
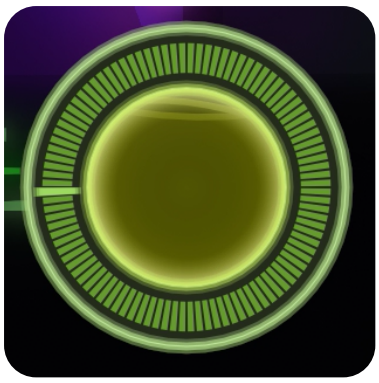
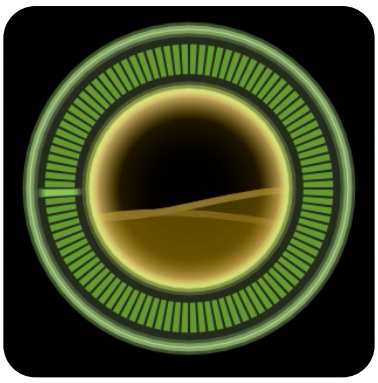
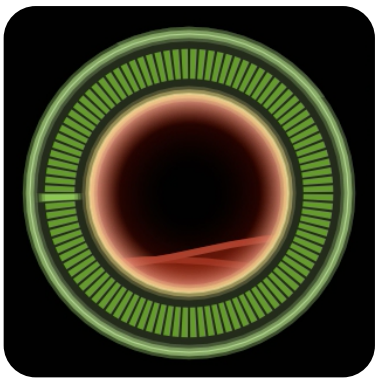


Figure 68: Spaceship stats

The spaceship stats consists of the spaceship's health, fuel, weapon selection, and the ammo count (Figure 68). Below are the possible scenarios for the spaceship's health and fuel UI:

 *Figure 69: Lumina*	 *Figure 70: Small ore*	 *Figure 71: Medium ore*
Complete green indicates a full health.	Health color will interpolate from green to reddish as it goes down.	Health color goes towards red when nearing death.

 *Figure 72: Full fuel*	 *Figure 73: Half fuel*	 *Figure 74: Empty fuel*
Complete yellow indicates full fuel.	Fuel meter slowly interpolate from yellow to reddish as it goes down.	Fuel meter will turn completely red on empty.

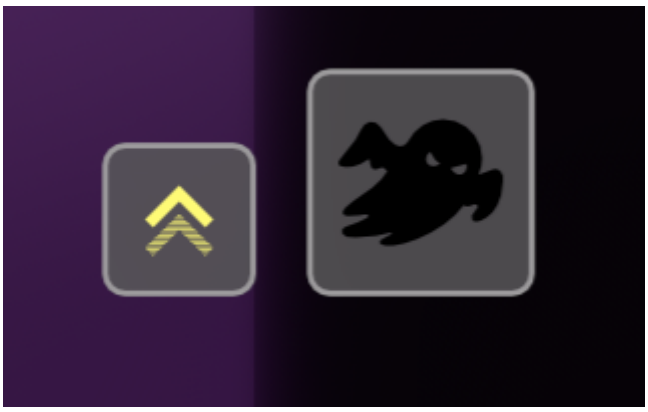
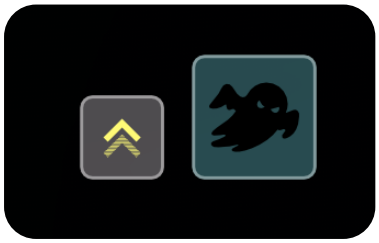
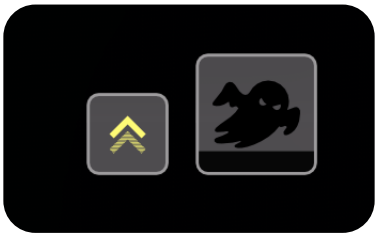


Figure 75: Spaceship abilities

The spaceship abilities UI indicates the current state of abilities (triggered, effective, cooldown).

 *Figure 76: Ability active*	 *Figure 77: Ability cooldown*	 *Figure 78: Dash cooldown*
Blue background indicates that the ability is currently active.	A cooldown shade indicates the percentage of the cooldown.	Works similar to the ability cooldown. (No active for *Dash* as the effect is immediate.)

4.4. Execution

4.4.1. Technology Stack

Lumina is made mainly using the Rust language. This means that most of the dependencies used will also be from the Rust ecosystem.

Here is the breakdown of the Lumina tech stack:

Component	**Tool/Library**
Game Engine	*Bevy*
UI/UX	*Velyst*
Global Illumination	*Radiance Cascades, an improvement from here*
Particle System	*Bevy Enoki*
Physics	*Avian*
Networking	*Lightyear*
Asset Management	*Blenvy*
Input Manager	*Leafwing Input Manager*
Coroutine	*Bevy Coroutine*
Motion Graphics	*Bevy MotionGfx*

4.4.2. UI Creation Workflow

This section will discuss how a user interface is designed, created, and implemented into the game using the Velyst pipeline discussed earlier. We will take the main menu (Figure 65) as an example:

4.4.2.1. Create Assets

Create the assets needed for the UI layout. This normally includes creating the SVG files using softwares like Graphite.rs or Adobe Illustrator. Some assets we created for the game includes:

Figure 79: Button 1

Figure 80: Button 2

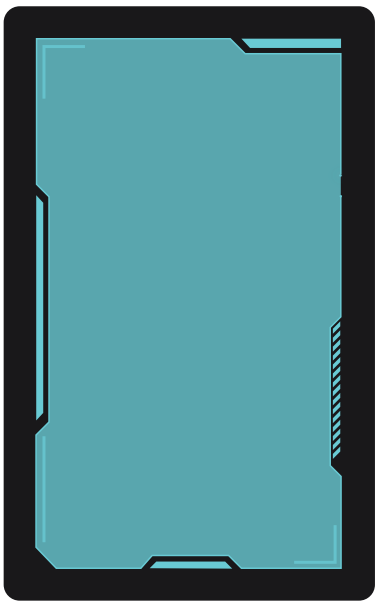


Figure 81: Card

4.4.2.2. Layout Content

Create the native Typst source file to layout the contents of the user interface.

```
#import "../monokai_pro.typ": *
#import "../utils.typ": *

#let main_menu(connected, connection_msg, dummy_update) = {
  box(width: 100%, height: 100%, inset: 4em)[
    #if connected == false {
      connect_server(connection_msg, dummy_update)
      return
    }

    #place(center + horizon)[
      #text(fill: yellow, size: 7em, font: "IBrand")[= Lumina]

      #text(fill: green, size: 2em)[
        #button(lbl: <btn:play>, inters: interactions())[== Start Engine!]
      ]
    ]

    #place(right + bottom)[
      #box(height: 3em)[
        #button(lbl: <btn:settings>, inters: interactions())[== #emoji.gear Settings]
      ]

      #box(height: 3em)[
        #text(fill: red)[
          #button(lbl: <btn:exit-game>, inters: interactions())[== Abort]
        ]
      ]
    ]
  ]
}
```

Figure 82: Main menu Typst source file

4.4.2.3. Definition of Source File and Function

The next step is to link the *.typ source file in Rust (Figure 83) and create a `struct` to match the definition of the Typst function (Figure 84).

```
#[derive(TypstPath)]
#[typst_path = "typst/client/main_menu.typ"]
struct MainMenuUi;
```

*Figure 83: Link *.typ source file*

```rust
#[derive(TypstFunc, Resource, Default)]
#[typst_func(name = "main_menu", layer = 1)]
pub struct MainMenuFunc {
    connected: bool,
    connection_msg: String,
    dummy_update: u8,
}
```

Figure 84: Define `struct` "function"

4.4.2.4. Register Asset and Typst Function

The final and last step is to register the asset file and function using the `Plugin` system in Bevy:

```rust
impl Plugin for MainMenuUiPlugin {
    fn build(&self, app: &mut App) {
        app
            // Register source file as asset.
            .register_typst_asset::<MainMenuUi>()
            // Register the function for compilation.
            .compile_typst_func::<MainMenuUi, MainMenuFunc>()
            // Register the function for rendering.
            // This step is optional, as users may need the compiled Content only.
            .render_typst_func::<MainMenuFunc>()
            // Insert function as Resource.
            .insert_resource(MainMenuFunc {
                connection_msg: "Connecting to server...".to_string(),
                ..default()
            });
    }
}
```

Figure 85: Register asset and Typst function

The function itself is also registered as a Bevy `Resource` so that it can be obtained from Bevy's systems to modify the arguments of the function when needed. Here's an example on how to do it:

```rust
impl Plugin for MainMenuUiPlugin {
    fn build(&self, app: &mut App) {
        // Registering of assets and functions goes here ...
        app.add_systems(OnEnter(Connection::Disconnected), disconnected_from_server);
    }
}

fn disconnected_from_server(
    mut func: ResMut<MainMenuFunc>,
) {
    func.connected = false;
    func.connection_msg = "Disconnected...".to_string();
}
```

Figure 86: Using Bevy's system to modify arguments of a Typst function

4.4.3. Blender Asset Workflow

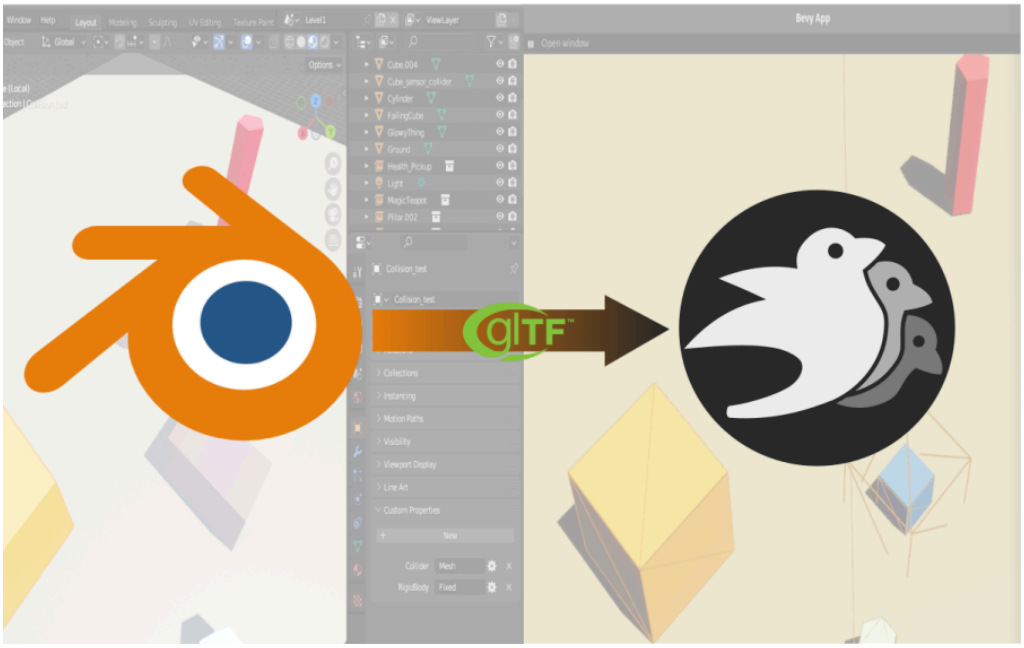


Figure 87: Blenvy

Blenvy is an tool for using Blender as the level editor for Bevy. At the time of writing this report, Bevy has yet to developed a functional editor. The first step of the Blenvy workflow is to create assets within Blender. Next, using the Blenvy's Blender add-on, users can attach components to the objects in Blender:

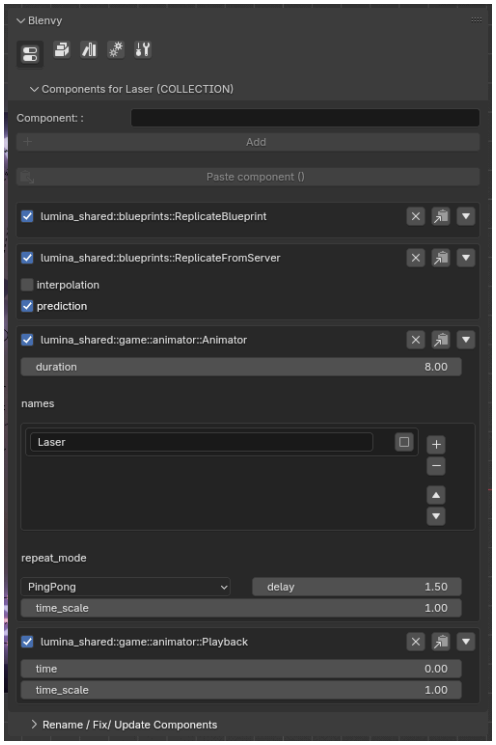


Figure 88: Blenvy

These components are required to be registered in the Rust soruce code as well.

```rust
impl Plugin for TypeRegistryPlugin {
    fn build(&self, app: &mut App) {
        app
            .register_type::<NoRadiance>()
            .register_type::<SpaceshipType>()
            // And many more..
    }
}
```

Figure 89: Register components

Because Blenvy utilizes the Bevy's reflection system to insert these components on load, the registered component type must also derive the `Reflect` trait with the `Component` trait reflected.

```rust
#[derive(Component, Reflect, Clone, Copy)]
#[reflect(Component)] // Reflection is required for components to be registable.
pub struct NoRadiance;
```

Figure 90: Deriving reflection

When the assets are needed to be used in the game, we load them into the game through a unique `enum` type that we create in the source code. The `strum` crate is used here to define the folder the asset is located in.

```rust
#[derive(Component, Reflect, AsRefStr, Debug, Clone, Copy)]
#[reflect(Component)]
#[strum(prefix = "spaceship_blueprints/")] // The folder the asset is located in.
pub enum SpaceshipType {
    Assassin, // Must match the asset file's name.
    Defender,
}

// Spawning using a normal Bevy system.
fn spawn_assassin(mut commands: Commands) {
    commands.spawn((SpaceshipType::Assassin.info(), SpawnBlueprint));
}
```

Figure 91: Spawning assets

4.4.4. Networking Protocols

The Lightyear crate is used as the networking solution for the game. All networking protocols are defined explicitly in the source code.

```
impl Plugin for ProtocolPlugin {
    fn build(&self, app: &mut App) {
        // Channels
        app.add_channel::<OrdReliableChannel>(ChannelSettings {
            mode: ChannelMode::OrderedReliable(ReliableSettings::default()),
            ..default()
        });

        // Messages
        // ============================
        app.register_message::<EnterSandbox>(ChannelDirection::Bidirectional);
        app.register_message::<Matchmake>(ChannelDirection::ClientToServer);
        // And many more..

        // ============================
        // Input
        app.add_plugins(LeafwingInputPlugin::<PlayerAction>::default());

        // Components
        // ============================
        // Only sync once when it was first added.
        app.register_component::<Transform>(ChannelDirection::ServerToClient)
            .add_prediction(ComponentSyncMode::Once);
        // Simple update sync.
        app.register_component::<MaxHealth>(ChannelDirection::ServerToClient)
            .add_prediction(ComponentSyncMode::Simple);
        // Full sync with rollback.
        app.register_component::<Position>(ChannelDirection::ServerToClient)
            .add_prediction(ComponentSyncMode::Full)
            .add_correction_fn(position::lerp);
        // And many more..
    }
}
```

Figure 92: Network protocols

The game uses channel to send important messages that needs to be ordered and reliable. Lightyear also has a strong support for the Leafwing input plugin that Lumina uses. Lastly, syncing of each entity's behaviour is done through the registration of components. For each protocol, developers can define the direction of the message. This allows for server authoritative operations where the message direction can only go from server to clients.

4.4.5. Running the game

To compile Lumina, you have to perform a recursive clone:

```
git clone --recursive https://github.com/nixon-voxell/lumina.git
```

Figure 93: Cloning the repository

Before running the game, the `assets` folder needs to be linked correctly to all the binary crates. You can do so by running:

```
create_asset_junctions.bat
```

Figure 94: Linking the assets

To run the game, you need to start the server and the client. You can do so manually using:

```
cargo run --bin lumina_server
cargo run --bin lumina_client
```

Figure 95: Running the binaries individually

For development purposes, a shell script has been created to speed things up:

```
# Windows
run.bat x
# Linux or MacOS
run.sh x
```

Figure 96: Running the shell script

With `x` being the number of clients you want to spawn.

4.5. Summary

In summary, Velyst lets developers build interactive, vector-based interfaces using Typst, rendered with Vello and powered by Bevy. It supports hot-reloading, function macros, auto-layout, and GPU rendering.

Velyst loads Typst files, compiles them into layouts, and turns them into `VelloScene`. UI elements become Bevy entities, making interaction easy through ECS. Procedural macros help bind Rust code to Typst components.

Lumina is a 2D PvP game where players collect Lumina to power a Tesseract. It features smooth online gameplay with custom physics and global illumination via Radiance Cascades. Players choose between two ships: the stealthy Assassin or the tanky Defender. The game includes four maps for different gameplay modes.

Together, Velyst and Lumina demonstrate a streamlined UI pipeline and its use in a live game setting.

CHAPTER 5: Result and Discussion

6. References

Add styles to UXML. Unity. Retrieved November 13, 2024, from https://docs.unity3d.com/Manual/UIE-add-style-to-uxml.html

Alvin, O. (2020). Rendering Resolution Independent Fonts in Games and 3D-Applications. *LU-CS-EX*.

Anderson, C. (2020, August 10). *Introducing Bevy 0.1*. https://bevyengine.org/news/introducing-bevy/

Anderson, C. (2024, August 10). *Bevy's Fourth Birthday*. https://bevyengine.org/news/bevys-fourth-birthday/

Batra, V., Kilgard, M. J., Kumar, H., & Lorach, T. (2015). Accelerating vector graphics rendering using the graphics hardware pipeline. *ACM Transactions on Graphics (TOG)*, *34*(4), 1–15.

Carlier, A., Danelljan, M., Alahi, A., & Timofte, R. (2020). Deepsvg: A hierarchical generative network for vector graphics animation. *Advances in Neural Information Processing Systems*, *33*, 16351–16361.

Comparison of UI systems in Unity. Unity. Retrieved November 13, 2024, from https://docs.unity3d.com/Manual/UI-system-compare.html

Crow, T. S. (2004). Evolution of the Graphical Processing Unit. *University of Nevada*.

Dalstein, B., Ronfard, R., & Van De Panne, M. (2015). Vector graphics animation with time-varying topology. *ACM Transactions on Graphics (TOG)*, *34*(4), 1–12.

DesLauriers, M. (2015, March 9). *Drawing Lines is Hard*. https://mattdesl.svbtle.com/drawing-lines-is-hard

DEWI, I. G. A. A. O. (2021). Understanding data collection methods in qualitative research: the perspective of interpretive accounting research. *Journal of Tourism Economics and Policy*, *1*(1), 23–34.

Filimowicz, M. (2023, January 4). *History of Video Games*. https://medium.com/understanding-games/history-of-video-games-9465b2eec44c

Gagnon, C. (2019, October 1). *UMG Best Practices*. https://www.unrealengine.com/en-US/tech-blog/umg-best-practices

Gaming Resolution and Hardware choices. (2023, September 6). IronClad Gaming. https://www.ironcladpcs.com/post/gaming-resolution-and-hardware-choices

Ganacim, F., Lima, R. S., De Figueiredo, L. H., & Nehab, D. (2014). Massively-parallel vector graphics. *ACM Transactions on Graphics (TOG)*, *33*(6), 1–14.

Glazkov, D. (2021, November 25). *Retained and immediate mode*. https://glazkov.com/2021/11/25/retained-and-immediate-mode/

Hana, R. (2023, March 28). *Getting started with egui in Rust*. https://whoisryosuke.com/blog

HeySERP-Team. (2023, June 9). *What Are the Advantages and Disadvantages of a WYSIWYG Editors*. HeySERP. https://heyserp.com/blog/wysiwyg-editors/

Higinbotham, W. (1958). *Tennis for two*.

Hoyer, A. W. *SVG + CSS =* ⭐. Retrieved November 12, 2024, from https://andrew.wang-hoyer.com/experiments/svg-animations/

Iché, T. (2016, November 28). *Free VFX image sequences and flipbooks*. https://unity.com/blog/engine-platform/free-vfx-image-sequences-flipbooks

Introduction to UXML. Unity. Retrieved November 13, 2024, from https://docs.unity3d.com/Manual/UIE-WritingUXMLTemplate.html

Jayathilaka, C. (2020, July 30). *Agile Methodology*. https://medium.com/@chathmini96/agile-methodology-30ec4cdf3fc

Jeremias, P., & Quilez, I. (2014). Shadertoy: Learn to create everything in a fragment shader. In *SIGGRAPH Asia 2014 Courses: SIGGRAPH Asia 2014 Courses* (pp. 1–15).

Johnson, B. (2024, February 28). *Simplifying Godot UI Development for All Levels*. https://gamedevartisan.com/tutorials/understanding-godot-ui-control-nodes

Juviler, J. (2024, April 3). *What Is GUI? Graphical User Interfaces, Explained*. HubSpot. https://blog.hubspot.com/website/what-is-gui

Kashivskyy, A. (2024, October 7). *Imperative vs. Declarative Programming - Pros and Cons*. https://www.rootstrap.com/blog/imperative-v-declarative-ui-design-is-declarative-programming-the-future

Kilgard, M. J., & Bolz, J. (2012). GPU-accelerated path rendering. *ACM Transactions on Graphics (TOG)*, *31*(6), 1–10.

Kok, B. (2021). Custom Editor with UI Toolkit. In *Beginning Unity Editor Scripting: Create and Publish Your Game Tools* (pp. 71–110). Apress. https://doi.org/10.1007/978-1-4842-7167-4_4

Kokojima, Y., Sugita, K., Saito, T., & Takemoto, T. (2006). Resolution independent rendering of deformable vector objects using graphics hardware. In *ACM SIGGRAPH 2006 Sketches: ACM SIGGRAPH 2006 Sketches* (p. 118–es).

Krogh-Jacobsen, T. (2023, November 27). *New UI toolkit demos for programmers and artists: Unity blog*. Unity. https://unity.com/blog/engine-platform/new-ui-toolkit-demos-for-programmers-artists

Levien, R. (2018, May 8). *Entity-Component-System architecture for UI in Rust*. https://raphlinus.github.io/personal/2018/05/08/ecs-ui.html

Levien, R. (2020, December 10). *The piet-gpu vision*. https://github.com/linebender/vello/blob/98192612d9f4b7aed9b3223680527473ec449ee9/doc/vision.md

Li, R., Hou, Q., & Zhou, K. (2016). Efficient GPU path rendering using scanline rasterization. *ACM Transactions on Graphics (TOG)*, *35*(6), 1–12.

Linebender. (2024, July 3). Linebender. https://linebender.org/

Loop, C., & Blinn, J. (2005). Resolution independent curve rendering using programmable graphics hardware. In *ACM SIGGRAPH 2005 Papers: ACM SIGGRAPH 2005 Papers* (pp. 1000–1009).

Mateja, D., Armbruster, R., Baumert, J., Bleil, T., Langenbahn, J., Schwedhelm, J. C., Sester, S., & Heinzl, A. (2023). AnimateSVG: autonomous creation and aesthetics evaluation of scalable vector graphics animations for the case of brand logos. *Proceedings of the AAAI Conference on Artificial Intelligence*, *37*(13), 15710–15716.

Mädje, L. (2022). A Programmable Markup Language for Typesetting. *Technical University of Berlin*, 1–77.

Nehab, D., & Hoppe, H. (2008). Random-access rendering of general vector graphics. *ACM Transactions on Graphics (TOG)*, *27*(5), 1–10.

Ratermanis, A. (2017, December 29). *Vector vs raster: What's best for your logo*. https://www.ratermanis.com/blog/2017/12/28/vector-vs-raster

Ray, N., Cavin, X., & Lévy, B. (2005). Vector Texture Maps on the GPU. *Inst. ALICE (Algorithms, Comput., Geometry Image Dept. INRIA Nancy Grand-Est/loria), Tech. Rep. ALICE-TR-05-003*.

Ricardo, A. (2020, September 23). *Imperative v Declarative UI Design - Is Declarative Programming the future?*. https://www.rootstrap.com/blog/imperative-v-declarative-ui-design-is-declarative-programming-the-future

Rick, D. W., McAllister, K. S., & Ruggill, J. E. (2024). Vector Graphics. In *Encyclopedia of Computer Graphics and Games: Encyclopedia of Computer Graphics and Games* (pp. 1967–1970). Springer.

Roadmap for 2023. Linebender. Retrieved November 14, 2024, from https://github.com/linebender/vello/blob/98192612d9f4b7aed9b3223680527473ec449ee9/doc/roadmap_2023.md

Rodney. (2024, July). *Trying egui: building a Cistercian Clock with Rust GUI*. https://rodneylab.com/trying-egui/

Rueda, A., De Miras, J. R., & Feito, F. R. (2008). GPU-based rendering of curved polygons using simplicial coverings. *Computers & Graphics*, *32*(5), 581–588.

Santos, R. (2023, May 25). *Unreal Engine 5 UI Tutorial*. https://www.kodeco.com/38238361-unreal-engine-5-ui-tutorial

Satran, M., & Radich, Q. (2019, August 24). *Retained Mode Versus Immediate Mode*. Microsoft Ignite. https://learn.microsoft.com/en-us/windows/win32/learnwin32/retained-mode-versus-immediate-mode

Sheldon, R. (2022, November). *Display*. https://www.techtarget.com/whatis/definition/display

Sigireddyviswesh. (2023, August 27). *Difference between Markup and Markdown language*. https://medium.com/@sigireddyviswesh/difference-between-markup-and-markdown-language-e087bfff06d2

Siwiec, A. (2023, March 21). *Week 6: The Egui Rust Framework*. https://siwiec.us/blog/week-6-the-egui-rust-framework/

Stanford, D. (2024, January 11). *Celebrating the Vectrex and Vector Graphics in Gaming*. https://www.gloo.digital/blog/celebrating-the-vectrex-and-vector-graphics-in-gaming

Theme Style Sheet (TSS). Unity. Retrieved November 13, 2024, from https://docs.unity3d.com/Manual/UIE-tss.html

Tian, X., & Günther, T. (2022). A survey of smooth vector graphics: Recent advances in representation, creation, rasterization and image vectorization. *IEEE Transactions on Visualization and Computer Graphics*.

Verdi, S. (2023, August 30). *Why Rust is the most admired language among developers*. https://github.blog/developer-skills/programming-languages-and-frameworks/why-rust-is-the-most-admired-language-among-developers/

Waseem, A., & Hart, R. (2023, March 2). *Waterfall Methodology: History, Principles, Stages & More*. https://management.org/waterfall-methodology

wgpu. wgpu. Retrieved November 14, 2024, from https://wgpu.rs/

7. Appendices

7.1. PPF

DRAFT PROJECT PROPOSAL FORM

Proposal ID	:	
Supervisor	:	
Student Name	:	**Cheng Yi Heng**
Student No	:	**TP058994**
Email Address	:	**tp058994@mail.epu.edu.my**
Programme Name	:	**Computer Games Development (CGD)**
Title of Project	:	**Real-time Global Illumination and Dynamic Compute-Centric Vector Graphics in Games**

Figure 97: Ethics Form 1

Contents

Figure 98: PPF 2

1. Introduction

Achieving visually rich and interactive content in real-time without compromising performance is a key aspect of immersive gameplay. This project addresses two major challenges in modern game development: creating dynamic, interactive user experiences and implementing accurate, real-time lighting models. Tackling these challenges requires three key innovations: a compute-centric vector graphics renderer, a programmable approach for developing interactive content, and a performant global illumination technique.

Vector graphics is a form of computer graphics where visual images are generated from geometric shapes, such as points, lines, curves, and polygons, defined on a Cartesian plane. Vector graphics are often used in situations where scalability and precision are essential. Common applications include: logos, typography, diagrams, charts, motion graphics, etc. Examples of softwares that generates or uses vector graphics content includes Adobe Illustrator, Adobe After Effects, Affinity Publisher, Graphite, and many more. Vector graphics is also used in a wide range of file formats including Scalable Vector Graphics (SVG), Portable Document Format (PDF), TrueType Font (TTF), OpenType Font (OTF), etc. However, these formats are rarely used in the game industry directly (they are often preprocessed into some other formats, i.e. triangulation or signed distance fields [SDF]), as game engines are often only tailored towards rendering triangles and bitmap textures instead of paths and curves that we see in the vector graphics formats.

Markup languages *(i.e. Hypertext Markup Language [HTML], Extensible Markup Language [XML])* and style sheets *(i.e. Cascading Style Sheets [CSS])* has dominated the way developers layout and style contents. Over the years, technologies like Unity UI Toolkit has evolved in the game industry to adopt the same pattern but with a user friendly editor, allowing users to layout content using a drag and drop manner while styling their content using sliders, color pickers, and input fields (Jacobsen, 2023). While this improves the user experience of content creation, it lacks the capability of integrating logic and custom contents right inside the user interfaces. These features are often delegated to the programmer which can lead to unintended miscommunications.

Calculating indirect lighting is extremely computationally expensive, as it requires simulating how light bounces off surfaces and interacts with the environment. Ray tracing is an algorithm that calculates these light interactions by tracing lights from the camera into the scene, following their paths as they bounce off surfaces and interact with materials. Each bounce contributes to the final color and lighting of the scene, accounting for reflections, refractions, and scattering.

Figure 99: PPF 3

Unfortunately, ray tracing is too slow for real-time applications, like games. New techniques like light probes and light baking has been employed to approximate global illumination in modern game engines. However, the major issue still exists for these techniques — scalability to larger and more complex scenes.

Figure 100: PPF 4

2. Problem Statement

2.1. Vector Graphics

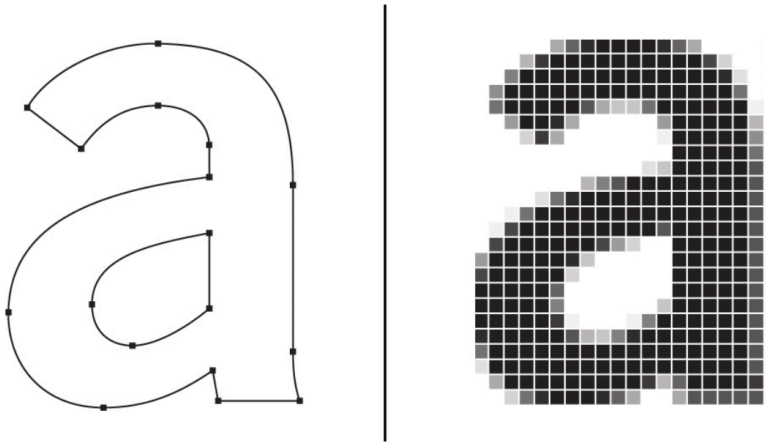


Figure 1: Vector vs Bitmap graphics (Ratermanis, 2017)

Traditional methods of rendering 2D graphics has always relied on bitmap-based texture mapping (Ray et al., 2005). While this approach is ubiquitous, it suffers a major drawback of the *pixelation* effect when being scaled beyond the original resolution (Nehab & Hoppe, 2008). Furthermore, creating animations using bitmap graphics can be extremely limited and complex because of the rigid grid-like data structure used to store the data. Animating bitmap graphics are commonly done through the use of shaders which directly manipulates the individual pixels, or relying on image sequences (flipbooks) which produces an illusion of movement.

Unlike raster graphics, which rely on a fixed grid of pixels, vector graphics are resolution-independent. This means that it can scale without losing quality (shown in Figure 1). A vector illustration is composed of multiple *paths* that define *shapes* to be painted in a given order (Ganacim et al., 2014). Each of these individual paths can be traced, altered, or even morphed into a completely different shape which allows for a huge variety of animation techniques.

Lastly, it is crucial to recognize that while vector graphics offer numerous benefits, it is only suitable for representing precise shapes — such as fonts, logos, and icons. In contrast, complex images with intricate details, like photographs of a cat are far better represented using bitmap formats.

Figure 101: PPF 5

2.2. Interactive UI/UX

Most game engines in the market like Unity, Godot, Game Maker, and Unreal Engine uses a WYSIWYG *(What You See Is What You Get)* editor for creating user interfaces. WYSIWYG editors are visual centric tools that let users work directly within the presentation form of the content (Mädje, 2022). Users normally layout their contents using a drag and drop editor and then style them using a style-sheet. To bind interactions or animations towards a content, users would need to label it with a unique tag and query them through code.

Complex content and logic wouldn't be possible through a typical WYSIWYG editor. For instance, it is virtually impossible to author a custom polygon shape in the editor with custom math based animation based on a time value. This can only be achieved through code, and is often limited by the application programming interface (API) layer provided by the WYSIWYG editor. This creates a huge distinction between the game logic and the visual representation that is needed to convey the messages.

While hot-reloading is applicable for the layout and styling (and simple logic to some extend) of a content. A WYSIWYG editor would not be capable of hot-reloading complex logic as these can only be achieved using code, which in most cases, requires a re-compilation. This could lead to frustration and lost of creativity due to the slow feedback loop.

In summary, WYSIWYG editors are great for prototyping but suffers from complex animations and interactions.

2.3. Global Illumination

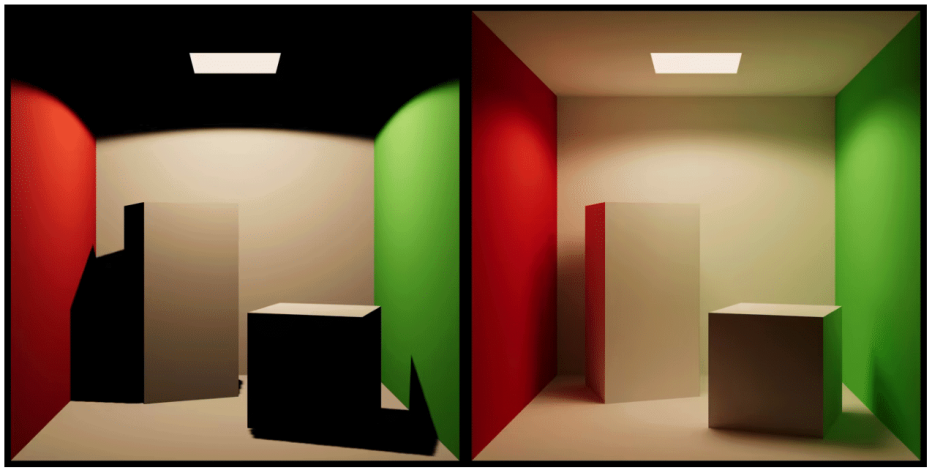


Figure 2: Local illumination vs global illumination (Jalnionis, 2022)

Figure 102: PPF 6

Global illumination has been a notoriously hard problem to solve in computer graphics. To put things into perspective, global illumination intends to solve the *many to many* interactions between light, obstacles, and volumes. In real-time game engines like Unity and Unreal Engine, light probes *(a.k.a radiance probes)* are placed around the scene to capture lighting information into a cube map, which can be applied to nearby objects. To smoothen out the transition between probes, objects interpolate between various nearby probes weighted by distance to approximate the global radiance.

Manual placement of probes leads to questions like “how many probes should a scene have?” or “how much probes is a good approximation?”. It ultimately becomes a trade-off between fidelity and performance with more probes resulting in better approximation, while fewer probes improve performance. This paradoxical issue raises the challenge of finding the optimal balance. This dilemma underscores the need for smarter, adaptive techniques, ensuring both visual fidelity and efficiency.

Figure 103: PPF 7

3. Project Aim and Objectives

In this project, we aim to empower creators to create rich and visually appealing content in games in an efficient and streamlined workflow. This allows the creator to have the luxury of focusing most of their time on quality content rather than the technical details needed to achieve the look or feel that they envisioned.

The objectives of this project are:

1. To utilize Vello, a compute-centric vector graphics renderer for rendering animated and dynamic vector graphics content.
2. To create an intuitive and yet powerful (programmable) workflow for generating animated and dynamic content.
3. To streamline the collaboration between UI/UX developer and gameplay programmers
4. To allow creators to focus on the creative aspects of game development.
5. To implement Radiance Cascades, a technique that provides realistic lighting without sacrificing real-time performance.

Figure 104: PPF 8

4. Literature Review

4.1. Vector Graphics

Scanline rendering is the process of shooting rays from one side of the screen to the other while coloring pixels in between based on collision checkings with paths in between. A GPU based scanline rasterization method is proposed by parallelizing over *boundary fragments* while bulk processing non-boundary fragments as horizontal spans (Li et al., 2016). This method allows fully animated vector graphics to be rendered in interactive frame rates.

Apart from scanline rasterization, tesselation method can also be used to convert vector graphics into triangles and then pushed to the GPU for hardware accelerated rasterization. Loop & Blinn (2005) further improved this method by removing the need of approximating curve segments into lines. Instead, each curve segments is evalulated in a *fragment shader* which can be calculated on the GPU. This allows for extreme zoom levels without sacrificing qualities.

Re-tesselation of vector graphics can be computationally expensive, especially when it's inherently a serial algorithm that often needs to be solved on the CPU. Kokojima et al. (2006) combines the work of Loop & Blinn (2005) with the usage of GPU's stencil buffer by using *triangle fans* to skip the tesselation process. This approch, however, does not extend to cubic Bézier segments as they might not be convex. Rueda et al. (2008) addressed this issue by implementing a fragment shader that evaluates the implicit equation of the Bézier curve to discard the pixels that fall outside it. The two-step "Stencil then Cover" (StC) method builds upon all of these work and unified path rendering with OpenGL's shading pipeline — `NV_path_rendering` (Kilgard & Bolz, 2012). This library was further improved upon by adding support for transparency groups, patterns, gradients, more color spaces, etc. (Batra et al., 2015). It was eventually integrated into Adobe Illustrator.

4.2. Interactive UI/UX

Beneath all graphical interfaces lies the underlying code that structure and renders the visual elements. The two most notable approach towards creating user interface frameworks are immediate-mode graphical user interface (IMGUI) and retained-mode graphical user interface (RMGUI). Some popular IMGUI frameworks includes Dear IMGUI and egui (*Dear Imgui*, n.d.; *Egui*, n.d.), while some popular RMGUI frameworks includes Xilem (*Xilem*, n.d.). Although powerful, these UI frameworks strongly relies on hardcoded programming.

Enter the web technologies. Modern browsers typically render UI elements using markup languages like HTML and SVG for structuring the content and style-sheets like CSS for styling

Figure 105: PPF 9

them. The use of markup structures allows developers to fully separate their UI layout from the codebase, simplifying the identification and management of UI components. With style sheets, developers can create, share, and reuse templates, enhancing consistency and streamlining the design process throughout the application. Notable frameworks that utilizes this model includes Unity UI Toolkit, React, Vue, etc (Jacobsen, 2023; *React*, n.d.; *Vue*, n.d.).

Markup languages also give rise to many WYSIWYG editors. These editors let users perform drag and drop actions to layout UI for quick prototyping as each components can now be represented using only markup syntax (no code required).

A major limitation of simple markup languages like HTML is that structure changes can only be achieved through code. For example, if you want a form to disappear after button press, you would need to alter the HTML via code. Typst offers an alternative towards this problem by introducing programming capabilities into markdown.

Typst is a competitor of LaTeX, designed to simplify the typesetting process with a modern and intuitive approach. Unlike its predecessors, Typst can directly embed logic. Using the previous example, developers would only need to pass in a boolean value and Typst will automatically exclude the form from being in the layout at all. This currently works only in theory, as Typst is primarily a document generator without a user-friendly interface for modifying defined variables.

This is where our project comes in, we aim to provide this interface through Velyst, which couple Typst with Vello for rendering dynamic and programmable content in real-time.

4.3. Global Illumination

Ray tracing is the de-facto standard for calculating light bounces which contributes to global illumination. Clever methods like backwards ray tracing has been introduced to speed up the algorithm, but still, it is no where near real-time frame rates (Arvo & others, 1986). Light baking is introduced to solve this issue, however, it lacks the ability to adapt to runtime scene changes.

Recent studies has shown great results of utilizing neural networks for approximating global illumination (Choi et al., 2024). However, neural network based methods tend to suffer from unpredictability as the output is highly basd upon the input training data, making it unreliable.

Recent works by McGuire et al. (2017) places light field probes around the scene to encode lighting information from static objects and sample them in real-time. Dynamic diffuse global

Figure 106: PPF 10

illumination (DDGI) further improves this technique by allowing light field probes to update dynamically based on scene changes (Majercik et al., 2019).

Radiance cascades improves upon this technique by using a hierarchical structure to place light probes (Osborne & Sannikov, 2024). This technique is based upon the *penumbra condition*, where closer distance require low angular resolution and high spatial resolution while further distance require high angular resolution and low spatial resolution.

Figure 107: PPF 11

5. Deliverables

This project introduces **Velyst**, an innovative approach towards generating interactive content. It utilizes **Vello**, a compute-centric vector graphics renderer (*Vello*, n.d.), and **Typst**, a programmable markup language for typesetting (Mädje, 2022). Velyst provides an extremely streamlined workflow that allows both UI/UX developers and gameplay programmers to easily collaborate in a project. The following are the deliverables that will be implemented in the Velyst library:

1. Allows the user to create interactive content that responds to user inputs in real-time.
2. Allows the user to perform hot-reloading during the development phase.
3. Allows the user to synchronize components with in-app states dynamically.
4. Allows the user to embed logic directly inside Typst scripts for rendering complex scenes.

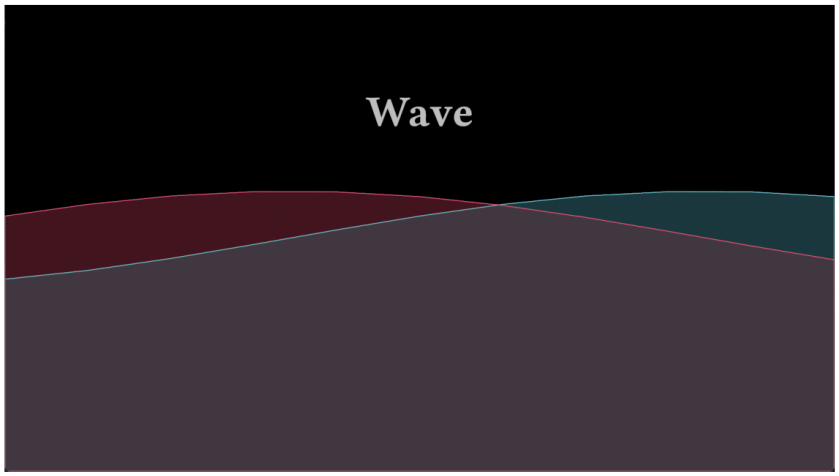


Figure 3: Velyst wave (custom shape + animation) demo

Figure 108: PPF 12

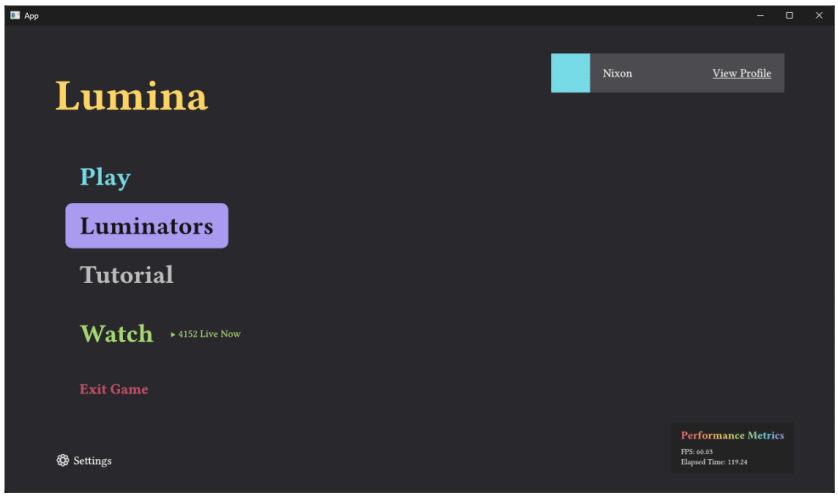


Figure 4: Velyst user interface demo

A library will also be created that utilizes the **Radiance Cascades** technique for rendering real-time 2D global illumination in the Bevy game engine. The following are the deliverables for the Radiance Cascades library:

1. Allows the user to implement global illumination using Radiance Cascades.
2. Allows the user to create adaptive lighting that responds to scene changes in real-time.
3. Allows the user to use arbritrary emmisive shapes to illuminate the scene in game.

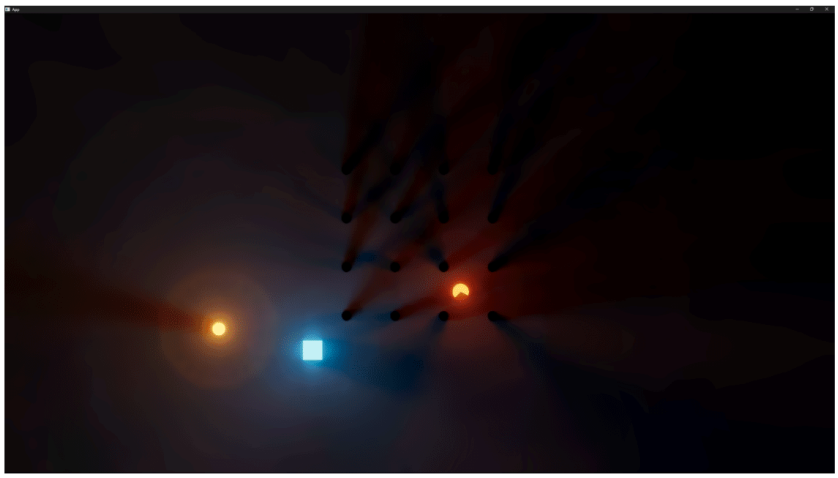


Figure 5: Radiance cascades demo

Figure 109: PPF 13

6. References

Arvo, J., & others. (1986). Backward ray tracing. *Developments in Ray Tracing, Computer Graphics, Proc. Of ACM SIGGRAPH 86 Course Notes*, 259–263.

Batra, V., Kilgard, M. J., Kumar, H., & Lorach, T. (2015). Accelerating vector graphics rendering using the graphics hardware pipeline. *ACM Transactions on Graphics (TOG)*, *34*(4), 1–15.

Choi, E., Carpentier, V., Shin, S., & Yoo, S. (2024). Baking Relightable NeRF for Real-time Direct/Indirect Illumination Rendering. *Arxiv Preprint Arxiv:2409.10327*.

Dear ImGui. Retrieved September 22, 2024, from https://github.com/ocornut/imgui

Egui. Retrieved September 22, 2024, from https://github.com/emilk/egui

Ganacim, F., Lima, R. S., De Figueiredo, L. H., & Nehab, D. (2014). Massively-parallel vector graphics. *ACM Transactions on Graphics (TOG)*, *33*(6), 1–14.

Jacobsen, T. K. (2023, November). *New UI toolkit demos for programmers and artists: Unity blog*. Unity. https://unity.com/blog/engine-platform/new-ui-toolkit-demos-for-programmers-artists

Jalnionis, K. (2022, September). Unity. https://unity.com/blog/engine-platform/5-common-lightmapping-problems-and-tips-to-help-you-fix-them

Kilgard, M. J., & Bolz, J. (2012). GPU-accelerated path rendering. *ACM Transactions on Graphics (TOG)*, *31*(6), 1–10.

Kokojima, Y., Sugita, K., Saito, T., & Takemoto, T. (2006). Resolution independent rendering of deformable vector objects using graphics hardware. In *ACM SIGGRAPH 2006 Sketches* (p. 118–es).

Li, R., Hou, Q., & Zhou, K. (2016). Efficient GPU path rendering using scanline rasterization. *ACM Transactions on Graphics (TOG)*, *35*(6), 1–12.

Loop, C., & Blinn, J. (2005). Resolution independent curve rendering using programmable graphics hardware. In *ACM SIGGRAPH 2005 Papers* (pp. 1000–1009).

Majercik, Z., Guertin, J.-P., Nowrouzezahrai, D., & McGuire, M. (2019). Dynamic diffuse global illumination with ray-traced irradiance fields. *Journal of Computer Graphics Techniques*, *8*(2).

McGuire, M., Mara, M., Nowrouzezahrai, D., & Luebke, D. (2017). Real-time global illumination using precomputed light field probes. *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3d Graphics and Games*. https://doi.org/10.1145/3023368.3023378

Figure 110: PPF 14

Mädje, L. (2022). A Programmable Markup Language for Typesetting. *Technical University of Berlin*, 1–77.

Nehab, D., & Hoppe, H. (2008). Random-access rendering of general vector graphics. *ACM Transactions on Graphics (TOG)*, *27*(5), 1–10.

Osborne, C. M., & Sannikov, A. (2024). Radiance Cascades: A Novel High-Resolution Formal Solution for Multidimensional Non-LTE Radiative Transfer. *Arxiv Preprint Arxiv:2408.14425*, 1–21.

Ratermanis, A. (2017, December 29). *Vector vs raster: What's best for your logo*. https://www.ratermanis.com/blog/2017/12/28/vector-vs-raster

Ray, N., Cavin, X., & Lévy, B. (2005). Vector texture maps on the GPU. *Inst. ALICE (Algorithms, Comput., Geometry Image Dept. INRIA Nancy Grand-Est/loria), Tech. Rep. ALICE-TR-05-003*.

React. Retrieved September 22, 2024, from https://react.dev/

Rueda, A., De Miras, J. R., & Feito, F. R. (2008). GPU-based rendering of curved polygons using simplicial coverings. *Computers & Graphics*, *32*(5), 581–588.

Vello. Retrieved September 21, 2024, from https://github.com/linebender/vello

Vue. Retrieved September 22, 2024, from https://vuejs.org/

Xilem. Retrieved September 22, 2024, from https://github.com/linebender/xilem

Figure 111: PPF 15

7.2. Ethics Form

Office Record	Receipt – Fast-Track Ethical Approval
Date Received:	Student name: Cheng Yi Heng Student number: TP058994 Received by:
Received by whom:	Date:

APU / APIIT FAST-TRACK ETHICAL APPROVAL FORM (STUDENTS)

Tick one box (level of study):
- ☐ POSTGRADUATE (PhD / MPhil / Masters)
- ☐✓ UNDERGRADUATE (Bachelors degree)
- ☐ FOUNDATION / DIPLOMA / Other categories

Tick one box (purpose of approval):
- ☐ ✓Thesis / Dissertation / FYP project
- ☐ Module assignment
- ☐ Other: _____

Title of Programme on which enrolled … … Bsc.Hons MMT(ARVR)…

Tick one box: ☐✓ Full-Time Study or ☐ Part-Time Study

Title of project / assignment: Establishing a Workflow for Real-time Global Illumination and Vector Graphics in Enhancing Dynamic Game Environments

Name of student researcher: Cheng Yi Heng

Name of supervisor / lecturer: Mr. Jacob Sow

Student Researchers- please note that certain professional organisations have ethical guidelines that you may need to consult when completing this form.

Supervisors/Module Lecturers - please seek guidance from the Chair of the APU Research Ethics Committee if you are uncertain about any ethical issue arising from this application.

		YES	NO	N/A
1	Will you describe the main procedures to participants in advance, so that they are informed about what to expect?	✓		
2	Will you tell participants that their participation is voluntary?	✓		
3	Will you obtain written consent for participation?	✓		
4	If the research is observational, will you ask participants for their consent to being observed?	✓		
5	Will you tell participants that they may withdraw from the research at any time and for any reason?	✓		
6	With questionnaires and interviews will you give participants the option of omitting questions they do not want to answer?	✓		
7	Will you tell participants that their data will be treated with full confidentiality and that, if published, it will not be identifiable as theirs?	✓		
8	Will you give participants the opportunity to be debriefed i.e. to find out more about the study and its results?	✓		

If you have ticked **No** to any of Q1-8 you should complete the full Ethics Approval Form.

		YES	NO	N/A
9	Will your project/assignment deliberately mislead participants in any way?		✓	
10	Is there any realistic risk of any participants experiencing either physical or psychological distress or discomfort?		✓	
11	Is the nature of the research such that contentious or sensitive issues might be involved?		✓	

If you have ticked **Yes** to 9, 10 or 11 you should complete the full Ethics Approval Form. In relation to question 10 this should include details of what you will tell participants to do if they should experience any problems (e.g. who they can contact for help). You may also need to consider risk assessment issues.

Figure 112: Ethics Form 1

			YES	NO	N/A
12	Does your project/assignment involve work with animals?			✓	
13	Do participants fall into any of the following special groups? **Note that you may also need to obtain satisfactory clearance from the relevant authorities**	Children (under 18 years of age)			
		People with communication or learning difficulties			
		Patients			
		People in custody		✓	
		People who could be regarded as vulnerable			
		People engaged in illegal activities (eg drug taking)			
14	Does the project/assignment involve external funding or external collaboration where the funding body or external collaborative partner requires the University to provide evidence that the project/assignment had been subject to ethical scrutiny?			✓	

If you have ticked **Yes** to 12, 13 or 14 you should complete the full Ethics Approval Form. There is an obligation on student and supervisor to bring to the attention of the APU Research Ethics Committee any issues with ethical implications not clearly covered by the above checklist.

STUDENT RESEARCHER
Provide in the boxes below (plus any other appended details) information required in support of your application, THEN SIGN THE FORM.

	Please Tick Boxes
I consider that this project/assignment has no significant ethical implications requiring a full ethics submission to the APU Research Ethics Committee.	✓
Give a brief description of participants and procedure (methods, tests used etc) in up to 150 words. *This survey targets professionals, hobbyists, and students with experience or interest in game development and interactive media. The aim is to assess the perceived importance and current state of **global illumination** and **vector graphics** in enhancing visual quality and immersion in real-time applications. Participants will complete a questionnaire evaluating the role of GI and VG in their projects and the extent to which popular game engines, such as Unity and Unreal Engine, support these technologies.* *The survey uses a mix of Likert-scale questions and multiple-choice options to gauge familiarity, preferences, and perceived challenges in implementing GI and VG. Responses will help identify trends in the industry and areas for improvement in engine support for realistic lighting and scalable graphics. Data will be collected anonymously, and responses will be analyzed to support research into advancing visual fidelity in real-time interactive applications.*	
I also confirm that: i) All key documents e.g. consent form, information sheet, questionnaire/interview are appended to this application. Or	✓
ii) Any key documents e.g. consent form, information sheet, questionnaire/interview schedules which need to be finalised following initial investigations will be submitted for approval by the project/assignment supervisor/module lecturer before they are used in primary data collection.	✓

Figure 113: Ethics Form 2

E-signature: Cheng Yi Heng
Print Name: Cheng Yi Heng
Date: 1/11/2024
(Student Researcher)

Please note that any variation to that contained within this document that in any way affects ethical issues of the stated research requires the appending of new ethical details. New ethical consent may need to be sought.
The completed form (and any attachments) should be submitted for consideration by your Supervisor/Module Lecturer

SUPERVISOR/MODULE LECTURER
PLEASE CONFIRM THE FOLLOWING:

	Please Tick Box
I consider that this project/assignment has no significant ethical implications requiring a full ethics submission to the APU Research Ethics Committee	✓
i) I have checked and approved the key documents required for this proposal (e.g. consent form, information sheet, questionnaire, interview schedule) Or	✓
ii) I have checked and approved draft documents required for this proposal which provide a basis for the preliminary investigations which will inform the main research study. I have informed the student researcher that finalised and additional documents (e.g. consent form, information sheet, questionnaire, interview schedule) must be submitted for approval by me before they are used for primary data collection.	

SUPERVISOR AND SECOND ACADEMIC SIGNATORY

STATEMENT OF ETHICAL APPROVAL (please delete as appropriate)

1) **THIS PROJECT/ASSIGNMENT HAS BEEN CONSIDERED USING AGREED APU/SU PROCEDURES AND IS NOW APPROVED**

2) **THIS PROJECT/ASSIGNMENT HAS BEEN APPROVED IN PRINCIPLE AS INVOLVING NO SIGNIFICANT ETHICAL IMPLICATIONS, BUT FINAL APPROVAL FOR DATA COLLECTION IS SUBJECT TO THE SUBMISSION OF KEY DOCUMENTS FOR APPROVAL BY SUPERVISOR (see Appendix A)**

for APU use only
E-signature… Jacob … … … … … … … … … Print Name… Jacob Sow … … … … … … … … … Date… 1-11-2024 …
(Supervisor/Lecturer)

E-signature… … … … … … … … … … … Print Name… … … … … … … … … … … Date… … … …

Figure 114: Ethics Form 3

(Second Academic Signatory)

Figure 115: Ethics Form 4

Office Record Date Received: Received by whom:	Receipt – Appendix A (Fast-Track Ethics Form) Student name: Student number: Received by: Date:

APPENDIX A
AUTHORISATION FOR USE OF KEY DOCUMENTS

Completion of Appendix A is required when for good reasons key documents are not available when a fast track application is approved by the supervisor/module lecturer and second academic signatory.

I have now checked and approved all the key documents associated with this proposal e.g. consent form, information sheet, questionnaire, interview schedule

Title of project/assignment… …

… ....

Name of student researcher ….

Student ID: ……………………………… Intake: … … … … … … … … … … … … … … … …

E-signature… … … … … … … … … … … Print Name… … … … … … … … … … … Date… … …….. …
(Supervisor/Lecturer)

Fast-Track Ethical Approval Form ver. 3.0 (Dec 2015) Page 5 of 5

Figure 116: Ethics Form 5

Office Record	*Receipt*
Date Received: Received by whom:	Student name: Cheng Yi Heng Student number: TP058994 Received by: Date:

ACADEMIC RESEARCH ETHICS DISCLAIMER

Declaration about ethical issues and implications of research project/assignment proposals to be included on project/assignment application forms

Project/Assignment Title:

Establishing a Workflow for Real-time Global Illumination and Vector Graphics in Enhancing Dynamic Game Environments

The following declaration should be made in cases where research project/assignment applicants for a particular project/assignment and the supervisor(s)/lecturer(s) for that project/assignment conclude that it is not necessary to apply for ethical approval for the research project/assignment.

We confirm that the University's guidelines for ethical approval have been consulted and that all ethical issues and implications in relation to the above project/assignment have been considered. We confirm that ethical approval need not be sought.

Cheng Yi Heng	Cheng Yi Heng	1/11/2024
Name of Research Project/Assignment Applicant	e-signature	Date
Jacob Sow	for APU use only Jacob	1-11-2024
Name of Research Project Supervisor/ Assignment Lecturer	e-signature	Date

Figure 117: Disclaimer Form

7.3. Log Sheets

(APU: Serial Number)

PLS V1.0

Project Log Sheet – Supervisory Session

Notes on use of the project log sheet:

1. This log sheet is designed for meetings of more than 15 minutes duration, of which there must be at minimum SIX (6) during the course of the project (SIX mandatory supervisory sessions).
2. The student should prepare for the supervisory sessions by deciding which question(s) he or she needs to ask the supervisor and what progress has been made (if any) since the last session, and noting these in the relevant sections of the form, effectively forming an agenda for the session.
3. A log sheet is to be brought by the STUDENT to each supervisory session.
4. The actions by the student (and, perhaps the supervisor), which should be carried out before the next session should be noted briefly in the relevant section of the form.
5. The student should leave a copy (after the session) of the Project Log Sheet with the supervisor and to the administrator at the academic counter. A copy is retained by the student to be filed in the project file.
6. It is recommended that students bring along log sheets of previous meetings together with the project file during each supervisory session.
7. The log sheet is an important deliverable for the project and an important record of a student's organisation and learning experience. The student **must** hand in the log sheets as an appendix of the final year documentation, with sheets dated and numbered consecutively.

Student's name: Cheng Yi Heng **Date: 29 Oct 2024** **Meeting No: #1**

Project title: Establishing a Framework for Interactive and Dynamic Vector Graphics Content in Real-Time Video Games
Intake: APU3F2408CGD

Supervisor's name: Mr. Jacob Sow Tian You **Supervisor's signature:** .....

Items for discussion (noted by student before mandatory supervisory meeting):

1. **Showcase state of current project.**
2. **Discuss about the game design including:**
   - **Game Mechanics**
   - **Game Loop**
   - **Game Art**
   - **Game Tech**
   - **Game Lore**
3. **Research topic selection.**
4. **Research aim, scope, objectives, methodologies.**

Record of discussion (noted by student during mandatory supervisory meeting):

1. **Needs a better documentation on the game – Game Design Document.**
2. **Game loop needs to be well defined.**
3. **Methodologies needs to fit the research objectives. Examples are given, questionnaires and interviews.**

Action List (to be attempted or completed by student by the next mandatory supervisory meeting):

1. **To prepare a game design document with all of the topics listed above.**
2. **To define the aim, objectives and problem backgrounds of the project and research.**
3. **To determine a scope for the project.**

Note: A student should make an appointment to meet his or her supervisor (via the consultation system) at least ONE (1) week prior to a mandatory supervisor session – please see document on project timelines. In the event a supervisor could not be booked for consultation, the project manager should be informed ONE (1) week prior to the session so that a meeting can be subsequently arranged.

Project Log Sheet

Figure 118: Project log sheet 1

(APU: Serial Number)

PLS V1.0

Project Log Sheet – Supervisory Session

Notes on use of the project log sheet:

1. This log sheet is designed for meetings of more than 15 minutes duration, of which there must be at minimum SIX (6) during the course of the project (SIX mandatory supervisory sessions).
2. The student should prepare for the supervisory sessions by deciding which question(s) he or she needs to ask the supervisor and what progress has been made (if any) since the last session, and noting these in the relevant sections of the form, effectively forming an agenda for the session.
3. A log sheet is to be brought by the STUDENT to each supervisory session.
4. The actions by the student (and, perhaps the supervisor), which should be carried out before the next session should be noted briefly in the relevant section of the form.
5. The student should leave a copy (after the session) of the Project Log Sheet with the supervisor and to the administrator at the academic counter. A copy is retained by the student to be filed in the project file.
6. It is recommended that students bring along log sheets of previous meetings together with the project file during each supervisory session.
7. The log sheet is an important deliverable for the project and an important record of a student's organisation and learning experience. The student **must** hand in the log sheets as an appendix of the final year documentation, with sheets dated and numbered consecutively.

Student's name: Cheng Yi Heng **Date: 8 Nov 2024** **Meeting No: #2**

Project title: Establishing a Framework for Interactive and Dynamic Vector Graphics Content in Real-Time Video Games
Intake: APU3F2408CGD

Supervisor's name: Mr. Jacob Sow Tian You **Supervisor's signature:** .....Jac.....

Items for discussion (noted by student before mandatory supervisory meeting):
1. Discuss about the Game Design Document.
2. Reconfirm about the project topic/title.
3. Go through the IR and discuss about the introduction section which includes:

- Abstract
- Problem Background
- Problem Statement
- Aim
- Objectives
- Questions
- Scope
- Potential Benefit

4. Discuss about next steps (literature review).

Record of discussion (noted by student during mandatory supervisory meeting):
1. Citation needs to be done accurately, following the APA 7 style in both reference page and in-text.
2. Citation needs to be added to the problem statement section.
3. Abstract must include expected results.
4. Abstract methodology section must include: what, how, and audience.
5. Introduction of literature review needs to go from history, to now. Basically, like a flow of "storytelling".

Project Log Sheet

Figure 119: Project log sheet 2

Action List (to be attempted or completed by student by the next mandatory supervisory meeting):
1. To perform edits on the Introduction as discussed
2. To double check the citations and references.
3. To write the literature review section.
4. To come up with questions for survey questionnaires and interviews.
5. To complete the methodology chapter.

Note: A student should make an appointment to meet his or her supervisor (via the consultation system) at least ONE (1) week prior to a mandatory supervisor session – please see document on project timelines. In the event a supervisor could not be booked for consultation, the project manager should be informed ONE (1) week prior to the session so that a meeting can be subsequently arranged.

Figure 120: Project log sheet 3

(APU: Serial Number)

PLS V1.0

Project Log Sheet – Supervisory Session

Notes on use of the project log sheet:

1. This log sheet is designed for meetings of more than 15 minutes duration, of which there must be at minimum SIX (6) during the course of the project (SIX mandatory supervisory sessions).
2. The student should prepare for the supervisory sessions by deciding which question(s) he or she needs to ask the supervisor and what progress has been made (if any) since the last session, and noting these in the relevant sections of the form, effectively forming an agenda for the session.
3. A log sheet is to be brought by the STUDENT to each supervisory session.
4. The actions by the student (and, perhaps the supervisor), which should be carried out before the next session should be noted briefly in the relevant section of the form.
5. The student should leave a copy (after the session) of the Project Log Sheet with the supervisor and to the administrator at the academic counter. A copy is retained by the student to be filed in the project file.
6. It is recommended that students bring along log sheets of previous meetings together with the project file during each supervisory session.
7. The log sheet is an important deliverable for the project and an important record of a student's organisation and learning experience. The student **must** hand in the log sheets as an appendix of the final year documentation, with sheets dated and numbered consecutively.

Student's name: Cheng Yi Heng **Date: 15 Nov 2024** **Meeting No: #3**

Project title: Establishing a Framework for Interactive and Dynamic Vector Graphics Content in Real-Time Video Games
Intake: APU3F2408CGD

Supervisor's name: Mr. Jacob Sow Tian You **Supervisor's signature:** ……

Items for discussion (noted by student before mandatory supervisory meeting):
1. Go through the IR and discuss about the literature review section.
2. Refine the content of literature review.
3. Discuss about the software development methodology.
4. Discuss about the data collection methodology (questionnaire and interview questions).
5. Discuss about potential interview candidates.

Record of discussion (noted by student during mandatory supervisory meeting):
1. Gaming and games do not hold the same meaning.
2. "Code" change to "Figure".
3. In text citation for 2 authors must replace "&" with "and".
4. Interviewees should be professionals (subject matter experts) from the industry.
5. Overall add more history and introductory elements to the literature review.

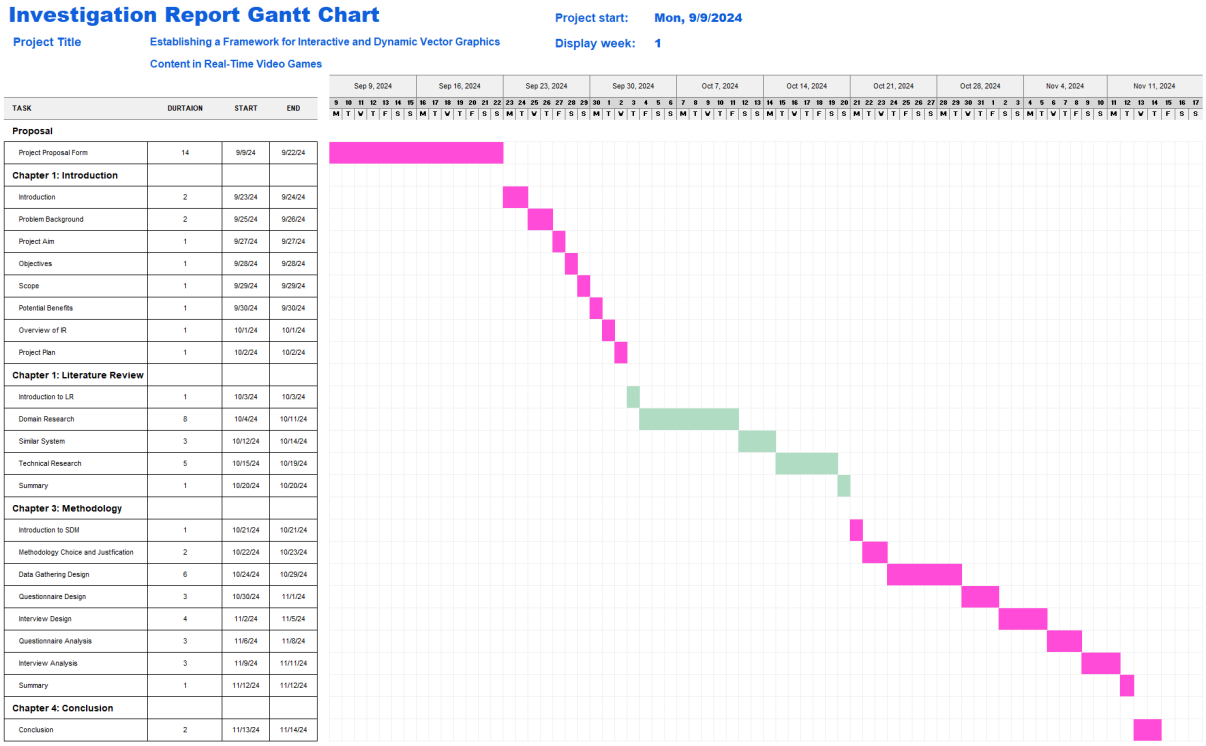
Action List (to be attempted or completed by student by the next mandatory supervisory meeting):
1. To add more history and introductory elements to the literature review
2. To look for interviewees.
3. To add introductory sections to each domain research.
4. To find participants to complete the survey.
5. To hold the interviews and collect data.
6. To analyse the interview answers.

Note: A student should make an appointment to meet his or her supervisor (via the consultation system) at least ONE (1) week prior to a mandatory supervisor session – please see document on project timelines. In the event a supervisor could not be booked for consultation, the project manager should be informed ONE (1) week prior to the session so that a meeting can be subsequently arranged.

Project Log Sheet

Figure 121: Project log sheet 4

7.4. Gantt Chart



Investigation Report Gantt Chart

Project Title: Establishing a Framework for Interactive and Dynamic Vector Graphics Content in Real-Time Video Games

Project start: Mon, 9/9/2024

Display week: 1

TASK	DURTAION	START	END
Proposal			
Project Proposal Form	14	9/9/24	9/22/24
Chapter 1: Introduction			
Introduction	2	9/23/24	9/24/24
Problem Background	2	9/25/24	9/26/24
Project Aim	1	9/27/24	9/27/24
Objectives	1	9/28/24	9/28/24
Scope	1	9/29/24	9/29/24
Potential Benefits	1	9/30/24	9/30/24
Overview of IR	1	10/1/24	10/1/24
Project Plan	1	10/2/24	10/2/24
Chapter 1: Literature Review			
Introduction to LR	1	10/3/24	10/3/24
Domain Research	8	10/4/24	10/11/24
Similar System	3	10/12/24	10/14/24
Technical Research	5	10/15/24	10/19/24
Summary	1	10/20/24	10/20/24
Chapter 3: Methodology			
Introduction to SDM	1	10/21/24	10/21/24
Methodology Choice and Justification	2	10/22/24	10/23/24
Data Gathering Design	6	10/24/24	10/29/24
Questionnaire Design	3	10/30/24	11/1/24
Interview Design	4	11/2/24	11/5/24
Questionnaire Analysis	3	11/6/24	11/8/24
Interview Analysis	3	11/9/24	11/11/24
Summary	1	11/12/24	11/12/24
Chapter 4: Conclusion			
Conclusion	2	11/13/24	11/14/24

Figure 122: Gantt Chart

7.5. Respondence Demographic Profile

Name	Gender	Age
Batman	Male	18 - 24
Jediah	Male	18 - 24
leelien	Female	18 - 24
Mohab	Male	18 - 24
Zi Qing	Female	18 - 24
vw	Male	25 - 30
Alireza	Male	18 - 24
Kah Boon	Female	18 - 24
Teh Cheng En	Male	18 - 24
Sze	Female	18 - 24
haojin	Female	18 - 24
Karma	Female	18 - 24
MJ	Female	18 - 24
Awacs	Male	18 - 24
e	Male	18 - 24

Name	Gender	Age
Rafaya	Female	18 - 24
Jia Hong	Male	18 - 24
Jeremiah	Male	18 - 24
Tan Jia Hao	Male	18 - 24
Marcus Cheng	Male	18 - 24
Goh Shao Hang	Male	18 - 24
Nick Jang	Male	25 - 30
GQKEK	Male	18 - 24
Loo	Male	18 - 24
Megat Syawaludin	Male	18 - 24
Jing Heng	Male	18 - 24
Jun Yong	Male	18 - 24
ShiJie	Male	18 - 24
Ming Yu	Male	18 - 24
Xiu Zhen	Male	18 - 24
Diviyan	Male	18 - 24
Brenden Tan	Male	18 - 24
roi	Male	18 - 24

Table 8: Respondence Demographic Profile

Note: the name of the participant can be pseudonyms and not the actual name in protecting the confidentiality