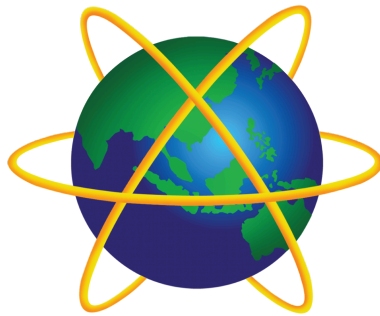


DRAFT PROJECT PROPOSAL FORM



A . P . U

**ASIA PACIFIC UNIVERSITY
OF TECHNOLOGY & INNOVATION**

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

Contents

1. Introduction	3
2. Problem Statement	5
2.1. Vector Graphics	5
2.2. Interactive UI/UX	6
2.3. Global Illumination	6
3. Project Aim and Objectives	8
4. Literature Review	9
4.1. Vector Graphics	9
4.2. Interactive UI/UX	9
4.3. Global Illumination	10
5. Deliverables	12
6. References	14

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.

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.

2. Problem Statement

2.1. Vector Graphics

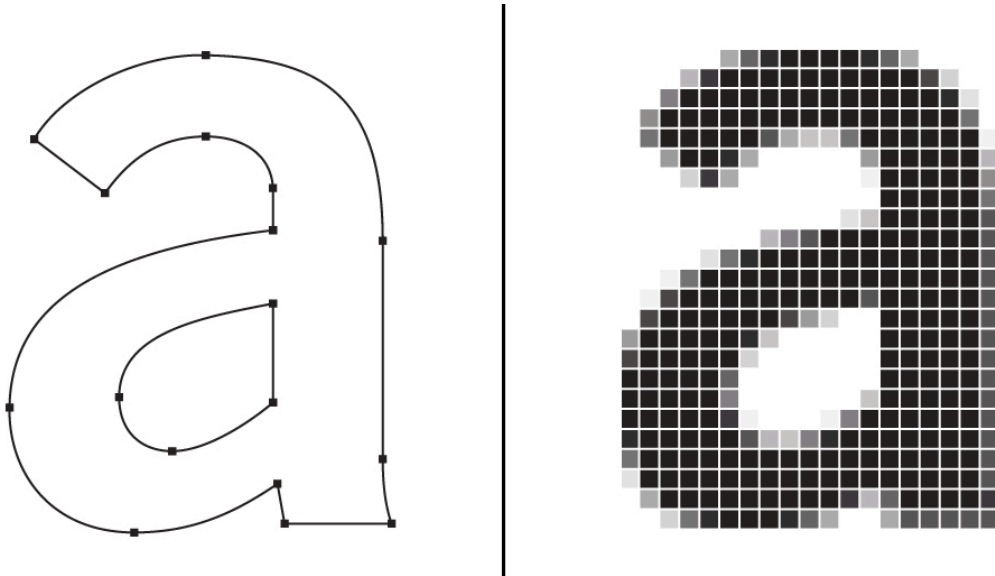


Figure 1: Vector vs Bitmap graphics (Ratnermanis, 2017)

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.

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

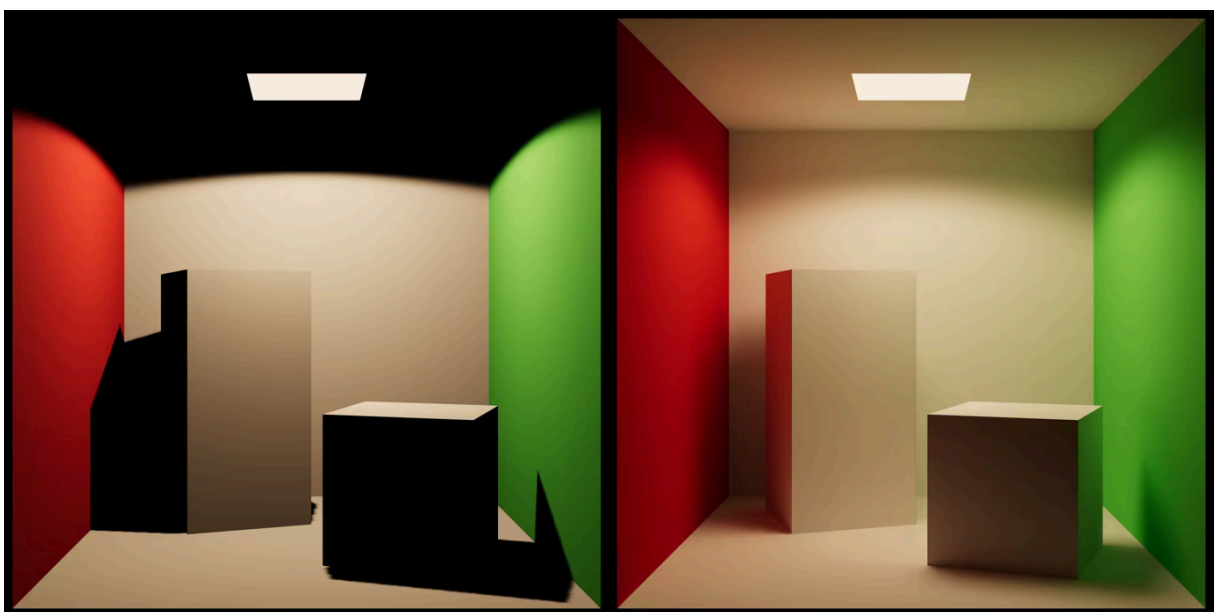


Figure 2: Local illumination vs global illumination (Jalnionis, 2022)

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.

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.

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, 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.

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

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 utilize this model include 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 component 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 couples 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 have been introduced to speed up the algorithm, but still, it is nowhere 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 have 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 based upon the input training data, making it unreliable.

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

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.

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.

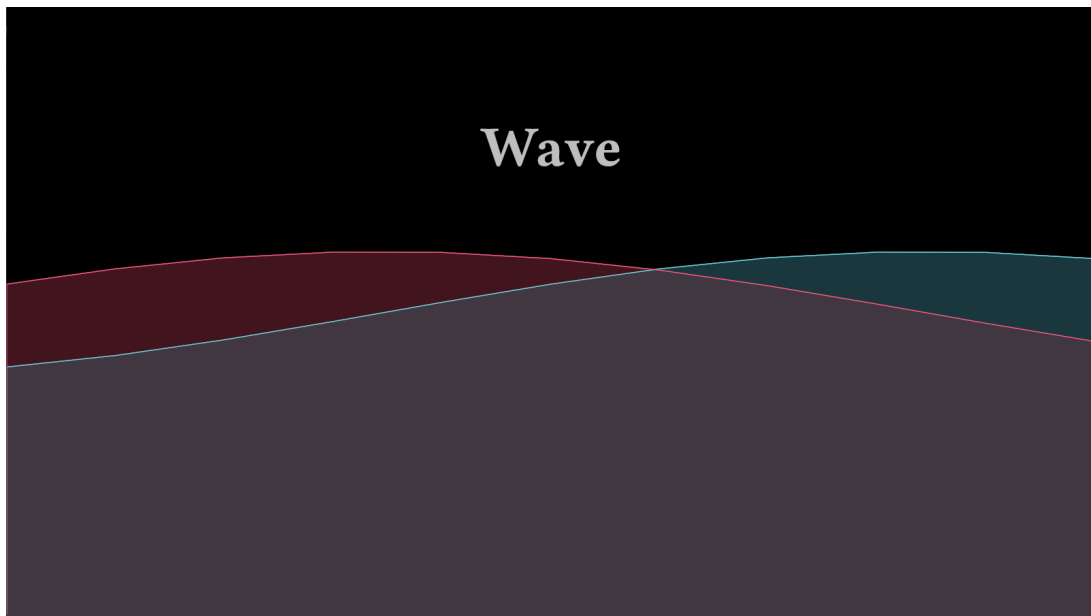


Figure 3: Velyst wave (custom shape + animation) demo

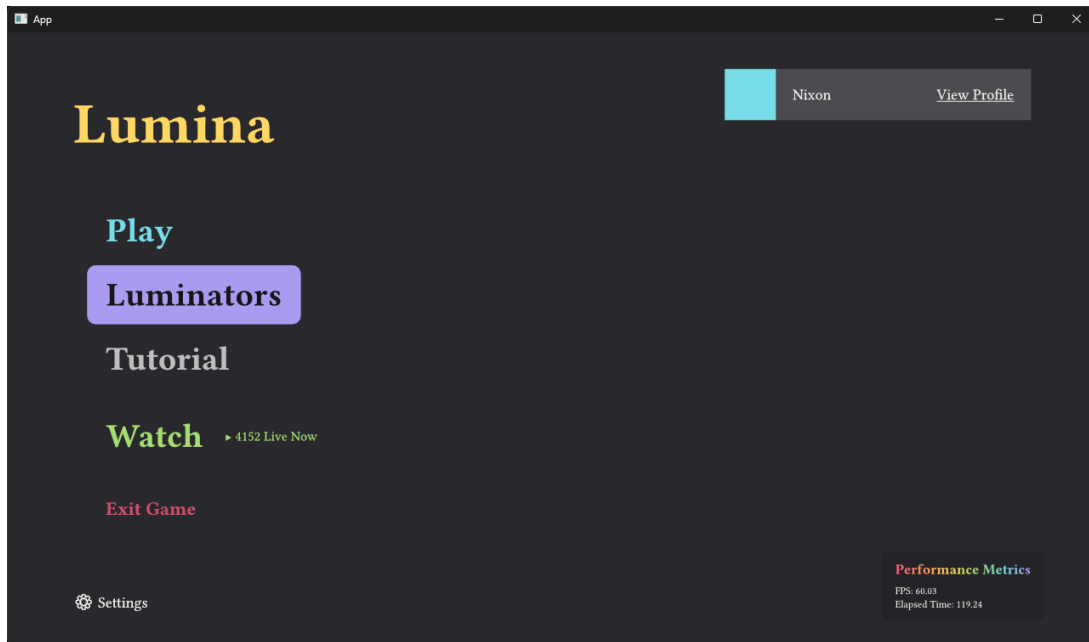


Figure 4: Velyst user interface demo

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 arbitrary emmissive shapes to illuminate the scene in game.

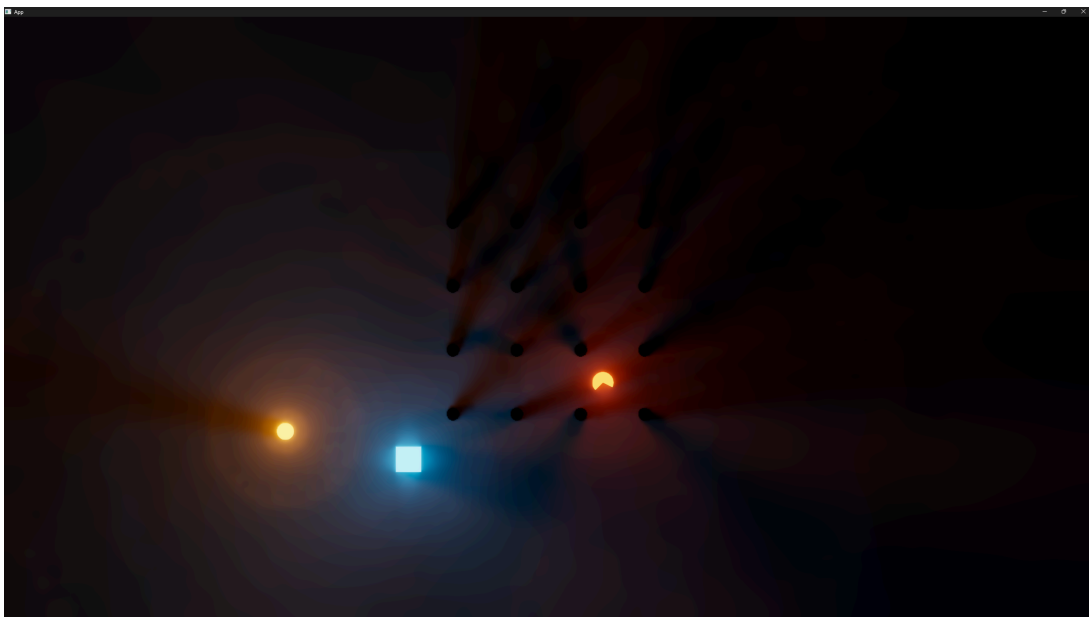


Figure 5: Radiance cascades demo

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>

- 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>