



## **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 fulfilment 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**

**2<sup>nd</sup> Marker: Assoc. Prof. Ts. Dr. Tan Chin Ike**

**27<sup>th</sup> November 2024**

# Acknowledgement

## 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. Our study employs purposive sampling to collect valuable insights from developers in the gaming and interactive application sectors. We will conduct three in-depth interviews with industry professionals to gain expert perspectives. Additionally, online survey questionnaires will be distributed to a broader developer audience to capture a wider range of opinions. 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. Our approach contributes to advancing infrastructure and fostering innovation, aligning with the goals of *Sustainable Development Goal (SDG) 9*.

**Keywords** — Typesetting, Markdown, Workflow, Dynamic content, Compute-centric, Typst

# Contents

<b>Acknowledgement .....</b>	<b>2</b>
<b>Abstract .....</b>	<b>3</b>
<b>CHAPTER 1: INTRODUCTION .....</b>	<b>7</b>
1.1. Introduction .....	7
1.2. Problem Statement .....	8
1.2.1. Scalability, precision, and animation .....	8
1.2.2. Lack of support on UI/UX creation for complex interactivity .....	9
1.2.3. Slow iteration time and feedback loops in development .....	9
1.3. Project Aim .....	10
1.4. Research Objectives .....	10
1.5. Research Questions .....	10
1.6. Scope .....	10
1.6.1. Tasks to be executed: .....	10
1.6.2. Constraints .....	11
1.6.3. What will be done in this project: .....	12
1.6.4. What will not be done in this project: .....	12
1.7. Potential Benefit .....	12
1.7.1. Tangible Benefit .....	12
1.7.2. Intangible Benefit .....	13
1.7.3. Target User .....	13
1.8. Overview of the IR .....	13
1.9. Project Plan .....	15
<b>CHAPTER 2: Literature Review .....</b>	<b>16</b>
2.1. Introduction .....	16
2.2. Domain Research .....	17
2.2.1. Rasterization of vector graphics .....	17
2.2.2. Graphical interface content creation .....	18
2.2.3. Summary of key findings .....	21
2.3. Similar Systems/Works .....	22
2.3.1. Unity UI Toolkit .....	22
2.3.2. LaTeX .....	22
2.3.3. React .....	22
2.4. Technical Research .....	22

2.4.1. Bevy .....	22
2.4.2. Vello .....	22
2.4.3. Typst .....	22
<b>CHAPTER 3: Methodology .....</b>	<b>22</b>
3.1. System Development Methodology .....	22
3.2. Data Gathering Design .....	22
3.3. Analysis .....	22
<b>References .....</b>	<b>22</b>
<b>Appendices .....</b>	<b>22</b>
<b>References .....</b>	<b>23</b>

## Figures

Figure 1: Tennis for Two (Higinbotham, 1958) .....	7
Figure 2: Vector vs Bitmap graphics (Ratermanis, 2017) .....	8

## Tables

Table 1: Scope .....	10
Table 2: Constraints .....	11
Table 3: Project Plan .....	15

## Codes

Code 1: Egui code example .....	18
Code 2: Xilem code example .....	18
Code 3: HTML example .....	19
Code 4: CSS example .....	19
Code 5: Typst example .....	19

# **CHAPTER 1: INTRODUCTION**

## **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 1 uses vector graphics to render their game on a repurposed oscilloscope in 1958 (Filimowicz, 2023). 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). These games showcases the potential of vector-based visuals to achieve fluid and interactive animations.



Figure 1: 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 gaming (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 (DesLauriers, 2015) and sign distance field (Alvin, 2020) due to technical constraints.

## 1.2. Problem Statement

### Scalability, precision, and animation

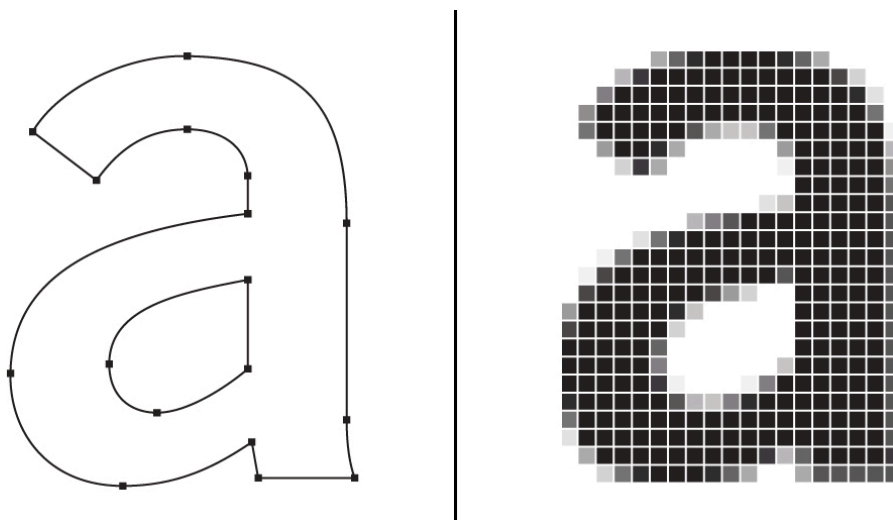


Figure 2: Vector vs Bitmap graphics (Ratemanis, 2017)

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-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 (Jeremias & Quilez, 2014), or relying on image sequences (a.k.a flipbooks) (Iché, 2016) which produces an illusion of movement.

Unlike raster graphics which rely on a fixed grid of pixels, vector graphics are precise and resolution independent. This means that it can scale without losing quality (shown in Figure 2). 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.).

### **Lack of support on UI/UX creation for complex interactivity**

WYSIWYG editors are visual centric tools that let users work directly within the presentation form of the content (Mädje, 2022). Most game engines in the market like Unity (Jacobsen, 2023), Unreal Engine (Santos, 2023), and Godot (Johnson, 2024) uses a *What You See Is What You Get* (WYSIWYG) editor for creating user interfaces (UI) and user experiences (UX). 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 application programming interface (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.

### **Slow iteration time and feedback loops in development**

While hot-reloading is applicable for the layout and styling (and simple logic to some extend) 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 and Unreal Blueprints. 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.

### 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, performance, and player experience 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 and computational requirements 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, performance, and overall player experience 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 and computational requirements associated with rendering dynamic vector graphics in real-time?

### 1.6. Scope

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

<b>Velyst</b>	An interactive Typst content creator using Vello and Bevy.
<b>Lumina</b>	A 2D top down fast paced objective based PvPvE game.

Table 1: Scope

#### Tasks to be executed:

1. Develop the **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.

### Constraints

Constraint	Reason
<b>Compatibility</b>	Our 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 runs on newer versions of browsers that supports WebGPU.
<b>Limited Documentation</b>	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.
<b>Limited Resource</b>	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.
<b>Vector Graphics Constraint</b>	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

### **What will be done in this project:**

1. **Typst compiler:** A custom implementation of Typst compiler will be created 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 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.

### **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:** Our 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.

## **1.7. Potential Benefit**

### **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.

### **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.

### **Target User**

**Velyst** will particularly be targeted towards UI/UX developers, motion graphics creators, and vector graphics enthusiasts. Because **Velyst** 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 loves fast paced multipalyer 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.

Chapter 4 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. The report closes 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	9 <sup>th</sup> of September 2024	22 <sup>nd</sup> of September 2024
Chapter 1: Introduction			
Introduction	2	23 <sup>rd</sup> of September 2024	24 <sup>th</sup> of September 2024
Problem Background	2	25 <sup>th</sup> of September 2024	26 <sup>th</sup> of September 2024
Project Aim	1	27 <sup>th</sup> of September 2024	27 <sup>th</sup> of September 2024
Objectives	1	28 <sup>th</sup> of September 2024	28 <sup>th</sup> of September 2024
Scope	1	29 <sup>th</sup> of September 2024	29 <sup>th</sup> of September 2024
Potential Benefits	1	30 <sup>th</sup> of September 2024	30 <sup>th</sup> of September 2024
Overview of IR	1	1 <sup>st</sup> of October 2024	1 <sup>st</sup> of October 2024
Project Plan	1	2 <sup>nd</sup> of October 2024	2 <sup>nd</sup> of October 2024
Chapter 2: Literature Review			
Introduction to LR	1	3 <sup>rd</sup> of October 2024	3 <sup>rd</sup> of October 2024
Domain Research	8	4 <sup>th</sup> of October 2024	11 <sup>th</sup> of October 2024
Similar System	3	12 <sup>th</sup> of October 2024	14 <sup>th</sup> of October 2024
Technical Research	5	15 <sup>th</sup> of October 2024	19 <sup>th</sup> of October 2024
Summary	1	20 <sup>th</sup> of October 2024	20 <sup>th</sup> of October 2024
Chapter 3: Methodology			
Introduction of SDM	1	21 <sup>st</sup> of October 2024	21 <sup>st</sup> of October 2024
Phases of SDM	2	22 <sup>nd</sup> of October 2024	23 <sup>rd</sup> of October 2024
Data Gathering Design	6	24 <sup>th</sup> of October 2024	29 <sup>th</sup> of October 2024
Research Methodology	2	30 <sup>th</sup> of October 2024	31 <sup>st</sup> of October 2024
Design of Questionnaire	3	1 <sup>st</sup> of November 2024	3 <sup>rd</sup> of November 2024
Design of Interview	3	4 <sup>th</sup> of November 2024	6 <sup>th</sup> of November 2024
Analysis of Questionnaire	3	7 <sup>th</sup> of November 2024	9 <sup>th</sup> of November 2024
Analysis of Interview	3	10 <sup>th</sup> of November 2024	12 <sup>th</sup> of November 2024
Summary	1	13 <sup>th</sup> of November 2024	13 <sup>th</sup> of November 2024
Chapter 4: Conclusion and Reflections			
Conclusion	2	14 <sup>th</sup> of November 2024	15 <sup>th</sup> of November 2024

Table 3: Project Plan

## **CHAPTER 2: Literature Review**

### **2.1. Introduction**

Vector graphics are increasingly valuable in game development due to their scalability and precision, yet their integration into real-time game engines involves distinct technical challenges. Unlike raster images, which degrade when resized, vector graphics are resolution-independent, making them ideal for various screen sizes and resolutions. This advantage, however, comes at the cost of more complex rendering requirements, as vector graphics must undergo a rasterization phase to translate their shapes into pixels for display. The demands of real-time rendering make this process particularly intricate, as maintaining both high visual quality and performance can be challenging.

To address the performance demands of rasterization in game engines, various rendering techniques have been developed to streamline this process. Efficient rasterization methods help achieve the fast rendering speeds required for interactive applications, while ensuring that image quality remains high. Techniques focused on converting vector shapes into compatible formats for real-time processing have allowed developers to create more responsive and visually detailed graphics without a loss in performance. These approaches improve frame rates and visual fidelity, making vector graphics a practical choice in game settings where performance is essential.

In addition to advances in rasterization, the tools used for creating and managing graphical content have also evolved. Modern UI frameworks play a foundational role in constructing interactive experiences, providing developers with ways to structure and style visual elements effectively. The shift towards markup languages and programmable typesetting has made it easier for teams to collaborate and iterate on designs, fostering a more flexible workflow. This evolution has supported the creation of more dynamic visual components that can be reused and customized, enhancing user engagement and experience.

This review provides a foundation for analyzing the technical aspects of vector graphic rasterization and content creation tools, highlighting how these technologies can drive efficiency, scalability, and visual impact in game design and interactive media.

## 2.2. Domain Research

### Rasterization of vector graphics

Vector graphics are often used in situations where scalability and precision are essential. This property comes with a cost. As mentioned in the previous chapter, rendering vector graphics in today's era requires a rasterization phase. Solving for this phase is non-trivial as it is often required to compute a partial differential equation (PDE) (Tian & Günther, 2022). 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, tessellation 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 evaluated in a fragment shader which can be calculated on the GPU. This allows for extreme zoom levels without sacrificing qualities.

Re-tessellation 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 tessellation process. 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.

## Graphical interface content creation

Beneath all graphical interfaces lies the underlying code that structures 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 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 hardcoded programming as shown in Code 1 and Code 2.

```
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"));
```

Code 1: Egui code example

```
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))
}
```

Code 2: Xilem code example

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 Code 3 and Code 4. 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.

```
<!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>
```

Code 3: HTML example

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

Code 4: 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 (Jacobsen, 2023).

A major limitation of simple markup languages like HTML is that structural changes can only be achieved through code. 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 markdown (Mädje, 2022).

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

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

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

Code 5: Typst example

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 (Code 5). 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.

## **Summary of key findings**

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 the gaming space.

## **2.3. Similar Systems/Works**

Unity UI Toolkit

LaTeX

React

## **2.4. Technical Research**

Bevy

Vello

Typst

## **CHAPTER 3: Methodology**

### **3.1. System Development Methodology**

### **3.2. Data Gathering Design**

### **3.3. Analysis**

## **References**

## **Appendices**

## **References**

- Alvin, O. (2020). Rendering Resolution Independent Fonts in Games and 3D-Applications. *LU-CS-EX*.
- 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.
- Battlezone*. (1980). Atari.
- Carrier, 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.
- 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>
- 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>
- 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.
- 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>

- 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>
- 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>
- 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: 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: 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*.
- 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.
- 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>
- Space Wars*. (1977). Cinematronics.
- 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>
- Tac/Scan*. (1982). Sega.
- 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*.