Tool creation for an in-house engine: How does one create a shader/post-processing tool for use in education?

Dannielle G. Smith

BSc (Hons) Computer Games Applications Development, 2025

School of Design and Informatics
Abertay University

Table of Contents

Table of Figures iv
Table of Tables v
Acknowledgements vi
Abstract vii
Abbreviations, Symbols and Notation viii
Chapter 1 Introduction 1
1.1. Research Question 1
1.2. Project overview 1
1.3. Project Aims 1
1.4. Scope 1
1.4.1. Objectives: 1
1.4.2. What It Does Not Include: 2
1.4.3. Scope Changes: 3
1.5 Hypothesis 3
Chapter 2 Literature Review 4
2.1. Educational Use 4
2.2. Shaders 6
2.1. The Basics of Shaders 6
2.2. Post-Processing Shaders 8
2.3. Shader Libraries 9
2.3. Render Pipelines 10
2.3.1. Renderers and the Rendering Pipeline 10
2.3.2. Structure of the Rendering Pipeline 10
2.3.3. Advanced and Modular Pipelines 11
2.2.4. Educational and Practical Implications 11
2.5. Tool Design 11
2.5.2. Usability and Accessibility in Game Development Tools 12
2.5.3. Tools as Cultural and Creative Enablers 13

2.5.4. Historical Perspectives: From CAD to Collaborative Design .. 13
2.5.6. Prototyping 14
2.7. Conclusion - Tying everything together 15
Chapter 3 Methodology 15
3.1. Planning 16
3.1.1. Main project pieces 17
3.1.2. Research and Development 18
3.1.3. Tool Architecture 18
3.1.4. Implementation Phases 19
3.1.5. Documentation and Tutorials 19
3.2. Implementing 19
3.2.1. DirectX11 Prototype 19
3.2.2. Transferring Prototype to Skateboard / DirectX12 22
3.3. Testing 25
3.5. Documentation 25
Chapter 4 Results & Discussion 26
4.1 Introduction 26
4.2 Pre-Tool – Initial Understanding of Concepts 27
4.3 Post-Tool - Outcomes and Learning Improvements 27
4.4 Tool Usability and User Experience 28
4.5 Interpretation of Findings 28
4.6 Comparison with Existing Research 29
4.7 Implications for Education and Tool Design 29
Educational Practices 29
Tool Design Considerations 29
4.8 Limitations and Challenges 30
Tool Limitations 30
Methodological Limitations 30

4.9 Conclusion 30
Chapter 5 Conclusion and Future Work 31
6.0. Conclusion 31
6.1. Summary of Key Findings 31
6.2. Implications for Educational Practices 31
6.3. Limitations of the Study 32
6.4. Final Reflection 33
6.0. Future Work 31
6.1. Improving the Educational Tool 31
6.2. Expanding the Scope of the Study 31
6.3. Exploring New Educational Approaches 31
6.4. Further Research Directions 31
6.5. Conclusion of Future Work 31
List of References 37
Bibliography 39
Appendices 40

Table of Figures

Table of Tables

Acknowledgements

I would like to express my sincere gratitude to Mr. Naman Merchant, my primary supervisor, for his invaluable guidance, support, and constructive feedback throughout this project. His expertise and encouragement have been instrumental in shaping the direction and execution of this research.

My thanks also go to Erin Hughes, my secondary supervisor, for her additional support and insights.

I would like to acknowledge Einar Bruverisv, a member of the Skateboard development team, whose advice and technical insights were highly beneficial throughout the project. Special thanks are due to the creators of the Skateboard Engine, whose work provided the game engine used as a foundation for this research, and to the developers of IMGUI, whose open-source contributions supported the creation and functionality of the educational tool.

Finally, I would like to extend my appreciation to all those who volunteered their time for testing and feedback. Their participation in tool evaluation and data collection was vital to the success of this study.

Abstract

//work in progress

This paper outlines the development of a user-friendly shader tool designed to assist developers, with a focus on those new to development, in creating immersive environments using shaders and post-processing effects/techniques.

As the gaming industry places a greater emphasis on game visuals the development of more advanced shaders and post-processing techniques are naturally produced. The complexity of shader programming can be a major obstacle for newcomers.

This tool will attempt to streamline developers' workflows by providing a user-friendly interface for shader creation and integration, as well as a library of pre-built shaders, allowing users to play around with various visual effects without needing extensive graphics coding skills.

This paper will go over the tool's creation, it uses in an educational environment and its efficiency as both a tool for development and a way of learning.

Abbreviations, Symbols and Notation

If required

GUI - Graphical user interface

Skateboard -

Tool – in this context the tool/tools that are discussed are specifically graphics related tools, like unreal engines material shader.

Chapter 1 Introduction

1.1. Research Question

Question: "Tool creation for an in-house engine: How does one create a shader/post-processing tool for use in education?"

1.2. Project overview

This project aims to develop a shader tool in the form of a library designed to help both developers and new programmers explore and effectively utilize post-processing effects. By providing a collection of pre-built shaders that can be easily applied and customized, the tool will allow users to dive into the world of graphics programming with minimal coding knowledge. The tool will also feature an intuitive visual editor, complete with a preview mode for real-time feedback, ensuring an engaging and accessible learning experience for users at all skill levels.

In addition to providing a user-friendly interface, the project will include a step-by-step guide to help users navigate the in-house engine tool. This will be coupled with comprehensive documentation that breaks down the concepts behind shader creation and post-processing effects. The goal is to bring greater awareness to the capabilities of graphics-related programming, showcasing its importance and applications in modern development. Furthermore, the plug-in will support education by acting as an interactive learning tool, allowing users to experiment, understand, and gain foundational knowledge in graphics programming.

1.3. Project Aims

Aims:

- To create a shader tool in the form of a library to help developers and new programmers explore and use post-possessing effects effectively.
- To have a step-by-step easy guide to an in-house engine tool that is easy to use and pick up.
- Bring more awareness to the uses of graphics-related programming.
- Find a way that this plug-in could support education on graphics programming.

1.4. Scope

1.4.1. Objectives:

1. **Shader Library Creation:**
    a. Develop a library containing pre-built shaders that developers can easily apply, customize, and integrate into their own project/s.
    b. Provide simplified code/functions/etc to make shader usage more accessible.
2. **Visual Shader Editor (GUI):**
    a. Create intuitive GUI that enables users to create and modify shaders without needing extensive coding knowledge.
3. **Real-Time Preview Feature:**
    a. Implement a preview mode to visualize shader changes in real-time.
    b. Ensure the preview feature is only available in GUI mode and while the program is running.
4. **Comprehensive Code Documentation:**
    a. Offer built-in documentation and a separate PDF guide explaining the "how" and "why" of shader creation.
    b. Include step-by-step tutorials and explanations to guide users at all levels.
5. **Optimized Performance:**
    a. Ensure that the tool operates efficiently without negatively affecting performance.
6. **Beginner-Friendly Design:**
    a. Provide intuitive code and tools that even beginners can easily pick up and use within the first 15 minutes of experimentation.
7. **Educational Value:**
    a. Design the tool as an educational resource, helping users gain a basic understanding of post-processing effects and shader programming.
8. **Accessibility Features for the UI**
    a. **High Contrast Mode and Colour Blindness:** Does the tool offer high-contrast themes or colourblind-friendly modes to ensure visibility for users with different visual needs?
    b. **Customizable Font Sizes/Layouts:** Can the user adjust font sizes or UI scaling for better readability?

1.4.2. What It Does Not Include:

1. **Particle Effects Creation:**
   The initial proposal included particle effects creation, but this has been removed due to the broad scope and complexity of the project.

1.4.3. Scope Changes:

1. The project has evolved from being a tool with educational potential into a full-on educational tool. This includes added documentation, and tutorial features to support learning.
2. Additionally, accessibility features will be incorporated, as this is an educational tool, and many users require such support to ensure inclusivity.

1.5 Hypothesis

Tools tend to help base understanding, but theory will only help so much when applied to programming as it won't teach syntax or give more in-depth knowledge. It's a quick and easy tool for those who don't have the time or full knowledge to create their own shaders and may help with the basics, but if one wants to gain more understanding of the process other more traditional means would be recommended. But overall, a good starting point and stepping stone for getting a more in-depth understanding with practical experience.

Chapter 2 Literature Review

Game development has become more accessible thanks to easy-to-use game development engines like Unity, Unreal, and RPG Maker. Letting a wider variety of game makers - including students and hobbyists- enter the industry. These easily available resources have made game development more accessible and encouraged creative and experimental ventures outside of conventional AAA games. However, while fundamental design and development tools have advanced, allowing for a lower entry barrier to the creation process, there remains a notable gap in easily available educational resources and entry points for more specialized areas—particularly post-processing and shader development. Post-processing effects, such as bloom and colour grading, are critical for shaping a unique visual identity. Shaders and visual effects not only enhance aesthetics but also influence gameplay, narrative, and emotional tone. Despite their importance, tools that teach these skills in an accessible and easy-to-learn way are limited and are made with professional developers in mind, not beginners who are looking to learn. While professional tools like Unreal's Material Editor or Unity's Shader Graph exist, they are often intimidating for beginners because of their high entry barriers and lack of tutorials; though they do have communities that can help fill the gaps, they can only help to a limited extent.

Current teaching methods—lectures, tutorials, and traditional classes—struggle to fully teach the length of post-processing work. Tools that are easy to use and made with education in mind are clearly needed so that students can learn by playing, experimenting, and receiving visual feedback with hands on experience.

This literature review looks at the growth of easily accessible tools for game development, what shaders are, and the importance of post-processing and tools in modern game development, highlighting the lack of educational resources for more specialty skills like shaders and post-processing effects.

It sets the stage for the development of specialised tools designed to bridge the gap for post-processing effects, supporting learners in understanding and creating post-processing effects through guided, hands-on experience.

2.1. Educational Use

Post-processing and shader development play a vital role in modern game design, particularly in shaping the visual aesthetics and emotional tone of interactive experiences. In educational settings—especially within game development and computer graphics courses—these tools provide an essential entry point for teaching advanced concepts such

as rendering pipelines, lighting models, colour theory, and optimization strategies. Students who interact with these systems tend to have a greater understanding of computer graphics in addition to learning the technical principles of underlying visual effects easier.

Teaching shader programming is still a difficult task as beginners find shaders intimating (A. Toisoul, D. Rueckert, B. Kainz, 2017) and often challenging as learning shaders tend to involve abstract mathematics and require familiarity with GPU-specific programming languages and APIs like DirectX, OpenGL and Vulkan. The intricate nature of shaders and visual effects is often difficult to explain using conventional classroom methods like lectures, textbooks, labs, and tutorials. When working with graphic systems, this disparity can limit students' understanding of topics and impede their ability to learn.

In order to solve this, interactive learning settings that provide instant visual feedback are becoming more and more acknowledged as crucial to education, particularly shader education. Tools like Unity's Shader Graph, Unreal Engine's Material Editor, and Godot's Visual Shader Editor allow users to build shaders through node-based interfaces, lowering the learning curve and allowing more determined learners a way to explore shaders and post-processing effects. Although it still requires a basic understanding of shaders, online tools like Shadertoy (*Shadertoy, Inigo Quilez, and Pol Jeremias, 2013*) further increase accessibility by allowing users to experiment with GLSL shaders right in the browser, providing instant visuals to changes in the code.

Despite these developments, most shader creation tools are designed for professionals or people with greater experience, with little consideration for educational uses or beginner-friendly instructional resources. Some tutorial features are included in engines like Unity and Unreal, but they frequently can't keep up with rapid software updates or don't provide the context required for beginners to learn the already complex ins and outs of shaders. Thanks to this, there is a discernible lack of learning material that concentrates only on shaders and post-processing effects.

The visualization subsystem is particularly crucial in this context, as up to 80% of human perception is processed visually, Branislav Sobota (2023). By allowing students to see the immediate impact of shader modifications—whether it's adjusting bloom intensity, changing surface materials, or applying colour grading effects—educational tools can turn abstract theory into tangible experience. This visual interactivity makes concepts like lighting models and fragment shading easier to understand and apply, especially for visual learners.

Game engines also serve as powerful educational platforms as they bridge theory and practice. Their modularity and support for visual scripting, plug-ins, and real-time rendering make them great for a wide range of development and educational means —from casual

createrscreators to professional teams. Tools such as Unreal's Blueprints or Godot's node system show how visual approaches to complex systems can simplify learning while preserving the depth and nuance of development.
While shaders and post-processing are essential components of modern game development and computer graphics, they are inaccessible to the majority because of the high learning curve and the lack of easy-to-understand educational resources on the topic. A platform that could bridge the gap between complex technical content and beginner-friendly learning is something that would play a critical role in improving how these topics are taught. Something that can help students visualize shader logic and post-processing effects in real-time yet "fun" enough to not given into frustration.
In the context of this project, the development of a shader/post-processing educational tool aims to fill the gap for an easy starting point to post-processing effects for beginners by prioritizing accessibility, real-time feedback, and intuitive UI. Unlike traditional learning methods, a tool that will focus on visual experimentation as a core teaching strategy, allowing students to manipulate shaders and post-processing effects in an environment that encourages exploration. This aligns with the principles of gamified learning, where engagement and curiosity drive understanding, and learners gain confidence and experience through active participation.

2.2. Shaders

Modern computer graphics increasingly relies on the use of shaders to improve many aspects of rendering, from unique visuals to effects, as well as to optimized performance and preform calculation for lighting and other such things. This section of the chapter is dedicated to shaders and is divided into three different sections: the basics, their use in post-processing, and the use shader libraries.

2.1. The Basics of Shaders

At the core of modern rendering pipelines are shaders. In computer graphics, shaders are small or "micro" programs that run on the GPU, they determine how images -and more specifically pixels- are drawn to the screen. Think of them as instructions that tell your computer how to colour, light, and transform objects in a 3D or 2D space. Shaders are essential in creating anything graphics related, from realistic lighting and shadows to stylized effects like cartoon outlines, or effects like bloom.

There are five primary shader types: vertex, pixel/fragment, geometry, tessellation, and compute. Each has a distinct function within the graphics pipeline. These are listed below:

Vertex Shaders; The vertex shader calculates each point or "vertex" of a shape, determining its position in 3D space and how it moves. They calculate where the vertex should appear on screen after applying transformations like scaling, rotation, or perspective. For example, if you're animating a wave, the vertex shader helps calculate the ripple effect.

Pixel Shaders, otherwise known as a fragment shader; This shader deals with each pixel that makes up the final image rendered to the screen, deciding its final colour and appearance. This is where effects like lighting, shadows, transparency, reflections, texture mapping and other visual effects are added. If something looks shiny, glowing, or realistically lit, the fragment shader is what runs it.

Geometry Shaders; These come after the vertex shader in the graphic pipeline and can dynamically add, remove, or alter geometry. Geometry shaders work with entire shapes, such as triangles, and can even create new ones, unlike vertex shaders that can only work with individual points. A geometry shader could be used to add grass blades to a surface or to make a trail behind a moving object.

Tessellation Shaders: These include the tessellation control and tessellation evaluation shaders. They're used to dynamically increase the detail of a surface by breaking it into smaller parts or patches. This is useful for things like smooth terrain, wrinkles in clothing, or highly detailed surfaces without manually modelling every detail.

Compute Shaders: Unlike the others, compute shaders aren't directly tied to drawing pixels on the screen. They're general-purpose and used for complex calculations that benefit from the GPU's speed, such as physics simulations, particle systems, or image processing tasks.

Shaders are typically written in specialized programming languages such as GLSL (OpenGL Shading Language) or HLSL (High-Level Shading Language). While the syntax might look a bit different from common languages like C++, C#, or Python, the core ideas—variables, functions, logic—are the same. Many game engines and graphics APIs provide tools and frameworks to help you get started without needing to write everything from scratch.

Understanding shaders opens a world of creative control. Whether you want to simulate a glowing force field, mimic the look of hand-drawn art, or recreate the subtle bounce of light off a shiny surface, shaders are the key.

2.2. Post-Processing Shaders

In today's graphics programming, post-processing shaders are an essential tool for applying full-screen effects to a scene after it has been rendered. These shaders are used on the finished image/texture and treat it like a flat canvas/plane for extra effects to be layered on rather than manipulating 3D points or textures during the rendering process. A more polished or stylised appearance is the end result. This is all done to give the piece of media -whether it be a game or a film- a unique look.

Bloom (which mimics light bleeding from bright areas), motion blur, depth of field, tone mapping, and colour grading are examples of common post-processing effects. You would anticipate these effects in video games, cinematic visuals, or stylised independent games. Without changing the underlying geometry or lighting, they are also employed to delicately add realism, highlight movement, or evoke particular moods.

These shaders are often run in a full-screen pass, where the rendered scene is fed into a fragment shader via a framebuffer texture. From there, the shader manipulates or processes the pixels using mathematical operations—blurring, blending, manipulating colours, or even applying custom filters like pixelation effects. Since they operate on the 2D image space, post-processing shaders can be extremely flexible and stackable, allowing developers to layer multiple effects in sequence or selectively apply them based on depth, colour, or object masks.

In practice, implementing post-processing frequently involves managing multiple render targets, custom buffers, and multi-pass pipelines. More complex techniques, like screen space ambient occlusion (SSAO) or god rays, require additional depth or lighting information from earlier stages of the render pipeline. Thanks to this, post-processing shaders can range from lightweight visual tweaks to performance-intensive calculations.

To streamline this complex and hard-to-manage process, modern tools like Unity's Universal Render Pipeline (URP) or Unreal Engine's Post Process Stack offer built-in support for post-processing effects that allow users to customize the parameters being fed in. These tools still allow developers to write their own post-process shaders, often in a

language like HLSL or with visual tools like Unity's Shader Graph, to tailor the exact look and behaviour of game visuals to the developers liking.

Another tool in this space is Tinsl, a domain-specific language designed for writing modular, multi-pass shader logic. By allowing developers to specify render blocks, chain passes together, and test effects in a live coding environment, Tinsl streamlines the process of creating intricate post-processing pipelines. Developers can also write their own post-process shaders, often in HLSL or with visual tools like Unity's Shader Graph, to tailor the exact look and behaviour of their game's visuals.

Shaders for post-processing are ultimately about flexibility; they allow developers to adjust the finished image to better suit their creative intent, a games narrative, or to just enhance game play. Many of those final artistic touches are realised in post-processing, whether their aiming for dreamy softness, surreal distortion, or brutal reality.

2.3. Shader Libraries

Shader libraries are collections of reusable shader programs designed to speed up development, keep consistency throughout a project, and promote best practices in graphics programming. These libraries often include a wide variety of shader types—from basic vertex and fragment shaders to more advanced effects like dynamic lighting, reflections, or surface deformations.

Shader libraries serve as a practical basis for the production process. They are essential to developers' ability to rapidly prototype, refine visuals, and maintain visual consistency throughout projects. They allow teams to build on pre-existing and optimised code rather than having to create shaders from the scratch each time. This allows programmers and artists a common language for visual aspects, and speeds up development, whilst lowering bugs.

Major game engines like Unity and Unreal integrate shader libraries directly into their ecosystems. Unity's Standard Assets, various GitHub repositories, and community-driven libraries provide everything from toon shading and procedural textures to complex water simulation and volumetric lighting. Unreal Engine's Material Editor also functions as a shader library system, letting users access and build on a wide range of pre-made materials and effects. Likewise, the open-source nature of Godot allows for extensive sharing and integration of community-made shaders.

Visual previews, consistent structure, and comprehensive documentation are essential for shader libraries to be used to their full potential. It makes it simpler for teams to understand the shaders -how they operate, how to adapt them to meet needs - and integrate them without requiring a more rewriting then necessary. On the other side, poorly organised or opaque libraries may create bottlenecks, making them challenging to use.

Tools like Unity's Shader Graph and Unreal's Material Editor takes the idea further, combining reusable shader logic with node-based interfaces. These systems make it possible to rapidly design and prototype visual effects without digging into low-level code. They offer a middle ground between visual art and technical implementation, allowing both technical artists and programmers to contribute directly to the visual pipeline.

In short, shader libraries are not just a convenience—they're a critical part of modern graphics workflows. They enable faster iteration, foster collaboration, and provide a reliable base for both everyday visuals and cutting-edge effects.

2.3. Render Pipelines

2.3.1. Renderers and the Rendering Pipeline

Rendering is the process through which a 3D scene is transformed into a 2D image that can be displayed on screen. As described in multiple sources, the rendering pipeline serves as the backbone of any real-time graphics system, orchestrating how geometry, textures, and lighting come together to produce a final image that is then rendered to the screen to be viewed. L. Crawford (2022) compares video games to flipbooks: "Video games are like flipbooks. Every few milliseconds, a new picture appears on the screen, which gives the illusion of fluid motion. The faster these images can be drawn to the screen, the smoother the game looks and feels to "play", the speed and efficiency of rendering these images directly influence the visual smoothness and interactivity of modern applications, especially in video games and simulation environments.

2.3.2. Structure of the Rendering Pipeline

The traditional graphics pipeline is made up of several stages, starting with vertex processing -using the vertex shader-, where the geometric data of 3D shapes/models made up of vertices, normals, and texture coordinates, are transformed into screen space. This is followed by rasterization, which converts the geometry into discrete pixels or fragments. Finally, fragment shaders determine the final colour and appearance of each pixel.

Shaders, small GPU-executed programs, are central to programmable rendering pipelines. Vertex shaders operate on individual vertices, transforming positions and attributes, while fragment shaders calculate per-pixel visual details such as colour, lighting, or texture mapping. Post-processing shaders—used for effects like motion blur or depth of field—run after the main scene rendering and are applied across the whole screen.

2.3.3. Advanced and Modular Pipelines

Recent research and development efforts have focused on building modular, runtime-editable shader pipelines. One such framework - built on MeVisLab - adopts the SuperShader concept to enable dynamic shader composition. This approach is diffrent from with earlier visual shader programming systems, such as Voreen or VolumeShop, which required static shader compilation and offered limited runtime flexibility.

This framework supports a visual interface where users can inject custom GLSL shader functions at specific points in the pipeline - For example, before or after volume classification steps- using shader nodes for parameters and includes. This design facilitates rapid prototyping and application-specific customization without requiring recompilation. Medical use cases, like using clip plains for multi-volume medical data or to assist in radiofrequency ablation planning where ellipsoid-shaped ablation zones are rendered based on applicator models. This demonstrates the framework's ability to handle complex, real-time visualizations made for medical needs.

2.2.4. Educational and Practical Implications

Comprehending the render pipeline helps students better understand computer graphics theory and develop their debugging and optimisation skills. Modern shader tools that are built on nodes enable faster iterations and interactive experimentation, facilitating hands-on learning in a variety of applications, such as simulations and games. The shift from fixed-function to programmable and now modular rendering pipelines marks a significant evolution, enabling more flexible and domain-specific visualizations, particularly valuable in fields like healthcare.

2.5. Tool Design

The design of game development tools plays a crucial yet often overlooked role in shaping how game creation is approached and can affect things like who can make games, how

games are made, and the kinds of creative practices that emerge across professional, educational, and independent/indie contexts. The emphasis on ease of use, intuitive interfaces, and real-time feedback from both causal creators to professional teams show a growing recognition that the role of tools has shifted from mere technical tools that only specialists use to dynamic educational and expressive mediums. Whether through visual scripting, procedural generation, or real-time feedback systems, tools increasingly shape not only workflows but also cultural and educational means of game design/creation.

This section reviews the literature on tool design. It draws connections between practices in industry and education, situates modern game development tools within a broader lineage of computer-aided design (CAD), and explores evolving frameworks for understanding the role of tools in the iterative game-making processes.

2.5.2. Usability and Accessibility in Game Development Tools

User-centred design is essential to the effectiveness of development tools in both an industry and educational context. Tools that offer visual editors simplified user interfaces and real-time feedback significantly lower barriers for beginners, particularly in technically complex domains like shader programming or post-processing. Shadertoy, for instance, offers instant visual feedback that facilitates trial-and-error learning, and Unreal's Material Editor and Unity's Shader Graph convert complex functionality into easily navigable, node-based workflows.

These design choices are critical in both formal and informal learning environments. Educational software, drawing from principles of cognitive scaffolding and constructivist learning (Ito, 2009), must balance simplicity with depth—offering enough structure to prevent frustration while still allowing room for creative exploration. Beginner-friendly engines like Klik n Play and Scratch have influenced contemporary tools by embracing drag-and-drop and visual programming paradigms, which allow users to produce meaning through making rather than instruction alone.

The same principles extend to professional contexts. A study of seven game development organizations—ranging from startups to major studios—found that the most valued tool features were adaptability and support for rapid prototyping, both of which facilitate iterative workflows and creative risk-taking.

2.5.3. Tools as Cultural and Creative Enablers

Accessible tools like GameMaker, RPG Maker, and Twine have been crucial in increasing involvement in the indie and amateur game-making communities.

In the indie and amateur game-making scenes, accessible tools like GameMaker, RPG Maker, and Twine have played a key role in broadening participation. These platforms - often used by creators without programming expirence- serve not only as technical environments but also as cultural frameworks. They shape the kinds of stories that can be told and support aesthetics that are often more personal, experimental, or politically expressive (Anthropy, 2012; Keogh, 2019).

The emergence of communities like Glorious Trainwrecks, Flatgames, and RPG Maker forums showcase how intuitive tools catalyze creative expression and often have communities built around them. By enabling rapid creation, collaboration, and iteration, these tools foster developmental and creative environments that reject the dominant industry ideas of polish and profitability in favor of accessibility, experimentation, and self-expression.

These communities can be compared to grassroots design ecosystems, where the line between tool and user blurs and where playful prototyping is not just a workflow but a mode of cultural production. As Salen and Zimmerman (2003) argue, play itself is integral to evaluating design goals—questions like "Is this fun?" or "Does this work?" can only be answered through direct interaction with a playable artifact rather than through static documentation.

2.5.4. Historical Perspectives: From CAD to Collaborative Design

The evolution of tool design can be contextualized within the history of the topic. Early CAD tools from the 1950s and 60s were conceived not only as drafting environments but also as reasoning systems—able to provide what Schön (1983) called "back talk," where a system reflects constraints, implications, or feedback on a designer's actions.

Knowledge-based AI systems started to improve this "back talk" in the 1980s by incorporating domain-specific expertise. This led to the development of Domain-Oriented Design Environments (DODEs), which integrated reusable design logic with factual knowledge to critique ideas, propose alternatives, and even justify those recommendations

(Nelson & Mateas, 2009). The role of the tool shifted from being a passive interface to an active participant in the design process.

Nelson and Mateas (2009) recorded this change by classifying the roles tools play in the design process.

- "Designer's slave," "nanny," "assistant": reducing cognitive load by automating tasks like helping with planning and organization tasks.
- "Advisor": enhancing design reasoning through intelligent feedback based on the content of the current design;
- "Coach", "colleague" or "expert": can contribute domain knowledge and design insight, offering suggestions and improvements to be made with their own "ideas".
- "Collaborator": has the ability to make autonomous design suggestions.
- "Designer": acting as a generative agent capable of full design production, basically being a "dominant" collaborator.

Another way that classifies these design tools are "instruments" and "effective tools" - Khaled et al. (2013), drawing on HCI research by St. Amant and Horton (2002) -, where "instruments" are tools that provide feedback and insight into a design and "effective tools" directly produce and transform design materials. Tools, like Gillian Smith's Tanagra, function as both—allowing level editing while also offering feedback on playability.

2.5.6. Prototyping

Prototyping occupies a central place in this evolving design landscape. More than a step in production, it is a mode of inquiry that reveals properties of games that are otherwise unknowable. As Raph Koster said, "Building a game of a design document is like filming a movie of the director's commentary." Playable prototypes, whether built with Unity, GameMaker, or Twine, allow both creators and stakeholders to assess aesthetic, technical, and experiential qualities.

They also help teams communicate, using it for evaluating art pipelines, gameplay feasibility, and player experience. Tool design is inherently tied to the material politics of

iteration — who can make a prototype, how quickly, and under what constraints determines whose ideas get tested, refined, and shipped.

2.7. Conclusion - Tying everything together

The literature on game development tools and informal game-making communities highlights a shared commitment to accessibility, experimentation, and user-centered design. Through user-friendly interfaces and instant feedback, tools like RPG Maker, Klik n Play, and Unity's visual editors reduce entry barriers and enable learners and hobbyists to participate in game creation.

Communities like Flatgames and Glorious Trainwrecks are prime examples of how accessible tools encourage expressive, cooperative activities that prioritise personal expression over more corporate and "polish" projects. . These spaces reframe game development as a creative, everyday activity shaped by both the affordances and limitations of the tools used.

By using visual programming and immediate feedback to reduce cognitive load and encourage exploration, the most effective instructional tools strike a balance between innovation and simplicity. This approach fosters technological literacy without overwhelming learners.

In general, tool design becomes a cultural force that influences the industry as a whole and how the medium is seen.

Chapter 3 Methodology

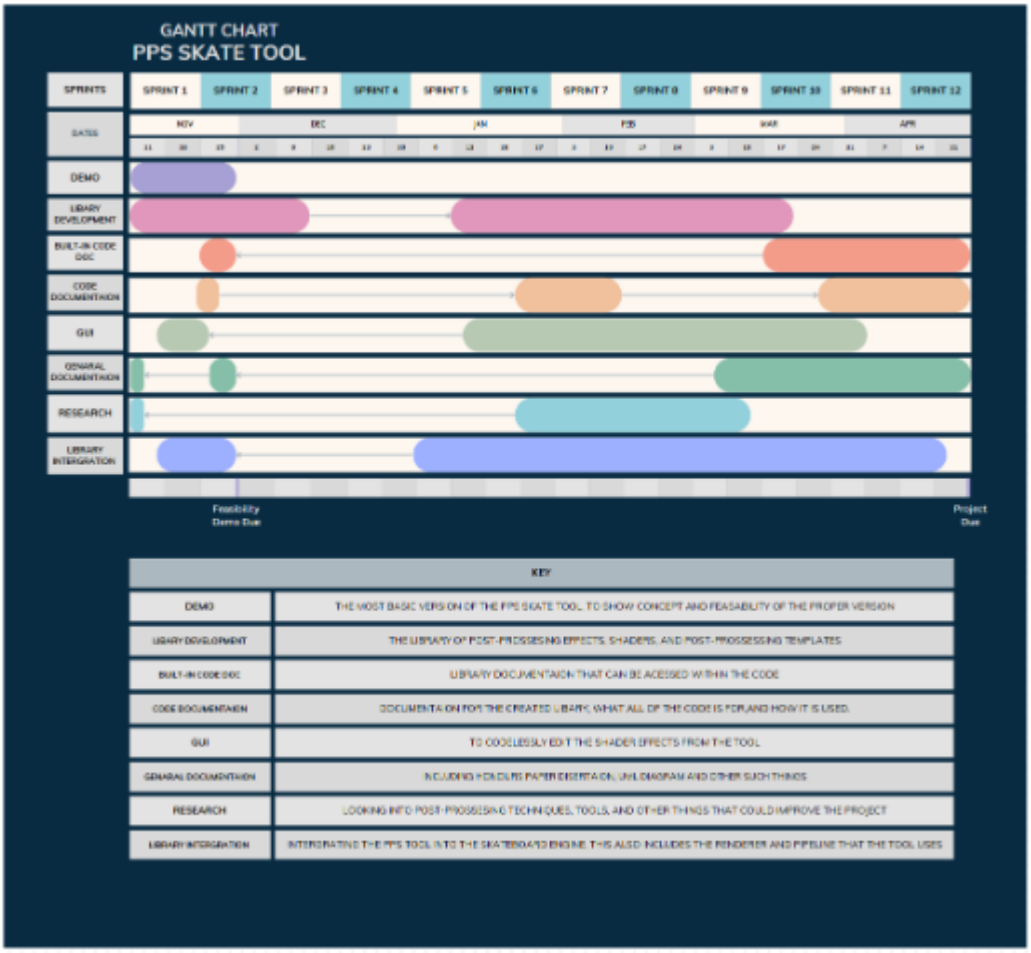
The purpose of this section is to outline the development process for the shader tool "PennyBoard" designed for educational and developmental purposes. This section will cover the planning, development, implementation, and testing of the tool. And will also

provide a comprehensive breakdown of the components that make up the tool including; the shader library, the visual editor, and the real-time preview feature.
This tool serves as a learning aid for those interested in graphics programming, more specifically post-processing effects, and a helpful way to hasten workflow for developers with limited time, resources and knowledge of post-processing. The development focuses on creating a practical, accessible tool that can assist students, developers, and educators alike.

3.1. Planning

Project Goals:
The primary goal of this project is to create a shader tool that supports both learning and developmental needs. This is a tool for helping those who wish to learn more about graphics programming - more specifically about the post-processing process – and those who wish to save time by using a tool instead of making and implementing their own shaders, a time-consuming process that requires a lot of specific knowledge that the average person doesn't have. This tool will help lower the barrier of entry for those without deep technical knowledge in shader development.

Timeline & Milestones:



Target Audience:

The target demographic for this tool is those interested in shader-programming at a beginner level, like students, and developers who wish to accelerate their workflow, saving time by using a tool with an already existing shader library that has pre-built shaders that can easily be customized. The tool could also help educators as a resource that can be used to give students relatively easy hands-on experience.

3.1.1. Main project pieces

Penny Board consists of 4 main components:

The Shader Library, that consists of a collection of pre-built shaders that are easily applied to any project within the engine using the tools interface (GUI). Developers are able to customize these shaders to fit their specific needs.

A Visual Editor; a user-friendly interface for creating and modifying shaders without requiring extensive coding knowledge. This allows users to experiment with different visual effects and see the results in real-time.

A Real-Time Preview, this feature allows users to see changes they made to the shaders in real-time, providing immediate feedback and a more interactive learning and development experience.
And most importantly a renderer for connecting the shaders to Skateboards rendering pipeline. It integrates the shaders from the tool's graphics pipeline, ensuring that they are applied correctly and function as intended. Providing the necessary infrastructure to manage rendering states, buffers, and textures used in post-processing processes.

3.1.2. Research and Development

This is the phase that dives into researching existing shader development tools and techniques, analysing tools such as Shader Graph (Unity), Material Editor (Unreal), and other similar tools to gather inspiration and insights into the logic behind them and features are most beneficial for educational purposes. As well as exploring various post-processing techniques that could be added to the shader library, prioritizing them based on their educational value, most sought after for current developers and customization potential. This research forms the foundation for the design and functionality of the tool.

3.1.3. Tool Architecture

The user interface will be built using "imgui" a reliable free resource that will save time in the development process as it is already a lightweight, and efficient resource. The UI design will prioritize clarity and ease of navigation. Included in the UI will also be a real-time preview screen to see the changes users are making whilst they work.
The renderer is responsible for handling the rendering pipeline and ensuring that shaders work correctly in real-time. It supports efficient shader buffers and ensures that the passing of data is optimized for performance.
The shader library currently consists of 12 unique pre-built shaders for the user to customize and use how they see fit. All shaders are made with efficiency and ease of access in mind, and all have comments guiding any user who wants to crack the hood open and see how the process works for themselves. With an organized and accessible shader library to provide a guiding hand for what any user is looking for.

3.1.4. Implementation Phases

The initial prototype focused on creating a basic version of the tool with a few essential shaders and effects. The visual editor interface also was developed and designed at this stage.
The prototype was then used as the testing ground for developing the shaders used in the shader library and any design changes for the UI before applying them to the skateboard version of the tool, as the DirectX11 version was more easily malleable and fully functioning making the perfect testing ground.
After the initial prototype was developed, transferring the tool into the Skateboard engine started, this in itself proved quite challenging due to skateboards lack of documentation and support as the engine is still currently in development. This stage involved making necessary adjustments for compatibility and ensuring that the tool functions as expected within the engine, this means creating a new render to integrate into skateboards rendering pipeline and editing buffers and making slight adjustments to the shaders code within the shader library.

3.1.5. Documentation and Tutorials

Comprehensive documentation is available to ensure that users can easily understand how to utilize the tool. This will include built-in documentation accessible from within the tool itself via tooltips and comments in the code –though the comments are not fully completed- as well as external resources such as PDF guides. A worksheet is available that can help guide users step-by-step through creating shaders, understanding the tools' GUI, and using the tool itself. Testing will focus on evaluating the performance, usability, and educational impact of the tool. This is done via user testing.

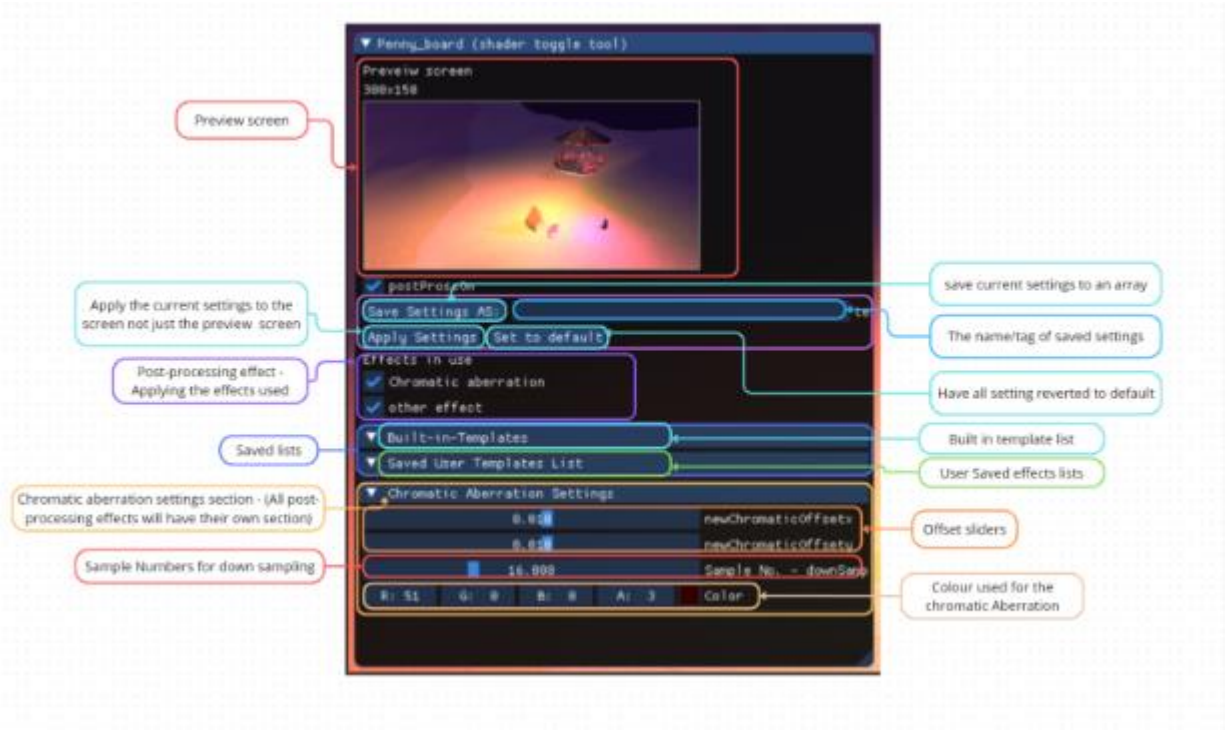
3.2. Implementing

3.2.1. DirectX11 Prototype

The initial prototype was developed in DirectX11 as I was familiar with the API enough to use it to create a fast and realistic prototype, that could easily be compared to the final project/tool. It was initially a basic version of the tool with few essential shaders and effects, mostly focusing on the GUI and how the real tool would interact with the user. The visual editor interface also was developed and designed at this stage.
The prototype was then used as the testing ground for developing the shaders used in the shader library and any design changes for the UI before applying them to the skateboard

version of the tool, as the DirectX11 version was more easily malleable and by its own right a fully functioning tool. Making it the perfect testing ground for shader creating and experimenting with the UIs design.

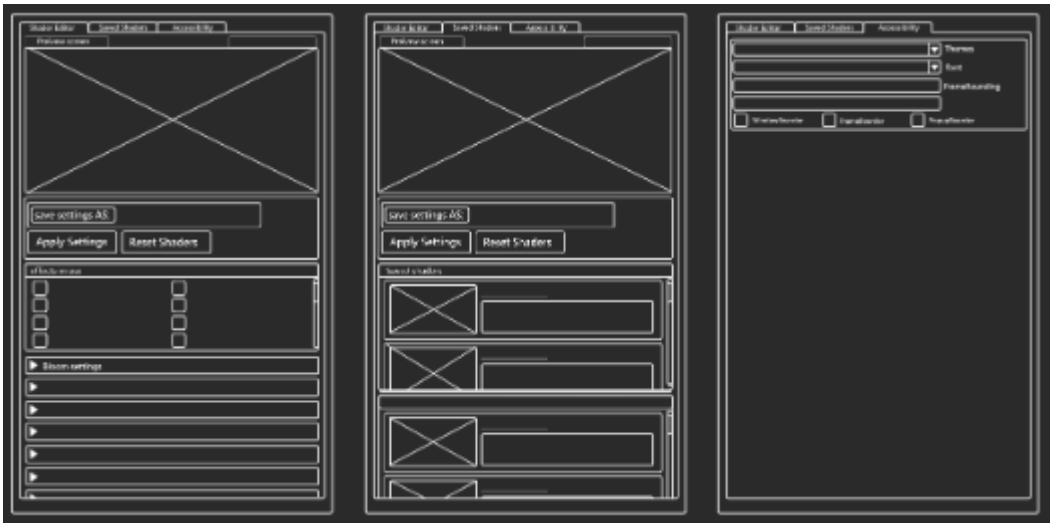
3.2.1.1. Prototyping the GUI



Selected area	Description
Preview screen	A small screen that shows the current settings.
Save settings As - button	Save current settings with the name given in the adjoining box.
Save settings - name space	Text box that contains the tag/name the user types in.
Apply settings - button	Apply current settings to the game screen, not just the preview screen.
Set to default - button	Set all settings to their default.
Effects in use list - section	The section where all the post-processing effects are toggled on or off.
Chromatic aberration – check mark	If “chromatic aberration” is wanted to check the mark.
Built-in-Templates - section	Section where all the pre-built templates are stored.
- Template button	Button to set values to the example pre-built template.
Saved user templates List - section	Section where all the users made/saved effects are stored.
- Saved effect button	Button to set values to the player saved effect that's stored.
Chromatic aberration settings – section	The section dedicated to all the chromatic aberration settings. All effects will have their own section for setting their values and such.
- Offset sliders	Set the offset wanted on the chromatic aberration. There is one for the x-axis and one for the y.

- Sample No. slider	Slide the slider to choose the number of samples the down sampler will take.
-Colour picker	Pick the colour you want to use for the chromatic aberration effect.

The image above shows the DirectX11 prototype. It was designed to be as simple and as easy to pick as possible. As well as being a good base point for continued improvement and additions to later down the line.



The wireframe shown above was a potential design to improve the tool to be later implemented in the final version of the tool in Skateboard. Having three distinct sections: the main editor for creating and editing effects, the saved and pre-built effects tab where all saved effects are stored, and lastly, the tool editor that allows the user to customize the tool itself. This relies mostly on IMGUI own customizable options as they are well maintained, offer a vast number of customizable options as well as save time with overall development. This includes being able to change the theme, font, colouring, and many other things that could help those that need certain visual accommodations.

3.2.1.2. Prototyping Shaders

Shaders are developed and tested individually within the prototype tool. This prototyping phase will ensure that the shaders function properly and efficiently before being transferred to the Skateboard engine.

A couple shaders that have gone through multiple rounds of prototyping to make them more efficient and readable, for example the first bloom effect was made up of 3 separate shaders, 1 to take any pixel that was above a certain threshold and apply it to a new texture, another to blur that new texture and the last shader merged the created texture and the original render texture. It has now been compressed down to one shader file, even though all those shaders have useful functions and can help make more unique effects as

stand-alone shaders like they are now, If a user was looking for specifically bloom it is a lot more efficient to process it all in 1 shader as use multiple buffers to pass data to different places for the same effect, since it would decrease performance and efficiency.

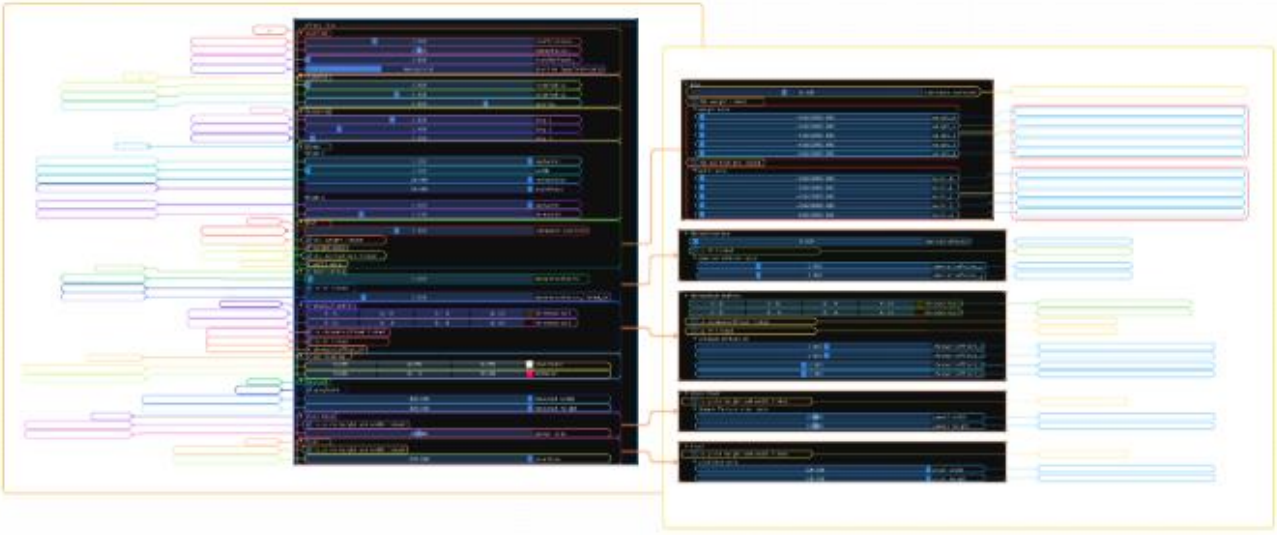
3.2.2. Transferring Prototype to Skateboard / DirectX12

After the initial prototype was developed, transferring the tool into the Skateboard engine started, this proved quite challenging due to Skateboards lack of documentation and support, as the engine is still currently in development causing delays in implementation. This stage involved making necessary adjustments for compatibility and ensuring that the tool functions as expected within the engine, this means creating a new render and multiple pipelines within that renderer to be compatible within skateboards engine. Implementing and editing buffers to work with Skateboard and adjusting the shaders within the shader library.

3.2.2.1. The tools UI

The UI for the tool remains largely unchanged in terms of core design and layout. However, there are key adjustments made to ensure compatibility with DirectX 12, as the syntax in "IMGUI" varies slightly depending on API used.


tab selection
screen size slider
preview screen
Save settings section
Apply settings button
effects selection
renderer selection
effects selection selector
effects editor list
Set to default
effects description
Editor
Saved Effects List
Tool Cue
Preview screen
Screen size
Save Settings As:
Apply Settings -- not currently implemented
Set to default
Effects selection
Render Pass 1
Render Pass 2 - not in use
Render Pass 3 - not in use
PIXELATE_
POSTERIZE_
SCANLINE_
VIGNETTE_
no effect added
effect list
scanline
Vignette
Posturize
Bloom
Blur
ChromaTrueFara
ChromaStylizedFara
Colour Grading
Embossed
Glass Panel
Pixel



For a more in-depth overview, go to the manual. (manual location)

3.2.2.2. Renderer

The renderer utilized in this tool is based on the "208" renderer provided by the most recent version of Skateboard.
However, some key changes have been made to tailor it to the specific needs of the shader tool. Implementing the post-processing buffers and allowing information to be passed from the UI into said buffers, enabling users to change the shaders values in real-time.
Adding functions that allow the user to switch between - and in the future layer- different shaders and post-processing effects.
And having a stand-alone renderer somewhat separate from the main render pipeline allows for easier conversion into a “plug-in”.

3.2.2.3. Adjusting Shaders

The way buffers and textures are handled in the prototype differs to the real tool due to the infrastructure of "Skateboard". In the engine, the texture needed for post-processing effects is encapsulated within an instance data buffer, which is a data structure designed to hold various values required for rendering. As a result, minor adjustments in the shaders code are necessary to accommodate this system. These changes involve altering how texture data is accessed or how certain buffers are handled within the shaders, ensuring compatibility with the engine's rendering framework. Thus, some fine-tuning of the shader code and buffer management will be required to integrate with skateboards infrastructure.

3.3. Testing

Usability testing and educational testing is done at the same time using a questionnaire with different sections contesting of; Basic Shader Concepts, Applied Concepts, and ease of use.

Usability tests in the "Ease of use" category will be conducted with volunteers to gauge the tool's effectiveness for clear and user-friendly UI, as this is supposed to be a tool centred around ease of use and education it's important that the UI is concise and easy to pick up early on. This section consists of 6 questions to test the tools ease of use and ask users for any feedback to improve upon this.

The Educational testing is broken up into two categories "Basic Shader Concepts" and "Applied Concepts" in the questionnaire but also includes the worksheet guide as well. These focus on what the user has learned from using the tool and going through the educational materials provided to assess whether the tool effectively supports learning.

The data from both the worksheet users complete as well as the questionnaires that volunteers have filled out both before and after using the tool will be collected and anonymized.

The tool's performance will also be assessed to identify any potential bottlenecks or areas for optimization. The goal is to ensure that the tool does not negatively affect the performance of the engine, even when multiple shaders are in use.

3.5. Documentation

Comprehensive documentation is available to ensure that users can easily understand how to utilize the tool. The built-in help and tooltips that guide users through the various features and functions will provide immediate support without having to leave the tool or the comments within the code, like the general comments to help users understand the steps the code takes and paragraph comments at the top of a file that describes the files general use these allow more curious users to crack open the hood of the tool and see how it works though some level of programming knowledge would be required.

For more detailed information and help external resources such as PDF guides are available for both the UI and coding sides of the tool. This documentation will cover everything from basic usage to more advanced and complicated aspects of the tool, think of this as a car manual for the Penny Board tool.

A tutorial worksheet is also available as an external resource to help guide users step-by-step through creating shaders and understanding the tool. There may also be an option for a built-in tutorial system that can be toggled on or off for convenience, later down the line.

-Documentation can be found in the appendix.

Chapter 4 Results & Discussion

4.1 Introduction

This chapter presents and interprets the results of the study done to evaluate the educational tool "Pennyboard", created to introduce users to post-processing and shader programming concepts in computer graphics and game development. This study was conducted using a questionnaire given to participants before and after they had used the tool. Participants were guided through the tool by a structured worksheet. This approach provided insight into changes in learners' understanding, their engagement with technical

content, and the overall usability of the tool. Additionally, we will analyse the significance of these findings, and compare them with our previous research, as well as investigate any possible considerations for future tool design and teaching methods.

4.2 Pre-Tool – Initial Understanding of Concepts

Before using the tool, participant understanding of shader programming was limited. When asked about the purpose of shaders, most responses were vague—e.g., "to make games look nice." Few demonstrated awareness of the technical role shaders play in rendering graphics. Additionally, all participants indicated they did not understand the difference between vertex and fragment/pixel shaders, which are fundamental to GPU-based rendering.

When asked about post-processing, the majority of users could not name any specific effects, with only a small amount identifying "bloom." Descriptions of post-processing effects were largely inaccurate or unclear. Most volunteers lacked awareness of how these effects are used in games or digital media and were unable to articulate how such effects contribute to visual storytelling or style.

These findings indicate a significant knowledge gap among participants prior to using the educational tool.

4.3 Post-Tool - Outcomes and Learning Improvements

After using the tool, learners showed noticeable improvement in their familiarity with post-processing terminology and visual effects. Many could now list several effects—bloom, vignetting, colour grading, blur, pixelate, and textured glass—suggesting that the tool helped them visually associate effects with terminology. Some users described post-processing as "a filter that distorts the main image," indicating a developing conceptual grasp, although confusion still existed regarding the distinction between shaders and post-processing.

While understanding of how to technically implement effects (e.g., bloom or texture blending) remained shallow, more participants offered plausible responses rather than avoiding the question entirely, showing that the tool also gave volunteers more confidence. A few were able to state that bloom involved "blurring pixels with high light values," showing a beginning awareness of the steps taken to create the effect.

In applied tasks (e.g., designing an apocalyptic game environment), users provided more relevant and structured answers, such as using desaturation or colour grading to evoke mood, though some participants didn't specify an effect by name they did state the wanted visuals known effects could provide. These were still basic but aligned with typical use cases, indicating that visual experimentation through the tool had strengthened contextual understanding.

4.4 Tool Usability and User Experience

Participants reported quite positive experiences with the tool's UI. It was described as easy to navigate, with clearly labelled elements and responsive controls, a minimalist layout - that helped streamline the learning process -, and helpful Tooltips. With users stating that they could complete tasks with minimal to no guidance. No technical issues, such as crashes or lag, were reported.

Most users did not feel the need for customization but welcomed the idea of additional post-processing effects, and one volunteer expressed the want for more font options. Volunteers showed confidence in their capacity to finish the task on the worksheet independently, and their overall satisfaction with the tool was good. Although one participant said that some steps of the given worksheet could be cut without affecting functionality, the worksheet's step-by-step structure was generally regarded as good.

Engagement and Motivation

Volunteers expressed enjoyment and curiosity during the task, often taking time to "play" with the tool, it also boosted confidence in users by volunteers proactively experimenting with the post-processing effects found with-in the tool. This aligns with the tool's aim of reducing the intimidation factor often associated with graphics programming. While the tool was not designed to teach advanced shader logic, its success in building foundational comfort with the medium was clearly shown in user feedback.

4.5 Interpretation of Findings

The study shows that visual and interactive tools like the one tested here are effective at bridging the gap between technical concepts and user intuition. While a deep understanding of GPU programming and shader logic remained limited, the tool significantly improved familiarity with key terminology, usage contexts, and visual outcomes of post-processing effects.

The most notable shift was in learners' ability to name, apply, and describe post-processing effects. This suggests that direct visual feedback plays a critical role in demystifying

abstract graphics concepts. Although learners were still unable to fully articulate the use of shaders in the context of graphics programming, their engagement with the tool suggests a readiness to pursue further learning.

4.6 Comparison with Existing Research

These findings align with existing literature emphasizing the value of intuitive, low-barrier educational tools in technical domains (Smith, 2015; Khaled et al., 2013). Prior studies on game design and visual programming tools have shown that interactivity and experimentation drive both engagement and retention. Similar to shader platforms like Unity Shader Graph or Unreal's Material Editor, this tool lowers the cognitive load traditionally associated with shader programming by eliminating the need for code-based interaction. Moreover, the observed increase in learner engagement supports theories in educational technology that advocate for playful, exploratory learning models. The tool reflects principles found in constructivist learning environments, where students build knowledge through hands-on experience rather than passive instruction.

4.7 Implications for Education and Tool Design

Educational Practices

The results of this study suggest that tools like this could be meaningfully integrated into early-stage modules within game development or computer graphics curricula. By using such tools, educators can provide students with visual context and immediate feedback before transitioning to more complex topics.
These tools could also be valuable in online or self-directed learning environments, especially for students who may lack access to high-performance computers or formal instruction in graphics programming..

Tool Design Considerations

From a design standpoint, this study highlights the importance of a clear interface, contextual feedback, and low-friction onboarding. Features such as tutorials, tooltips, and visual previews contributed significantly to the tool's usability and learning impact.
Future iterations could include additional functionality for intermediate learners — such as basic node-based scripting, live-coding features, or dynamic debugging tips — to support continued learning beyond the basics.

4.8 Limitations and Challenges

Tool Limitations

While the tool was successful in teaching basic visual concepts, its capabilities were limited to entry-level shader understanding. It lacked the ability to display or manipulate underlying shader code, which could hinder learners seeking to transition to programming-based shader development.

Methodological Limitations

This study was conducted with a small and relatively homogenous sample, primarily composed of beginners. Results may differ with a larger or more diverse participant group, particularly if it includes learners with prior graphics experience. Also, learning gains were measured over a short duration, and long-term retention was not evaluated.
As with many qualitative studies, some interpretations rely on self-reported data, which may introduce bias or inconsistencies in the perception of learning outcomes.

4.9 Conclusion

The educational tool demonstrated clear effectiveness in helping novice users gain initial familiarity with shaders and post-processing effects. Through an accessible interface and visual experimentation, users developed confidence and were able to apply newly learned concepts in simple design tasks. Although advanced shader logic remained out of reach for most participants, the tool met its goal of introducing key terms and visual principles in a friendly, engaging manner.
These findings support the continued use and development of educational tools that prioritize usability and experimentation in complex technical domains. With thoughtful iteration and expanded functionality, tools like this have the potential to play a pivotal role in shaping the future of graphics education—lowering the barrier to entry and fostering a new generation of creative, technically skilled designers.

Chapter 5 Conclusion and Future Work

5.1. Conclusion

5.1.1. Summary of Key Findings

Main Results:

This paper explored the creation and effectiveness of an educational tool designed to bring awareness to post-processing effects and graphics programming, as well as introduce foundational concepts in shader development and post-processing to newcomers in game development. Volunteers were given a structured pre- and post-questionnaire to gauge the tool's effectiveness. Users' comprehension of shader-related terminology, the role and intent of post-processing effects, and how visual improvements affect game aesthetics all showed improvement.

Prior to using the tool, participants expressed little to no knowledge of shaders, pipelines, or post-processing effects, with most responses being vague or incomplete. However, after following guided use of the tool - using the worksheet- volunteers were able to name multiple post-processing techniques, talk about their visual impact, and demonstrate a more applied understanding of how such effects contribute to mood, tone, and gameplay immersion. While some technical gaps remained in areas requiring deeper programming knowledge (e.g., combining textures in shaders), the tool clearly enabled some improvements in understanding basic concepts.

Tool’s Contribution to Education:

The tool proved effective in simplifying shader creation for beginners, the tool successfully turned what is usually seen as a daunting field into a more approachable, interesting, and most importantly fun learning environment. The tool's intuitive UI, real-time feedback, and hands-on workflow supported users, giving them a strong sense of satisfaction whilst boosting their confidence. Most participants described the tool as easy to navigate, well-labelled, and effective at showcasing the practical applications of post-processing effects. As a result, the tool served not only as an educational platform but also as a creative sandbox that invited exploration, play, and experimentation.

5.1.2. Implications for Educational Practices

Incorporating interactive, visual learning resources like this shader tool into curricula can greatly improve students' understanding and engagement when they are first learning computer graphics. The tool's hands-on interface and instant visual feedback provided

learners with an easier way to understand complicated programming and visual concepts than traditional lecture-based or text-heavy learning materials.
This tool can complement instructor-led teaching or self-paced learning environments by providing students with opportunities to apply theoretical knowledge in a visual and interactive format easily providing almost instant gratification encouraging users to continue further. Feedback further suggests that tools like this can serve as a great bridge between beginner-level curiosity and more advanced shader learning.
In the broader context of game development education, this tool fills an important gap between conceptual understanding and technical implementation of post-processing/visual effects. By focusing on visual outcomes and providing simplified interfaces for shader manipulation, it becomes the perfect stepping stone, allowing learners to build confidence before progressing to more complex topics like GLSL/HLSL scripting, multi-pass rendering, or engine-level shader integration.

5.1.3. Limitations of the Study

Despite its promising results, the study faced a few limitations. Because of the limited sample size of volunteers and the fact that most of the participants were novices, the findings cannot be applied to a larger range of experience levels. Additionally, the assessment's length was restricted to brief tool usage, making the assessment unable to gauge long-term learning results or retention. Future research would benefit from a more diverse sample that includes intermediate and advanced users, as well as a longer study to measure how learning with the tool evolves over time

Tool Limitations:

The tool itself, while reasonably effective for beginners, lacks more advanced features that would appeal to more experienced users or those wishing to delve deeper into shader coding. Areas such as multi-pass rendering, custom scripting, and real-time engine integration were identified as beneficial next steps in development. Some participants also shown a lack of clarity in differentiating shader types and the ability to teach this inside the tool would be greatly beneficial using beginner-friendly explanations or examples to bridge knowledge gaps.

Performance-wise, the tool performed reliably during testing, with no reported crashes or delays. However, its limited customization and absence of features like precise parameter control or performance profiling may reduce its appeal for some users.

5.1.4. Final Reflection

This research has demonstrated the educational value of a shader and post-processing tool tailored for beginners. By providing an accessible interface that prioritised immediate visual feedback, the tool significantly enhanced users' experience and understanding of both technical and aesthetic aspects of shader development. It provided a structured but flexible learning environment that encouraged exploration and creativity whilst also addressing one of the common barriers in computer graphics education.

Carving out a clear view of how such tools can be designed to meet the needs of new users while laying a foundation for deeper, more technical learning in the future. These findings contribute to ongoing discussions about interactive learning tools in game development education and offer practical insights for educators, developers, and researchers aiming to lower the barrier to the complex field of graphics programming.

5.2. Future Work

Although PennyBoard's current version provides a solid basis for creative expression, intuitive UI, and real-time effect creation, there are a number of improvements that can be made that could significantly improve both the usability and functionality of the tool. These additions will expand the way the tool can be used and will be able to accommodate users of all skill levels, from novice beginners to more advanced users. Based on user feedback, observational data, and broader educational goals, this chapter outlines areas for future improvement. These changes will hopefully develop this project into a more robust learning tool that can support deeper learning and greater understanding of shader programming and graphics concepts while also catering to a wider range of users.

5.2.1. Improving the Tool

Tool Additions

The tool received positive feedback by participants for its easy-to-use, minimalistic and intuitive design. Nonetheless, there is plenty of ways to improve the tool further, especially for users unfamiliar with technical tools.

A more modular, customizable workplace could allow users with different skill levels better customise their learning experience. Integrating pop-out panels and multi-monitor support

would enhance focus and workflow efficiency, especially for users accustomed to digital art or animation tools. Allowing users to rearrange elements such as the parameter controls and the preview window, making the tool more appealing to both technical and creative users.
Additions like more beginner-friendly navigation, clearer visual cues, and detachable UI panels could prove invaluable.

[add img/figure/wireframes]

A requested improvement included precise input for values that currently rely on sliders - making the tool more accessible, especially for users who find it difficult to manipulate sliders for exact values, and more font options for readability. These changes have been noted, and plans to put them in place have already been made. One feature that couldn't be produced during the project's time frame but is a planned addition is multi-pass rendering and support for layered post-processing. These additions would allow for the creation of more complex and customizable visual outputs and would give learners a deeper understanding and appreciation of real-time rendering techniques.
custom shader scripting with live preview and a dual-mode system that combines visual node editing with code-based control for intermediate or advanced users. A live coding mode, enhanced by built-in syntax documentation, code templates, and an integrated API for exporting effects to the game engine, would significantly broaden the tool's utility beyond entry-level education and developmental uses.
Currently, effects are viewable only in the dedicated preview panel. In future updates, users will be able to apply and view effects directly on the main screen, making the tool properly usable in the long run.
Future versions will include the ability to properly save and apply user-made or pre-built effects.
The inclusion of pre-built effects will give users ready-to-use shaders they can apply, dissect, or modify to suit their own needs. These effects will serve both as practical tools and learning references.

More Advanced Shader and Coding Features

To support more advanced learners and to extend the educational value of the tool, future versions might include features such as a live coding mode to allow users to write and modify shader code with immediate visual feedback -Similar to ShaderToy's interface. This

dual-mode approach—visual and textual—caters to a wider range of users and learning preferences, offering flexibility for both rapid experimentation and fine-grained code editing, even though it will be targeted to more advanced/intermediate learners. To go with this and other more generalised API included in this tool there are plans for built-in syntax documentation, code templates, as well as polishing the existing code to be closer to a proper API. This would significantly broaden the tool's utility beyond entry-level education and developmental uses.

5.2.2. Exploring New Educational Approaches

Gamification and Interactive Learning

To increase motivation and sustained engagement, the tool could benefit from gamification elements. This approach could transform shader learning into a more compelling, goal-oriented process, especially for younger or self-directed learners.
Leaning more into interactive learning elements, such as real-time feedback and guided built-in and video tutorials, could also help bridge the gap between visual exploration and theoretical understanding. These features would make learning more intuitive while reinforcing key concepts through practice.

Collaborative Learning and Peer Feedback

As proven by [insert refrence here] communities and collaboration can greatly improve tool usability and make learning a smoother more fun and engaging time.
Providing collaborative features like shared shader libraries and peer-to-peer feedback using potential community pages. Encouraging users to upload and interact with a community would not only foster creativity but also build user confidence, create an environment for problem-solving, and encourage mutual learning.

5.2.3. Further Research Directions

Educational Tool Evaluation

Future studies could involve proper comparisons of different shader tools to determine which are most effective for teaching specific topics such as general shaders, lighting, effects, or texture blending. This comparative approach would help situate the tool within the broader educational landscape and provide insights into best practices for digital tool design in computer graphics education.

5.2.4. Conclusion of Future Work

These upcoming features show a commitment to accessibility, and creativity in shader creation. By refining both the technical foundation and user-facing design of the tool, with the aim to create an environment where shader development is not only efficient and effective but also welcoming and not as intimidating to a broader audience.

List of References

Almeida, M.S.O. and Da Silva, F.S.C. (2013) 'A systematic review of game design methods and tools,' *Lecture Notes in Computer Science*, pp. 17–29. https://doi.org/10.1007/978-3-642-41106-9_3.

Clarke, M.J. and Wang, C. (2020) *Indie games in the digital age*. Bloomsbury Publishing USA.

Crawford, L. and University of Edinburgh, School of Informatics, Institute of Computing Systems Architecture, (2022) 'Shader Optimization and Specialization,' *Shader Optimization and Specialization* [Preprint].

Game Engines Evaluation for Serious game development in education (2021a). https://ieeexplore.ieee.org/document/9559053.

Game Engines Evaluation for Serious game development in education (2021b). https://ieeexplore.ieee.org/document/9559053.

Granof, C.J. (2021) 'TINSL: Tinsl Is Not a Shading Language,' *TINSL: Tinsl Is Not a Shading Language* [Preprint]. https://digital.wpi.edu/concern/etds/rx913s769.

Hasu, J. (2018) 'Fundamentals of Shaders with Modern Game Engines,' *Fundamentals of Shaders With Modern Game Engines* [Preprint]. https://lutpub.lut.fi/bitstream/handle/10024/158721/FundamentalsOfShadersWithModernGameEnginesJoonaHasu.pdf?sequence=1.

Kasurinen, J., Strandén, J.-P. and Smolander, K. (2013) 'What do game developers expect from development and design tools?,' *What Do Game Developers Expect From Development and Design Tools?*, pp. 36–41. https://doi.org/10.1145/2460999.2461004.

Neil, K. (2015) *Game design tools : Can they improve game design practice?* https://theses.hal.science/tel-01344638/.

'Rendering pipeline, shaders, and effects' (2009) in *Apress eBooks*, pp. 227–240. https://doi.org/10.1007/978-1-4302-1818-0_9.

Rieder, C. *et al.* (2011) 'A shader framework for rapid prototyping of GPU-Based volume rendering,' *Computer Graphics Forum*, 30(3), pp. 1031–1040. https://doi.org/10.1111/j.1467-8659.2011.01952.x.

Smith, A., Nelson, M. and Mateas, M. (2009) 'Computational support for play testing game sketches,' *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 5(1), pp. 167–172. https://doi.org/10.1609/aiide.v5i1.12368.

Sobota, B. and Pietriková, E. (2023) 'The role of game engines in game development and teaching,' in *IntechOpen eBooks*. https://doi.org/10.5772/intechopen.1002257.

Toisoul, A., Rueckert, D. and Kainz, B. (2017) 'Accessible GLSL Shader programming,' *Eurographics*, pp. 35–42. https://doi.org/10.2312/eged.20171024.

Itō, M. (2012) *Engineering play: A cultural history of children's software*. Cambridge, Mass: MIT Press.

Bibliography

I

Appendices

Abertay University

GDPR Research Data Management
Data Sign Off Form

For undergraduate or postgraduate student projects supervised by an Abertay staff member.

This form MUST be included in the student's thesis/dissertation. Note that failure to do this will mean that the student's project cannot be assessed/examined.

Part 1: Supervisors to Complete

By signing this form, you are confirming that you have checked and verified your student's data according to the criteria stated below (e.g., raw data, completed questionnaires, superlab/Eprime output, transcriptions etc.)

Student Name:	Dannielle G. Smith		
Student Number:	2101323		
Lead Supervisor Name:	Naman Merchant		
Lead Supervisor Signature			
Project title:	Tool creation for an in-house engine: How does one create a shader/post-processing tool for use in education?		
Study route:	PhD ☐	MbR ☐	MPhil ☐
	Undergraduate ☑	PhD by Publication ☐	

Part 2: Student to Complete

	Initial here to confirm 'Yes'
I confirm that I have handed over all manual records from my research project (e.g., consent forms, transcripts) to my supervisor for archiving/storage	
I confirm that I have handed over all digital records from my research project (e.g., recordings, data files) to my supervisor for archiving/storage	

Abertay University

GDPR Research Data Management
Data Sign Off Form

I confirm that I no longer hold any digital records from my research project on any device other than the university network and the only data that I may retain is a copy of an anonymised data file(s) from my research	
I understand that, for undergraduate projects, my supervisor may delete manual/digital records of data if there is no foreseeable use for that data (with the exception of consent forms, which should be retained for 10 years)	

Student signature : 

Date: **30/04/2025**

Abertay University

Appendix C(i): Participant Information Sheet and Research Consent Form Template

Project Title: "Tool creation for an in-house engine: How does one create a shader/post-processing tool for use in education?"
Researcher Name: Dannielle G. Smith
Researcher Supervisor: Naman Merchant
Contact Information: 2101323@uad.ac.uk

What is the research about?
We invite you to participate in a research project about…
You are being invited to participate in a research study to test a new tool used for educational and developmental use in postprocessing effects and evaluate the ease of use and how it compares to other education means.
The study will help improve the tools development and how it can be used for educational/developmental purposes.

Do I have to take part?
This form has been written to help you decide if you would like to take part. It is up to you and you alone whether you wish to take part. If you do decide to take part you will be free to withdraw at any time without providing a reason and without penalty.

What will I be required to do?
If you agree to participate, you will be asked to:

- Complete 2 different work sheets during the allotted time and finish a questionnaire for what you have learned during the time, there may also be a presentation to start the study off.
- This study will take approximately 1 hr done on a specified date; the date will be given on a later date.

How will you handle my data?
Your data will be stored in an anonymized form and willonly be accessible to Dannielle G. Smith, see contact details below. This means that nobody including the researchers could reasonably identify you within the data. Your data will be stored in a secure database, with data fully anonymized at the point of collection. Your responses are treated in the strictest confidence - it will be impossible to identify individuals within a dataset when any of the research is disseminated (e.g., in publications/presentations/datasets). Abertay University acts as Data Controller (DataProtectionOfficer@abertay.ac.uk).

Retention of research data

Researchers are obliged to retain research data for up to 10 years' post-publication, however your anonymized research data may be retained indefinitely (e.g., so that researchers engage in open research, and other researchers can access their data to confirm the conclusions of published work). Consistent with our data retention policy, researchers retain consent forms for as long as we continue to hold information about a data subject andfor 10 years for published research (including Research Degree thesis).

Consent statement:
Abertay University attaches high priority to the ethical conduct of research. Please consider the following before indicating your consent on this form. Indicating your consent confirms that you are willing to participate in the research, however, indicating consent does not commit you to anything you do not wish to do and you are free to withdraw your participation at any time. You are indicating consent under the following assumptions:

- I understand the contents of the participant information sheet and consent form.
- I have been given the opportunity to ask questions about the research and have had them answered satisfactorily.
- I understand that my participation is entirely voluntary and that I can withdraw from the research (parts of the project or the entire project) at any time without penalty and without having to provide an explanation.
- I understand who has access to my data and how it will be handled at all stages of the research project.

PLEASE INITIAL BOX:	Yes, I do consent	No, I do not consent
I consent to take part in this study conducted by **Dannielle G. Smith** who intend to use my data for further research examining the given graphics tools' development and the way it will be taught/used.		

Signature:
I confirm that I am willing to take part in this research:
PRINT NAME:
SIGNATURE:
DATE:

You can find our procedure for complaints (regarding research projects) and our privacy notice and legal basis for processing research data at: https://www.abertay.ac.uk/legal/privacy-notice-for-research-participants/

Penny-board GUI work sheet

Objective:

By the end of the lesson, students should be able to:

1. Understand the concept of post-processing shaders in graphics programming.
2. Use provided post-processing shader tool to modify the final rendered image.
3. Write basic post-processing shader code to modify the final rendered image.
4. Implement a few common post-processing effects (e.g., bloom, sepia, grayscale).
5. Understand the pipeline in which post-processing occurs.

Materials Needed:

- Skateboard engine
- The Penny-board tool
- Text editor or integrated IDE (Visual Studio)

1. Setting Up a Post-Processing Tool

Objective: Set up the **Penny-board** tool and familiarize yourself with its features.

1. Run the tool and ensure everything compiles correctly. If there are any issues, ask your supervisor for assistance.
2. Review the **accessibility features** and explore them.

```
#include "Penny-Board/PennyRender.h"
```

```
pen::PennyRender screen_Renderer_;
pen::PennyRender scene_Renderer_;
```

Make sure to initialise the renders in the h file of the game level your using.

```
screen_Renderer_.Init();
scene_Renderer_.Init();
```

Initialize the renders in the level init function, as shown above.

```
Scene::OnRender();

OnRender_firtPass();

screen_Renderer_.renderScreen(scene_Renderer_.getCurrentOutputSRVData().Get());
//test_Renderer2.MultiRenderPass(scene_Renderer_.getCurrentOutputSRVData().Get());
screen_Renderer_.renderToScreen_end(scene_Renderer_.getCurrentOutputRTVData(), scene_Renderer_.getCurrentOutputSRVData());
```

Have the screen used for post-processing effects drawn in the renderer, like shown in the above img.

```
scene_Renderer_.SetRenderTargets_(scene_Renderer_.getCurrentOutputTexture() , scene_Renderer_.getCurrentOutputRTVData());

scene_Renderer_.Begin();

//--
//scene stuff -- user can make scene here and not go within the tool to change anything
uint32_t Indexdata[]{0,1,2};
std::vector<pen::Vertex> movedvertices = vertices;
pen::InstanceData testdata;
testdata.TextureIndex = renderTexture.get()->GetViewIndex();

//draw triangle
for (auto& a : movedvertices) { a.Position += float3(2.5, 0, 0); }
scene_Renderer_.DrawVertices(movedvertices.data(), 3, Indexdata, 3, &testdata);

matrix World = glm::translate(float3(-5, 0, 0));
testdata.World = World * glm::translate(float3(5, -5 + Ypos, 0)) * glm::eulerAngleXYZ(glm::radians(Rotation.x), glm::radians
testdata.ColourScale = float4(1, 1, 1, 0.9);
testdata.SpecularColor = SpecularC;
testdata.SpecularPower = SpecularPower;
testdata.SpecularWeight = SpecularWeight;


pen::Vertex Quad[4] =
{
        {float3(-1,1,0),float3(0,0,0),float2(0,0),      float3(0,0,-1)  },
        {float3(-1,-1,0),float3(0,0,0),float2(0,1),     float3(0,0,-1)  },
        {float3(1,-1,0),float3(0,0,0),float2(1,1),      float3(0,0,-1)  },
        {float3(1,1,0),float3(0,0,0),float2(1,0),       float3(0,0,-1)  }
};

uint32_t quadindices[6] =
{
        0,1,2,2,3,0
};

scene_Renderer_.DrawVertices(Quad, 4, quadindices, 6, &testdata);

//--

scene_Renderer_.End();

scene_Renderer_.SetRenderTarget_ToBackBuffer();
```

Make sure to set the render target as shown above before the render begin, after the begin you can create the scene/ game environment then end the scene and set the render target to the back buffer.

```
screen_Renderer_.PennyBoardIMGUI();
```

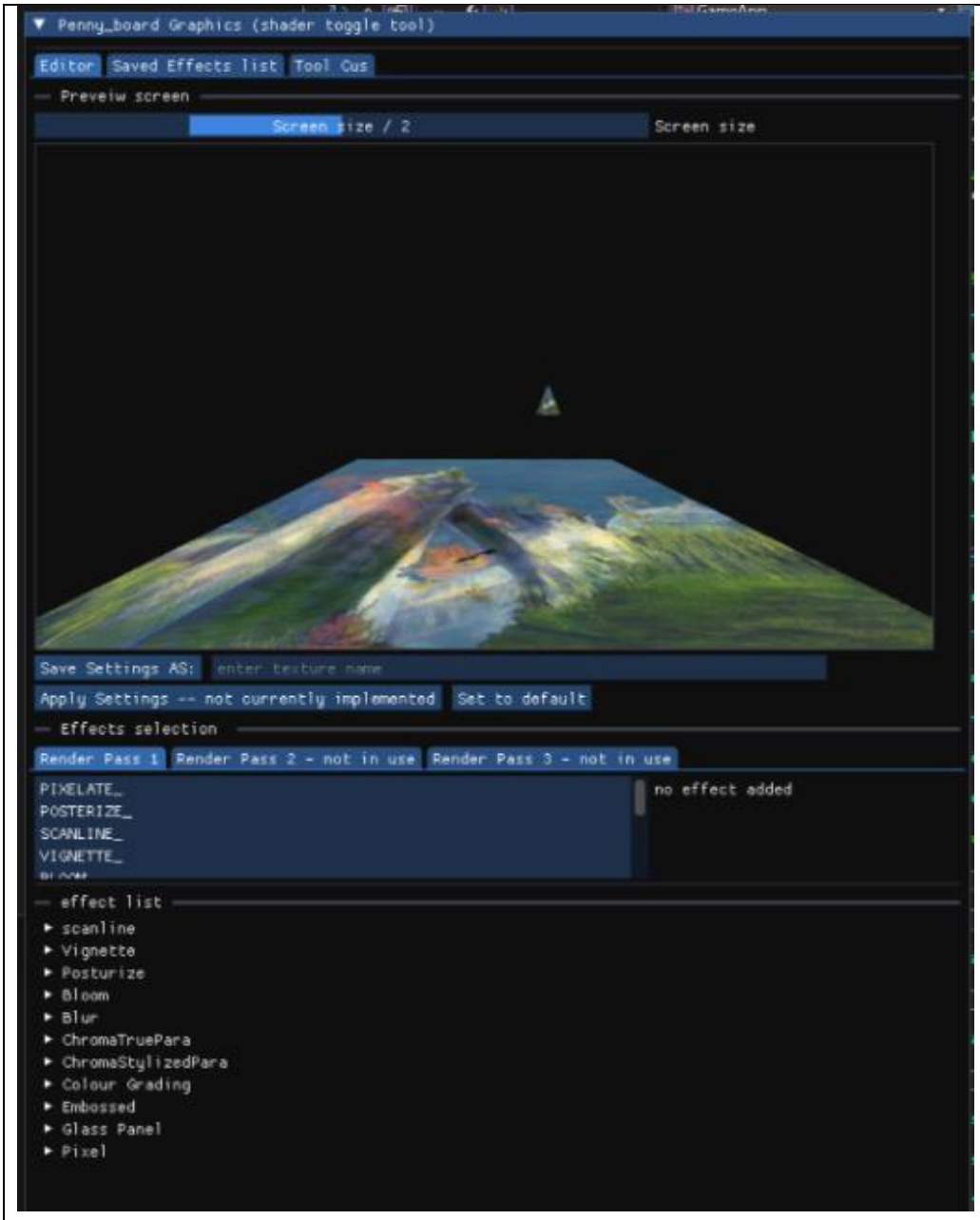
Have the tools GUI added to the IMGUI function in the levels code.

2. Set up Vignette in GUI

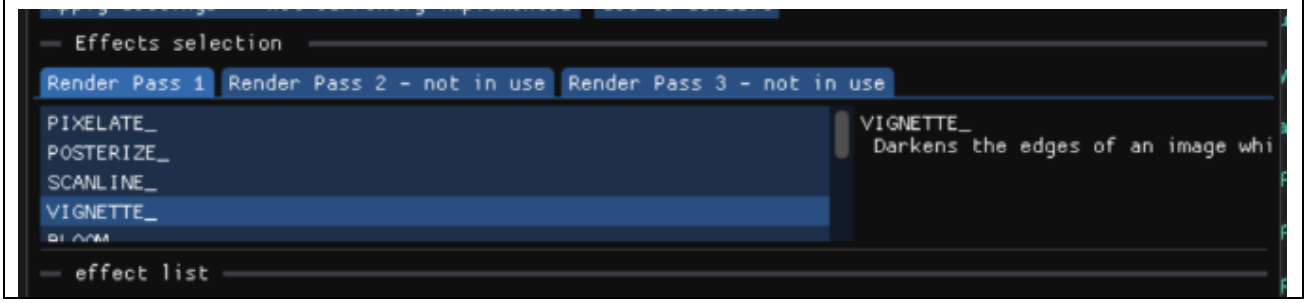
Objective: Set up a Vignette effect within the GUI.

A **vignette** in graphics and photography refers to a **visual effect that darkens (or sometimes lightens) the edges of an image**, drawing the viewer's attention toward the centre. Often used in game for horror or a damage/hurt effect.

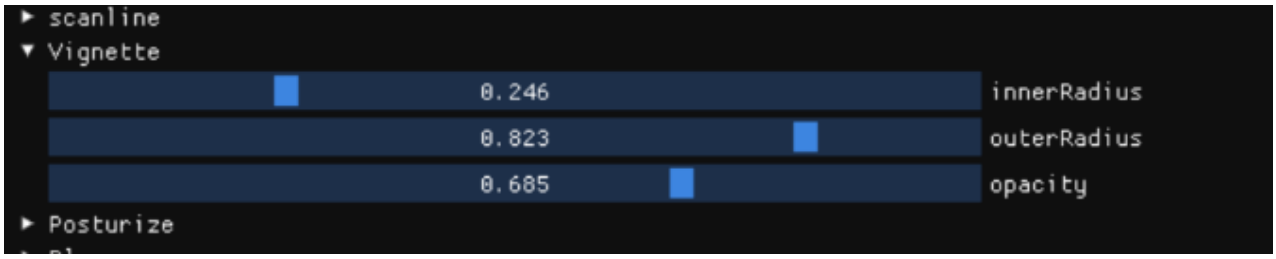
1. Run the programme, then open the Penny-board application.



2. It will look like this, (go through layout). There is tool tips if you hover above the chosen area.

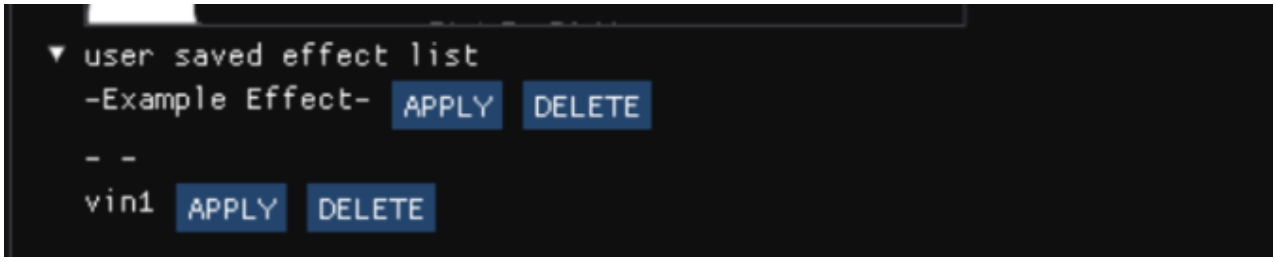


3. Ensure that **vignette** has been selected. As shown in the above image.



4. Adjust the **inner radius, outer radius**, and **opacity** settings to your preference

5. Name your settings/shader and click the **Save Settings AS** button.



6. The saved effect will appear in the **user saved effect List** as "vin1" (or the name you chose).

3. Walking through Vignette

Objective: Walk through the code to explain how the vignette shader is set up and used.

The files used here are located in the pennyboard file and are called “PennyRenderer.cpp” and “PennyRenderer.h”

--header-

```
Skateboard::MultiResource<Skateboard::BufferRef> VignettingDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> VignettingDataBuffer;
BufferViewDesc vignetting_desc;
VignettingBuffer m_VignettingBuffer = VignettingBuffer();
```

With-in the header the buffer ref, buffer view description and the struct “m_VingnettingBuffer” used to contain all the data/values the buffer will be passing along to the shaders, are initialised.

-- init everything/buffer --

```
void PennyRender::Init()
{
```

```
//all post-processing buffers data // 4 - slot
Layout.AddConstantBufferView(2);
```

The buffer slot used for passing all the values is initialized/set using the layout and is slot 2 as show above.

-raster pipeline

```
void PennyRender::Init_ShaderPermutation(RasterizationPipelineDesc Raster) {
        //piepline permutations
        //for (uint32_t p = 0; p < 10; p++)
        for (uint16_t p = 0; p < EFFECT_PERMUTATIONS; p++)
        {
                auto RSPD = Raster;
                RSPD.DepthStencil.DepthEnable = true;
```

This is where the pipelines differ. As shown above this will loop for the number of pixel shaders/effects to create pipelines for each of them as pixel shaders can't be swapped out of a pipeline in directX12.

```
case VIGNETTE_:
        RSPD.SetPixelShader(L"vignetting_ps");
        break;
```

```
  PipelineDesc PiplDesc;
  PiplDesc.GlobalLayoutSignature = RootSig;
  PiplDesc.Type = PipelineType_Graphics;
  PiplDesc.TypeDesc = &RSPD;
  Pipelines[p] = ResourceFactory::CreatePipelineState(PiplDesc);
```

This is where the pipeline is created within the loop, having the discription of the pipeline being made first.

```
//-- postprocessing buffers
InitBuffers(bufferdesc);

 //VignettingDataBuffer Buffer
 VignettingDataBuffer.ForEach([&](BufferRef& ref)
{ref = ResourceFactory::CreateBuffer({ .AccessFlags = ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead, .Size = sizeof(VignettingBuffer) }); });
```

After the pipelines were initialized, it is then the buffers turn. Using the InitBuffers function, the vignette buffer is created. Making sure that it can write to the cpu and read to the gpu, with the buffers size coming from the size of the struct used for the buffer.

```
//-- postprocessing data desc
InitData(InstanceDataBufferSize);

 //vignettingDATA
 vignetting_desc.InitAsStructuredBuffer<VignettingBuffer>(offset, InstanceDataBufferSize / sizeof(VignettingBuffer));
```

The description of the buffer is then set/initialized using the buffers struct and the struct size.

//-- begin

```
BeginIncrementCounter();//
VignettingDataBuffer.IncrementCounter();
```

In the begin the IncrementCounter is called for the vignetting buffer to consistently update the buffer every frame. (basically it “takes” the buffer to the next frame) ///change this to be more understandable

//--end

```
EndCopyToBuffer();
    //switch (EFFECT_PERMUTATIONS)
    switch (CURRENT_STATE_)
    //switch (ShaderToggles)
    {
case VIGNETTE_:
      GraphicsContext::CopyDataToBuffer(VignettingDataBuffer.Get().get(), offset, sizeof(VignettingBuffer), &m_VignettingBuffer);
      break;
```

At the end the values are all copied/passed into the buffer and to make sure it's the right buffer/pipeline a switch statement is used.

//- drawing

```
    DrawVBIB(&vbv, &ibv, data, CurrentRenderPass_in);
  DrawVBIB_SetInlineResourceViewGraphics();//
switch (CURRENT_STATE_)
{
case VIGNETTE_:
      RenderCommand::SetInlineResourceViewGraphics(4, VignettingDataBuffer.Get().get(), vignetting_desc, ViewAccessType_ConstantBuffer);
      break;
```

When ”drawing” with this renderer the resource view is also set, taking in the data buffer data, the buffers description and the type of view access it is (all our effect view assesses are going to be a constant buffer).

// -- the shader

```
#include "../../CMP203/CMP203_Shaders/Structs.hlsli"
#include "PennyStructs_2.hlsli"
```

```
//buffers
ConstantBuffer<Constants> PushData : register(b0, space0);
StructuredBuffer<InstanceData> Instances : register(t0, space0);

ConstantBuffer<vignettingPara> Parameters : register(b2, space0);
```

//setting values

Setting all the values being used from the buffer we just looked at, as well as getting the texture from the instance data.
Getting the Image Pixel Color, It looks up the color of the current pixel from a texture.

Vertical Fade Effect
This line creates a vertical dimming (fade) by using a sine wave on the y-position of the pixel:
Pixels at the top and bottom are dimmer.
The center stays brighter.

Convert to Center-Based Coordinates
To make the vignette, it needs to know how far each pixel is from the screen center:
Now the center of the screen is (0,0), and the corners are closer to (±1, ±1).

Calculate Distance from Center
This calculates how far a pixel is from the center, like drawing a circle:

Smooth Transition Between Radii
It figures out how close the pixel is to the outer edge (based on innerRadius and outerRadius), then fades it smoothly. This creates the vignette gradient—pixels near the center stay bright; those near the edge darken.

Apply Both Effects
It multiplies the pixel color by:
the vignette effect (factor)
the vertical fade (verticalDim)

Blend With Original Based on Opacity
Using the opacity value:
It mixes the original color and the vignette result.
Low opacity means almost no effect; high opacity means strong vignette.

Then the final step adds the two colors together to create the final pixel color for the screen.
That it done, and it will then render to screen.

5. Do it yourself with another shader

Objective: Create a custom effect using the GUI, and, if desired, explore the code to see how the shader/s is constructed. Chose a brief bellow.

- 1. Create a **night-time environment** in a game, how would you use **post-processing effects** like fog, ambient light adjustments, and colour filters to enhance the atmosphere?
- 2. Create a **dream-like sequence**, what **post-processing effects** would you combine (e.g., blurring, soft lighting, colour shifts) to achieve that effect?

6. Do the questionnaire

Objective: Fill out the questionnaire.

Thank you for completing the worksheet. Please turn this in with the questionnaire.

Questionnaire:

Section 1: Basic Shader Concepts:

1.1. What is the main purpose of shaders in graphics programming?

1.2. What is the difference between a vertex shader and a fragment/pixel shader?

1.3. When/where are post processing effects used the most?

1.4. Please name 1 postprocessing effect. / Write/list down as many postprocessing effects you know.

1.5. What is a post-processing effect?

1.6. How do post-processing effects enhance the visual appeal of a scene or image?

Section 2: Applied Concepts:

2.1. What is the process for combining two textures in a shader, specifically when adding one texture's pixel colour to another? Please outline the steps involved.

2.2. Describe the steps involved in creating a bloom effect.

2.3. What is the impact of using multiple layers of post-processing effects on frame rates?

2.4. You are tasked with creating an apocalyptic environment in a game. How would you utilize post-processing effects for example desaturation, and colour grading to convey and enhance the atmosphere?

2.5. How would you enhance the sense of danger and urgency in the environment using post-processing effects to emphasize chaotic or deteriorating conditions?

Thank you for completing the questionnaire. Please turn this in with the worksheet.

Questionnaire:

Section 1: Basic Shader Concepts:

1.1. What is the main purpose of shaders in graphics programming?

1.2. What is the difference between a vertex shader and a fragment/pixel shader?

1.3. When/where are post processing effects used the most?

1.4. Please name 1 postprocessing effect. / Write/list down as many postprocessing effects you know.

1.5. What is a post-processing effect?

1.6. How do post-processing effects enhance the visual appeal of a scene or image?

Section 2: Applied Concepts:

2.1. What is the process for combining two textures in a shader, specifically when adding one texture's pixel colour to another? Please outline the steps involved.

2.2. Describe the steps involved in creating a bloom effect.

2.3. What is the impact of using multiple layers of post-processing effects on frame rates?

2.4. You are tasked with creating an apocalyptic environment in a game. How would you utilize post-processing effects for example desaturation, and colour grading to convey and enhance the atmosphere?

2.5. How would you enhance the sense of danger and urgency in the environment using post-processing effects to emphasize chaotic or deteriorating conditions?

2.6. how do you feel you did on task 5? Describe what you did and how you feel about the result and using the tool with no assistance or instructions.

Section 3: Ease of use:

3.1. How easy was the UI to navigate. In your opinion was the layout easy to understand/accurately labelled.

3.2. Does the tool offer a smooth learning curve with onboarding tutorials, hints, or tooltips to help new users?

3.3. Are the documentation or help features easily accessible for users who need assistance?

3.4. Does the GUI respond quickly to user input without delays? Or does the tool crash or freeze often?

3.5. Are tasks streamlined in a logical, step-by-step manner to avoid confusion or redundant actions?

3.6. Would customization options interest you?

4. This is where you can say anything you particularly liked or disliked about the tool as well as any changes you would like to see made, this is not mandatory but appreciated.

Thank you for completing the questionnaire. Please turn this in with the worksheet.

Penny-board Plug-in Manual

Honours

Dannielle G. Smith

2101323

Table of Contents

Setting up the tool

```
#include "Penny-Board/PennyRender.h"
```

```
    pen::PennyRender screen_Renderer_;
    pen::PennyRender scene_Renderer_;
```

Make sure to initialise the renders in the h file of the game level your using.

```
screen_Renderer_.Init();
scene_Renderer_.Init();
```

Initialize the renders in the level init function, as shown above.

```
Scene::OnRender();

OnRender_firtPass();

screen_Renderer_.renderScreen(scene_Renderer_.getCurrentOutputSRVData().Get());
//test_Renderer2.MultiRenderPass(scene_Renderer_.getCurrentOutputSRVData().Get());
screen_Renderer_.renderToScreen_end(scene_Renderer_.getCurrentOutputRTVData(), scene_Renderer_.getCurrentOutputSRVData());
```

Have the screen used for post-processing effects drawn in the renderer, like shown in the above img.

```
scene_Renderer_.SetRenderTargets_(scene_Renderer_.getCurrentOutputTexture() , scene_Renderer_.getCurrentOutputRTVData());

scene_Renderer_.Begin();

//--
//scene stuff -- user can make scene here and not go within the tool to change anything
uint32_t Indexdata[]{0,1,2};
std::vector<pen::Vertex> movedvertices = vertices;
pen::InstanceData testdata;
testdata.TextureIndex = renderTexture.get()->GetViewIndex();

//draw triangle
for (auto& a : movedvertices) { a.Position += float3(2.5, 0, 0); }
scene_Renderer_.DrawVertices(movedvertices.data(), 3, Indexdata, 3, &testdata);

matrix World = glm::translate(float3(-5, 0, 0));
testdata.World = World * glm::translate(float3(5, -5 + Ypos, 0)) * glm::eulerAngleXYZ(glm::radians(Rotation.x), glm::radians
testdata.ColourScale = float4(1, 1, 1, 0.9);
testdata.SpecularColor = SpecularC;
testdata.SpecularPower = SpecularPower;
testdata.SpecularWeight = SpecularWeight;


pen::Vertex Quad[4] =
{
        {float3(-1,1,0),float3(0,0,0),float2(0,0),      float3(0,0,-1)  },
        {float3(-1,-1,0),float3(0,0,0),float2(0,1),     float3(0,0,-1)  },
        {float3(1,-1,0),float3(0,0,0),float2(1,1),      float3(0,0,-1)  },
        {float3(1,1,0),float3(0,0,0),float2(1,0),       float3(0,0,-1)  }
};

uint32_t quadindices[6] =
{
        0,1,2,2,3,0
};

scene_Renderer_.DrawVertices(Quad, 4, quadindices, 6, &testdata);

//--

scene_Renderer_.End();

scene_Renderer_.SetRenderTarget_ToBackBuffer();
```

Make sure to set the render target as shown above before the render begin, after the begin you can create the scene/ game environment then end the scene and set the render taget to the back buffer.

```
screen_Renderer_.PennyBoardIMGUI();
```

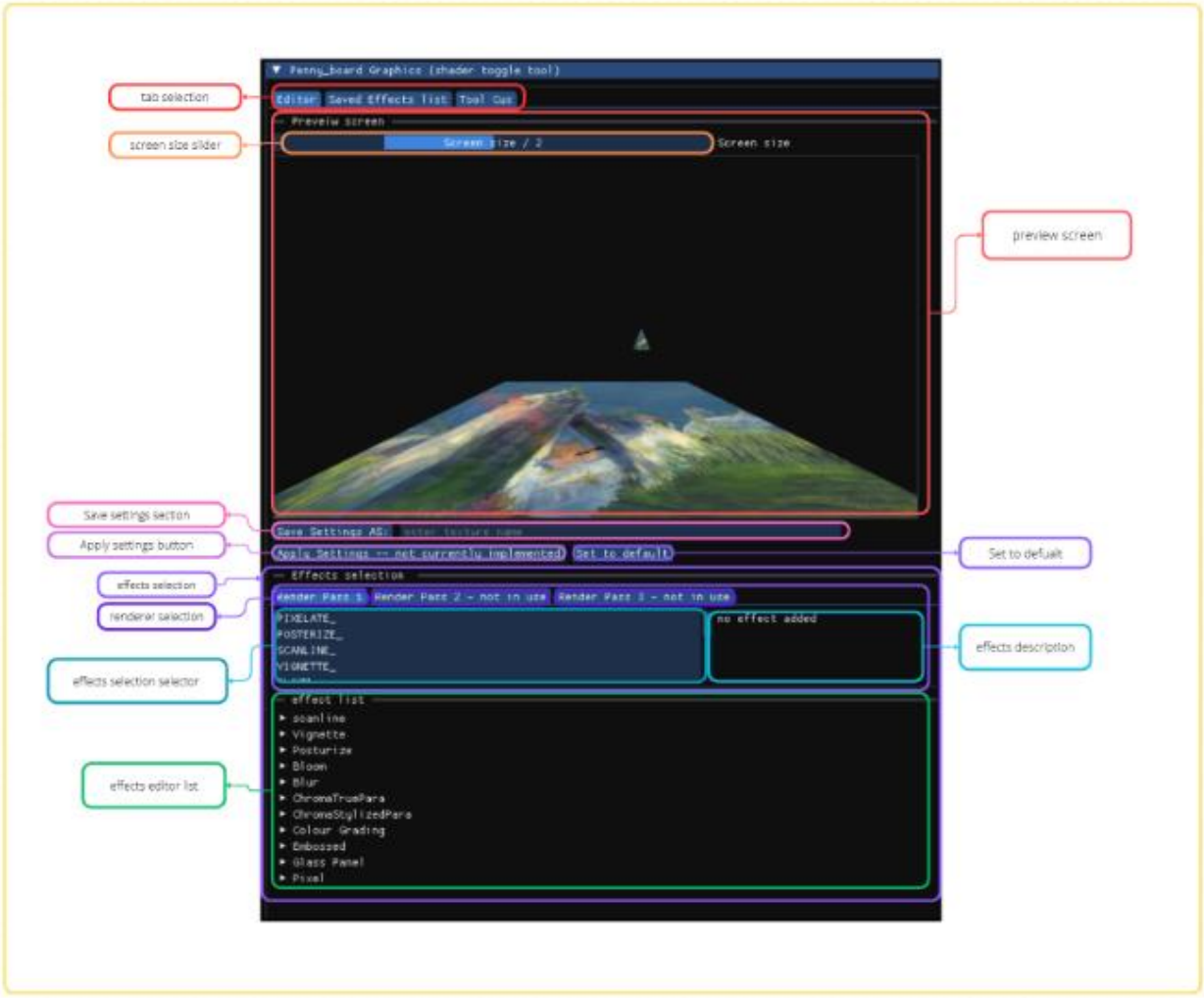
Have the tools GUI added to the IMGUI function in the levels code.

The tool should now work.

GUI

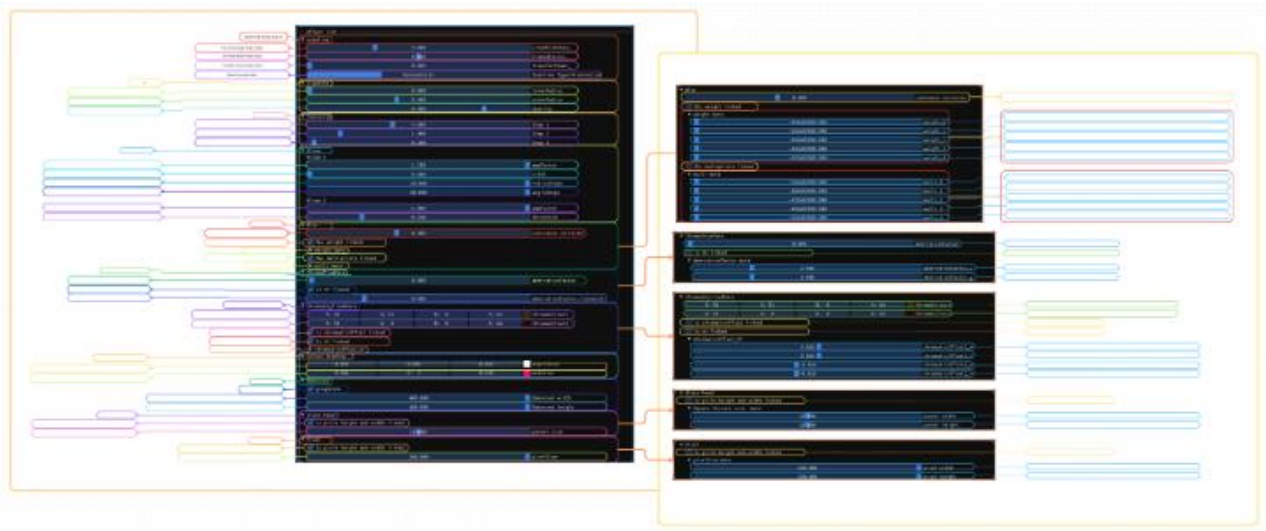
GUI layout

Editor tabs



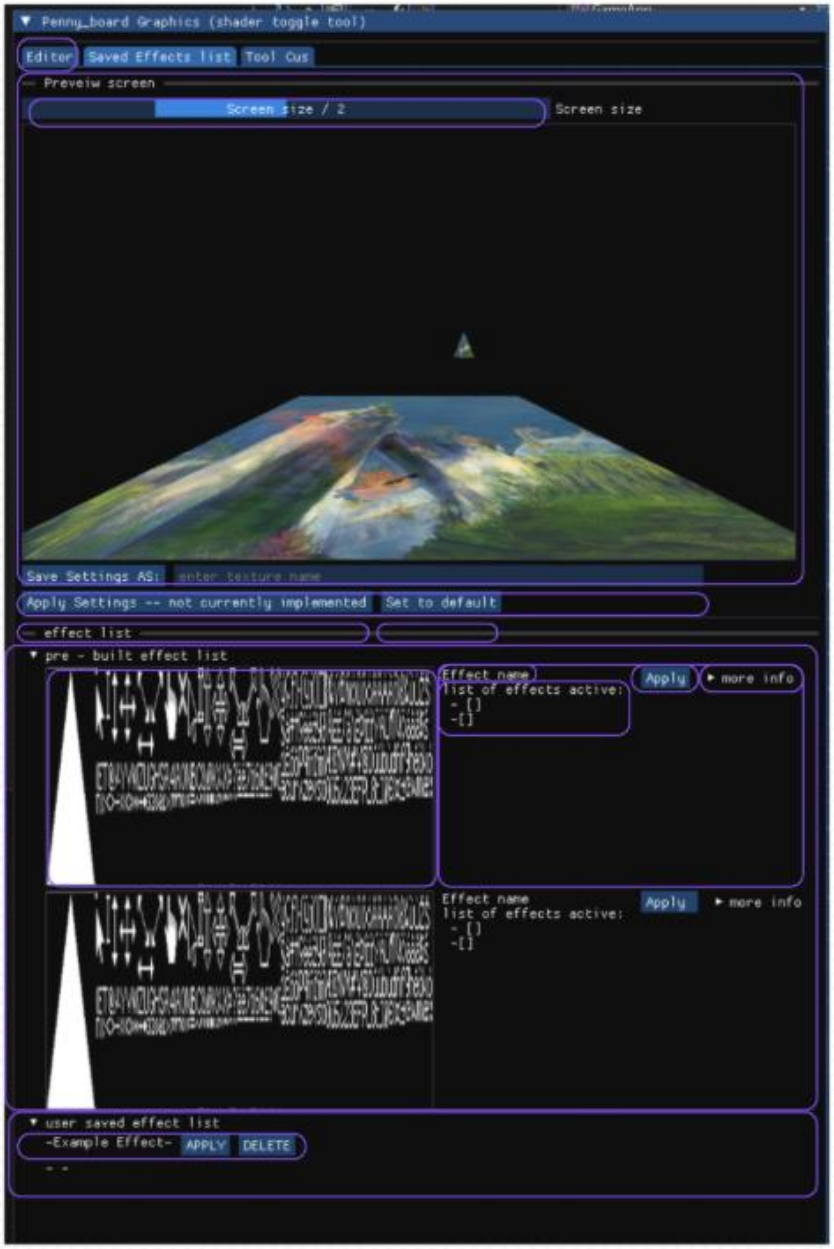
Selected area	Description
Preview screen	A small screen that shows the current settings.
Save settings As - button	Save current settings with the name given in the adjoining box.
Save settings - name space	Text box that contains the tag/name the user types in.
Apply settings - button	Apply current settings to the game screen, not just the preview screen.
Set to default - button	Set all settings to their default.
Effects in use list - section	The section where all the post-processing effects are toggled on or off.

- Offset sliders	Set the offset wanted on the chromatic aberration. There is one for the x-axis and one for the y.
- Sample No. slider	Slide the slider to choose the number of samples the down sampler will take.
-Colour picker	Pick the colour you want to use for the chromatic aberration effect.

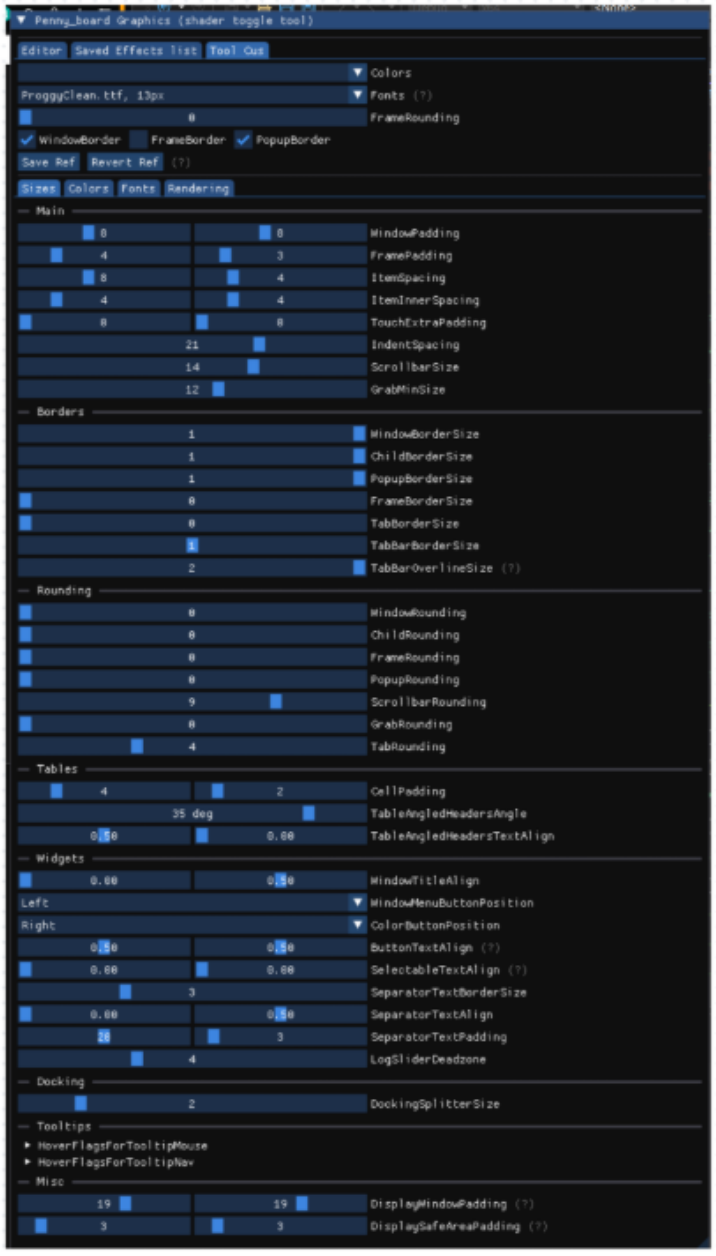


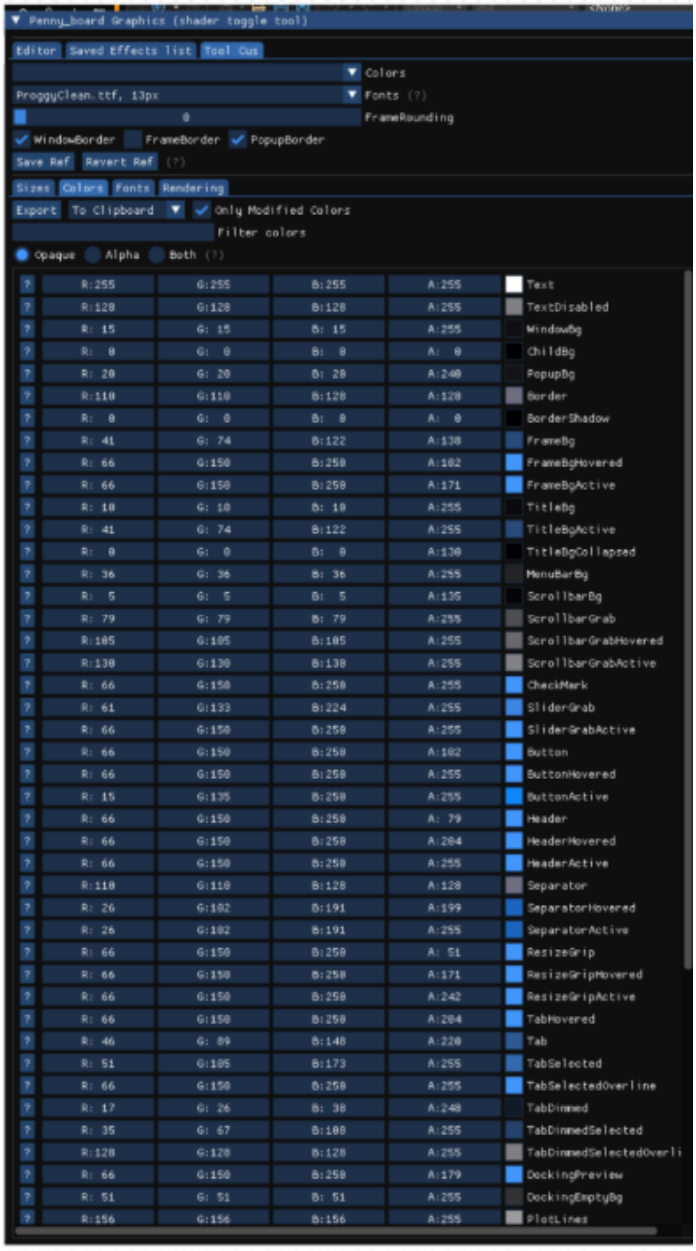
Selected area	Description

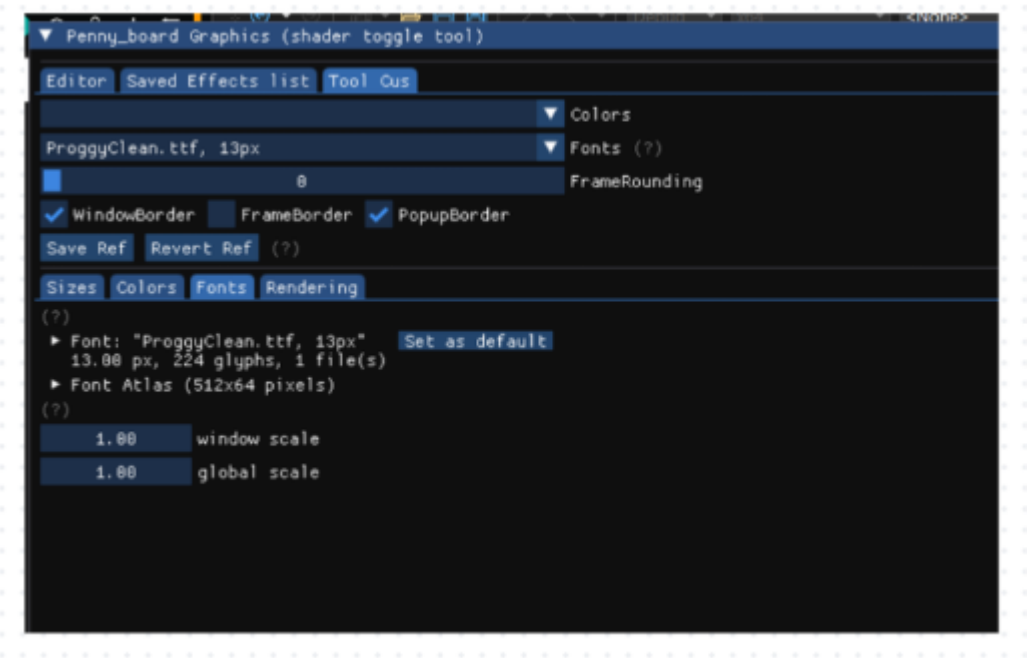
Saved lists tab



Tool customizer tab


Penny_board Graphics (shader toggle tool)
Editor
Saved Effects list
Tool Cus
Colors
ProggyClean.ttf, 13px
Fonts (?)
FrameRounding
WindowBorder
FrameBorder
PopupBorder
Save Ref
Revert Ref (?)
Sizes
Colors
Fonts
Rendering
Main
WindowPadding
FramePadding
ItemSpacing
ItemInnerSpacing
TouchExtraPadding
IndentSpacing
ScrollbarSize
GrabMinSize
Borders
WindowBorderSize
ChildBorderSize
PopupBorderSize
FrameBorderSize
TabBorderSize
TabBarBorderSize
TabBarOverlineSize (?)
Rounding
WindowRounding
ChildRounding
FrameRounding
PopupRounding
ScrollbarRounding
GrabRounding
TabRounding
Tables
CellPadding
TableAngledHeadersAngle
35 deg
TableAngledHeadersTextAlign
Widgets
WindowTitleAlign
WindowMenuButtonPosition
Left
ColorButtonPosition
Right
ButtonTextAlign (?)
SelectableTextAlign (?)
SeparatorTextBorderSize
SeparatorTextAlign
SeparatorTextPadding
LogSliderDeadzone
Docking
DockingSplitterSize
Tooltips
HoverFlagsForTooltipMouse
HoverFlagsForTooltipNav
Misc
DisplayWindowPadding (?)
DisplaySafeAreaPadding (?)


Penny_board Graphics (shader toggle tool)
Editor
Saved Effects list
Tool Cus
Colors
ProggyClean.ttf, 13px
Fonts (?)
0
FrameRounding
WindowBorder
FrameBorder
PopupBorder
Save Ref
Revert Ref
(?)
Sizes
Colors
Fonts
Rendering
Export
To Clipboard
Only Modified Colors
Filter colors
Opaque
Alpha
Both (?)
R:255 G:255 B:255 A:255 Text
R:128 G:128 B:128 A:255 TextDisabled
R: 15 G: 15 B: 15 A:255 WindowBg
R: 0 G: 0 B: 0 A: 0 ChildBg
R: 20 G: 20 B: 20 A:240 PopupBg
R:110 G:110 B:128 A:128 Border
R: 0 G: 0 B: 0 A: 0 BorderShadow
R: 41 G: 74 B:122 A:138 FrameBg
R: 66 G:150 B:250 A:102 FrameBgHovered
R: 66 G:150 B:250 A:171 FrameBgActive
R: 10 G: 10 B: 10 A:255 TitleBg
R: 41 G: 74 B:122 A:255 TitleBgActive
R: 0 G: 0 B: 0 A:130 TitleBgCollapsed
R: 36 G: 36 B: 36 A:255 MenuBarBg
R: 5 G: 5 B: 5 A:135 ScrollbarBg
R: 79 G: 79 B: 79 A:255 ScrollbarGrab
R:105 G:105 B:105 A:255 ScrollbarGrabHovered
R:130 G:130 B:130 A:255 ScrollbarGrabActive
R: 66 G:150 B:250 A:255 CheckMark
R: 61 G:133 B:224 A:255 SliderGrab
R: 66 G:150 B:250 A:255 SliderGrabActive
R: 66 G:150 B:250 A:102 Button
R: 66 G:150 B:250 A:255 ButtonHovered
R: 15 G:135 B:250 A:255 ButtonActive
R: 66 G:150 B:250 A: 79 Header
R: 66 G:150 B:250 A:204 HeaderHovered
R: 66 G:150 B:250 A:255 HeaderActive
R:110 G:110 B:128 A:128 Separator
R: 26 G:102 B:191 A:199 SeparatorHovered
R: 26 G:102 B:191 A:255 SeparatorActive
R: 66 G:150 B:250 A: 51 ResizeGrip
R: 66 G:150 B:250 A:171 ResizeGripHovered
R: 66 G:150 B:250 A:242 ResizeGripActive
R: 66 G:150 B:250 A:204 TabHovered
R: 46 G: 89 B:148 A:220 Tab
R: 51 G:105 B:173 A:255 TabSelected
R: 66 G:150 B:250 A:255 TabSelectedOverline
R: 17 G: 26 B: 38 A:248 TabDimmed
R: 35 G: 67 B:100 A:255 TabDimmedSelected
R:128 G:128 B:128 A:255 TabDimmedSelectedOverli
R: 66 G:150 B:250 A:179 DockingPreview
R: 51 G: 51 B: 51 A:255 DockingEmptyBg
R:156 G:156 B:156 A:255 PlotLines



How the GUI is set-up

//--

```
void PennyRender::PennyBoardIMGUI_PreveiwScreen() {
        //-- barriers to tye/sync render target
        TextureBarrier renderTargetBarrier{};
        renderTargetBarrier.SyncBefore = SKTBD_SYNC_RENDER_TARGET;
        renderTargetBarrier.SyncAfter = SKTBD_SYNC_PIXEL_SHADING;
        renderTargetBarrier.AccessBefore = SKTBD_ACCESS_RENDER_TARGET;
        renderTargetBarrier.AccessAfter = SKTBD_ACCESS_COMMON;
        renderTargetBarrier.LayoutBefore = SKTBD_LAYOUT_RENDER_TARGET;
        renderTargetBarrier.LayoutAfter = SKTBD_LAYOUT_COMMON;
        renderTargetBarrier.Resource = OutputTexture.Get().get();
        renderTargetBarrier.SubresourceRange = TextureSubresourceRange();//

        BarrierGroup grouptoRTV(&renderTargetBarrier);
        RenderCommand::Barrier(&grouptoRTV, 1);
        //--

        ImGui::SeparatorText("Preveiw screen");

        ImGui::SliderInt("Screen size", &previewscreenSize, 1, 4, "Screen size / %d%");
        ImGui::SetItemTooltip("Preveiw screen size, Fullsize/ n");

        //preveiw screen
        ImGuiIO& io = ImGui::GetIO();
        ImTextureID my_tex_id = OutputSRV.Get()->GetImTextureID();//turning srv to a usable ImTextureID - for preview screen
        //move to next frame
        renderToScreen_end(OutputRTV, OutputSRV);

        float my_tex_w = (float)1366 / previewscreenSize;
        float my_tex_h = (float)768 / previewscreenSize;
        {
                static bool use_text_color_for_tint = false;

                ImVec2 uv_min = ImVec2(0.0f, 0.0f);                 // Top-left
                ImVec2 uv_max = ImVec2(1.0f, 1.0f);                 // Lower-right
                ImVec4 tint_col = use_text_color_for_tint ? ImGui::GetStyleColorVec4(ImGuiCol_Text) : ImVec4(1.0f, 1.0f, 1.0f, 1.0f);   // No tint
                ImVec4 border_col = ImGui::GetStyleColorVec4(ImGuiCol_Border);
                ImGui::Image(my_tex_id, ImVec2(my_tex_w, my_tex_h), uv_min, uv_max, tint_col, border_col);
                ImGui::SetItemTooltip("Preveiw screen;\nshow the effected in development before adding to the proper screen/view");
        }
```

//--

```
void PennyRender::PennyBoardMAUI_PassApply(int PassNo){

	const char* items[] = { "PIXELATE_", "POSTERIZE_", "SCANLINE_", "VIGNETTE_",
		"BLOOM_", "CHROMABER_TRUE_", "CHROMABER_STYLE_", "COLOURGRAD_", "GLASS_", "EMBOS_", "BLUR_GAU_","n/a"};

	const char* items_desc[] = {
		"PIXELATE_ \n Turns given value into a pixelated version, like looking at a lower resolution",
		"POSTERIZE_\n Reduces the number of colors in an image, creating a stylized, high-contrast look with distinct color bands or gradients.",
		"SCANLINE_\n Simulates the appearance of lines running across the screen, for a retro or vintage CRT monitor look by adding subtle dark bands to an image.",
		"VIGNETTE_\n Darkens the edges of an image while keeping the center bright, drawing attention to the focal point and creating a more dramatic or cinematic atmosphere.",
		"BLOOM_ \n Creates a glowing haze around the bright areas of an image, enhancing light sources.",
		"CHROMABER_TRUE_\n simulates color fringing by slightly offsetting the red, green, and blue channels along the edges of objects, similar to what 3D movies look like without the glasses.",
		"CHROMABER_STYLE_ \n Similar to true Chromatic aberation, this effect mimics the fringing whilst having the colours used customiseable giving a unique look. ",
		"COLOURGRAD_\n Adjusts the colours and tones of an image to create a specific mood or aesthetic, enhancing visual storytelling through subtle shifts in hue, saturation, and contrast.",
		"GLASS_ \n Simulates the appearance of looking through panelled textured glass",
		"EMBOS_ \n Creates the illusion of raised or recessed surfaces on an image, adding depth and texture by highlighting edges with a contrast of light and shadow.",
		"BLUR_GAU_ \n Smooths the image by averaging pixel values based on a Gaussian distribution, creating a soft, evenly blurred effect.",
		"no effect added"};
```

```
		static int item_current = 11;
		ImGui::ListBox(items_desc[item_current], &item_current, items, IM_ARRAYSIZE(items), 4);
		//ImGui::ListBox("listbox", &PassNo, items, IM_ARRAYSIZE(items), 4);

		switch (item_current)
		{
		case 0:
			//CURRENT_STATE_ = PIXELATE_;
			SetCurrentState(PIXELATE_);
			break;
		case 1:
			//CURRENT_STATE_ = POSTERIZE_;
			SetCurrentState(POSTERIZE_);
			break;
		case 2:
			//CURRENT_STATE_ = SCANLINE_;
			SetCurrentState(SCANLINE_);
			break;
		case 3:
			//CURRENT_STATE_ = VIGNETTE_;
			SetCurrentState(VIGNETTE_);
			break;
		case 4:
			//PSPEL.SetPixelShader(1"bloom.ps");
			//CURRENT_STATE_ = BLOOM_;
			SetCurrentState(BLOOM_);
			break;
		case 5:
			//CURRENT_STATE_ = CHROMABER_TRUE_;
			SetCurrentState(CHROMABER_TRUE_);
			break;
		case 6:
			//CURRENT_STATE_ = CHROMABER_STYLE_;
			SetCurrentState(CHROMABER_STYLE_);
			break;
		case 7:
			//CURRENT_STATE_ = COLOURGRAD_;
			SetCurrentState(COLOURGRAD_);
			break;
		case 8:
			//CURRENT_STATE_ = GLASS_;
			SetCurrentState(GLASS_);
			break;
		case 9:
			//CURRENT_STATE_ = EMBOS_;
			SetCurrentState(EMBOS_);
			break;
		case 10:
			//CURRENT_STATE_ = BLUR_GAU_;
			SetCurrentState(BLUR_GAU_);
			break;

		default:
			//CURRENT_STATE_ = 11;
			SetCurrentState(POSTPROSSESSING_);
			break;
		}

		m_BlurGauBuffer;
	}
```

///---

```
void PennyRender::PennyBoardIMGUI_editorTab()
{
        PennyBoardIMGUI_PrevelwScreen();

        ImGui::SeparatorText("Effects selection ");
        ImGui::SetItemTooltip("Where you can select effects in any given order using 'render passes' to create you unique effect ");

        if (ImGui::BeginTabBar("##tabs", ImGuiTabBarFlags_None))
        {
                if (ImGui::BeginTabItem("Render Pass 1")) {

                        PennyBoardIMGUI_PassApply(0);

                        ImGui::EndTabItem();

                }
                if (ImGui::BeginTabItem("Render Pass 2 - not in use")) {
                        PennyBoardIMGUI_PassApply(1);

                        ImGui::EndTabItem();

                }
                if (ImGui::BeginTabItem("Render Pass 3 - not in use")) {
                        PennyBoardIMGUI_PassApply(2);

                        ImGui::EndTabItem();

                }

                ImGui::EndTabBar();
        }
```

```
{
        ImGui::Separator();

        ImGui::SeparatorText("effect list");
        ImGui::SetItemTooltip("Lists all effects and allows user to edit said effects");

        if (ImGui::TreeNode("scanline"))
        {
                //ImGui::Text("scanline");

                ImGui::SliderFloat("LineThickness_", (float*)&m_ScanlineBuffer.LineThickness_, 0, 10);
                ImGui::SetItemTooltip("The thickness of the bands that have a tint/saturation change.");
                ImGui::SliderFloat("DimmedFactor_", (float*)&m_ScanlineBuffer.DimmedFactor_, 0, 1);
                ImGui::SetItemTooltip("How 'dimmed' the darker screen bands are");
                ImGui::SliderFloat("transferPower_", (float*)&m_ScanlineBuffer.transferPower_, 0, 1);
                ImGui::SetItemTooltip("The 'brightness' of the lighter screen bands");

                enum Element { scanline_H, scanline_C, scanline_V, scanline_COUNT };
                static int scanline_Type = scanline_H;
                const char* scanline_Type_names[scanline_COUNT] = { "Horozontial", "Cross/Matrix", "Vertical" };
                const char* scanline_Type_name = (scanline_Type >= 0 && scanline_Type < scanline_COUNT) ? scanline_Type_names[scanline_Type] : "Unknown";
                ImGui::SliderInt("Scanline Type/Orientation", &scanline_Type, 0, scanline_COUNT - 1, scanline_Type_name); // Use ImGuiSliderFlags_NoInput
                ImGui::SetItemTooltip("The Type/Orientation of the screen bands/lines (Horozontial,Vertical,Cross/Matrix)");

                switch (scanline_Type)
                {
                case scanline_H:
                        m_ScanlineBuffer.vertical_ = 0;
                        //vertical_ = 0;
                        break;
                case scanline_C:
                        m_ScanlineBuffer.vertical_ = 0.5;
                        //vertical_ = 0.5;
                        break;
                case scanline_V:
                        m_ScanlineBuffer.vertical_ = 1;
                        //vertical_ = 1;
                        break;
                default:
                        scanline_Type = scanline_H;
                        break;
                }

                ImGui::TreePop();
        }
        ImGui::SetItemTooltip("Simulates the appearance of lines running across the screen, for a retro or vintage CRT monitor look by adding subtle dark bands to an image.");

        if (ImGui::TreeNode("Vignette")) {

                ImGui::SliderFloat("innerRadius", (float*)&m_VignettingBuffer.innerRadius, 0, 1);
                ImGui::SetItemTooltip("where the gradient of the vingette ends");
                ImGui::SliderFloat("outerRadius", (float*)&m_VignettingBuffer.outerRadius, 0, 1);
                ImGui::SetItemTooltip("where the gradient of the vingette starts");
                ImGui::SliderFloat("opacity", (float*)&m_VignettingBuffer.opacity, 0, 1);
                ImGui::SetItemTooltip("The opacity/transparency of the vignette boarder");
                ImGui::TreePop();
        }
        ImGui::SetItemTooltip("Darkens the edges of an image while keeping the center bright, drawing attention to the focal point and creating a more dramatic or cinematic atmosphere.");
```

```
if (ImGui::TreeNode("Posturize")) {

        ImGui::SliderFloat("Step 1", (float*)&m_PosterizeBuffer.step_areas.x, 0, 10);
        ImGui::SetItemTooltip("The first 'tonal level'/ band .");

        ImGui::SliderFloat("Step 2", (float*)&m_PosterizeBuffer.step_areas.y, 0, 10);
        ImGui::SetItemTooltip("The second 'tonal level'/ band .");

        ImGui::SliderFloat("Step 3", (float*)&m_PosterizeBuffer.step_areas.z, 0, 10);
        ImGui::SetItemTooltip("The third 'tonal level'/ band .");
        //number of tonal levels
        ImGui::TreePop();
}
ImGui::SetItemTooltip("Reduces the number of colors in an image, creating a stylized, high-contrast look with distinct color bands or gradients.");
//might need to change des

if (ImGui::TreeNode("Bloom")) {

        ImGui::Text("Bloom 1");

        ImGui::SliderFloat("ampFactor", (float*)&m_BloomBuffer.ampFactor, 0, 1);
        ImGui::SetItemTooltip("How bright the brightest areas of the screen are");
        ImGui::SliderFloat("width", (float*)&m_BloomBuffer.width, 0, 10);
        ImGui::SetItemTooltip("How wide the blur radius will be");
        ImGui::SliderFloat("radiusSteps", (float*)&m_BloomBuffer.radiusSteps, 0, 10);
        ImGui::SetItemTooltip("---");
        ImGui::SliderFloat("angleSteps", (float*)&m_BloomBuffer.angleSteps, 0, 10);
        ImGui::SetItemTooltip("---");

        ImGui::Text("Bloom 2");//this is more a threshold thing at this point

        ImGui::SliderFloat("ampFactor ", (float*)&m_Bloom2Buffer.ampFactor, 0, 1);
        ImGui::SetItemTooltip("How bright the brightest areas of the screen are");
        ImGui::SliderFloat("threshold", (float*)&m_Bloom2Buffer.threshold, 0, 1);
        ImGui::SetItemTooltip("This decides what the 'brightesr' areas of the view are.");//should i describe the whole colour number thing or would that be a bit much?

        ImGui::TreePop();
}
ImGui::SetItemTooltip("creates a glowing haze around bright areas of an image, enhancing light sources.");
```

```
if (ImGui::TreeNode("Blur")) {

	ImGui::SliderFloat("Luminance correcter", (float*)&m_BlurGauBuffer.Luminance, 0, 1);
	ImGui::SetItemTooltip("This correct/ changes the brightness of the produced effect.");

	ImGui::Checkbox("ALL weight linked", &link_weight);
	ImGui::SetItemTooltip("If yes the blur will be even throughout the given veiw");
	if (link_weight == true) {

		ImGui::SliderFloat("weight_all", (float*)&weight_all, 0, 0.4);
		ImGui::SetItemTooltip("--");
		for (int i = 0; i < 5; i++) {
			m_BlurGauBuffer.weight[i] = weight_all;
		}
	}
	else {
		if (ImGui::TreeNode("weight more")) {

			ImGui::SliderFloat("weight_0", (float*)&m_BlurGauBuffer.weight[0], 0, 0.4);
			ImGui::SetItemTooltip("Center of the point being blurred/ the orign");
			ImGui::SliderFloat("weight_1", (float*)&m_BlurGauBuffer.weight[1], 0, 0.4);
			ImGui::SetItemTooltip("Middle point of blur, close to the orign of the blur");
			ImGui::SliderFloat("weight_2", (float*)&m_BlurGauBuffer.weight[2], 0, 0.4);
			ImGui::SetItemTooltip("Middle point of blur");
			ImGui::SliderFloat("weight_3", (float*)&m_BlurGauBuffer.weight[3], 0, 0.4);
			ImGui::SetItemTooltip("Middle point of blur, close to the edge of the blur");
			ImGui::SliderFloat("weight_4", (float*)&m_BlurGauBuffer.weight[4], 0, 0.4);
			ImGui::SetItemTooltip("Edge of the point being blurred");

			ImGui::TreePop();
		}
	}
```

```
        ImGui::Checkbox("ALL multipliers linked", &link_Multiplier);// maybe i shouldn't include this
        ImGui::SetItemTooltip("If yes the blur will not be very visable");
        if (link_Multiplier == true) {

                ImGui::SliderFloat("multi_all", (float*)&multi_all, 0, 10);
                ImGui::SetItemTooltip("The value used to multiply all blur points");
                for (int i = 0; i < 5; i++) {
                        m_BlurGauBuffer.texscrMultiplier[i] = multi_all;
                }
        }
        else {
                if (ImGui::TreeNode("multi more")) {

                        ImGui::SliderFloat("multi_0", (float*)&m_BlurGauBuffer.texscrMultiplier[0], 0, 1);
                        ImGui::SetItemTooltip("The value used to multiply the center of the point being blurred/ the origin");
                        ImGui::SliderFloat("multi_1", (float*)&m_BlurGauBuffer.texscrMultiplier[1], 0, 1);
                        ImGui::SetItemTooltip("The value used to multiply the middle point of blur, close to the origin of the blur");
                        ImGui::SliderFloat("multi_2", (float*)&m_BlurGauBuffer.texscrMultiplier[2], 0, 1);
                        ImGui::SetItemTooltip("The value used to multiply the middle point of blur");
                        ImGui::SliderFloat("multi_3", (float*)&m_BlurGauBuffer.texscrMultiplier[3], 0, 1);
                        ImGui::SetItemTooltip("The value used to multiply the middle point of blur, close to the edge of the blur");
                        ImGui::SliderFloat("multi_4", (float*)&m_BlurGauBuffer.texscrMultiplier[4], 0, 1);
                        ImGui::SetItemTooltip("The value used to multiply the edge of the point being blurred");

                        ImGui::TreePop();
                }
        }

        ImGui::TreePop();
}
ImGui::SetItemTooltip("Smooths the image by averaging pixel values based on a Gaussian distribution, creating a soft, evenly blurred effect.");
```

```
if (ImGui::TreeNode("ChromaTruePara")) {

        ImGui::SliderFloat("aberrationFactor", (float*)&m_ChromaTrueBuffer.aberrationFactor, 0, 1);
        ImGui::SetItemTooltip("The chromatic aberration layers will be linked and the values will be mirrored for the r,y and b values.");

        ImGui::Checkbox("is XY linked", &link_XY);
        ImGui::SetItemTooltip("Whether the chromatic aberration x-axis and y-axis are linked / 'mirrored' with each other.");//

        if (link_XY == true) {

                ImGui::SliderFloat("aberrationFactor_linked_XY", (float*)&aberrationFactor_XY, 0, 2);
                ImGui::SetItemTooltip("the diffrence betewn the r,g, and b values in terms of the x and y axis ");

                m_ChromaTrueBuffer.aberrationFactor_y = aberrationFactor_XY;
                m_ChromaTrueBuffer.aberrationFactor_x = aberrationFactor_XY;
        }
        else {
                if (ImGui::TreeNode("aberrationFactor more")) {

                        ImGui::SliderFloat("aberrationFactor_x", (float*)&m_ChromaTrueBuffer.aberrationFactor_x, 0, 2);
                        ImGui::SetItemTooltip("The diffrence betewn the r,g, and b values in terms of the x axis ");
                        ImGui::SliderFloat("aberrationFactor_y", (float*)&m_ChromaTrueBuffer.aberrationFactor_y, 0, 2);
                        ImGui::SetItemTooltip("The diffrence betewn the r,g, and b values in terms of they axis ");

                        ImGui::TreePop();
                }
        }
        ImGui::TreePop();
}
ImGui::SetItemTooltip("Simulates color fringing by slightly offsetting the red, green, and blue channels along the edges of objects, similar to what 3D movies look like");
```

```
if (ImGui::TreeNode("ChromaStylizedPara")) {

        chromaticWeights1[0] = { m_ChromaStylizedBuffer.chromaticWeights1.x };
        chromaticWeights1[1] = { m_ChromaStylizedBuffer.chromaticWeights1.y, };
        chromaticWeights1[3] = { m_ChromaStylizedBuffer.chromaticWeights1.a };

        chromaticWeights2[0] = { m_ChromaStylizedBuffer.chromaticWeights2.x };
        chromaticWeights2[1] = { m_ChromaStylizedBuffer.chromaticWeights2.y, };
        chromaticWeights2[3] = { m_ChromaStylizedBuffer.chromaticWeights2.a };

        ImGui::ColorEdit4("ChromaColour1 ", chromaticWeights1);
        ImGui::SetItemTooltip("The colour used for the first chromatic aberration layer");

        ImGui::ColorEdit4("ChromaColour2 ", chromaticWeights2);
        ImGui::SetItemTooltip("The colour used for the second chromatic aberration layer");

        m_ChromaStylizedBuffer.chromaticWeights1 = float4(chromaticWeights1[0], chromaticWeights1[1], 0.f, chromaticWeights1[3]);
        m_ChromaStylizedBuffer.chromaticWeights2 = float4(chromaticWeights2[0], chromaticWeights2[1], 0.f, chromaticWeights2[3]);

        ImGui::Checkbox("is chromaticOffset linked", &offsetlinked);
        ImGui::SetItemTooltip("The chromatic aberration layers will be linked and the values will be mirrored for both layers");
        ImGui::Checkbox("Is XY linked", &XYlinked);
        ImGui::SetItemTooltip("Whether the chromatic aberration x-axis and y-axis ar equal/mirrored with one another.");

        chromaticOffset_XY.x = m_ChromaStylizedBuffer.chromaticOffset1.x;
        chromaticOffset_XY.y = m_ChromaStylizedBuffer.chromaticOffset1.y;

        if (offsetlinked == true) {

                if (ImGui::TreeNode("chromaticOffset_XY")) {

                        if (XYlinked == true) {
                                ImGui::SliderFloat("chromaticOffset_XY", (float*)&chromaticOffset_XY.x, -0.1, 0.1);

                                chromaticOffset_XY.y = chromaticOffset_XY.x;
                                m_ChromaStylizedBuffer.chromaticOffset1.x = chromaticOffset_XY.x;
                                m_ChromaStylizedBuffer.chromaticOffset1.y = chromaticOffset_XY.y;

                                m_ChromaStylizedBuffer.chromaticOffset2.x = -chromaticOffset_XY.x;
                                m_ChromaStylizedBuffer.chromaticOffset2.y = -chromaticOffset_XY.y;
                        }
                        else {
                                ImGui::SliderFloat("chromaticOffset_X", (float*)&chromaticOffset_XY.x, -0.1, 0.1);
                                ImGui::SliderFloat("chromaticOffset_Y", (float*)&chromaticOffset_XY.y, -0.1, 0.1);

                                m_ChromaStylizedBuffer.chromaticOffset1.x = chromaticOffset_XY.x;
                                m_ChromaStylizedBuffer.chromaticOffset1.y = chromaticOffset_XY.y;

                                m_ChromaStylizedBuffer.chromaticOffset2.x = chromaticOffset_XY.x;
                                m_ChromaStylizedBuffer.chromaticOffset2.y = chromaticOffset_XY.y;
                        }

                        ImGui::TreePop();
                }
        }
```

```
        else {
                if (ImGui::TreeNode("chromaticOffset_XY")) {

                        if (XYlinked == true) {
                                ImGui::SliderFloat("chromaticOffset_1_XY", (float*)&m_ChromaStylizedBuffer.chromaticOffset1.x, -0.1, 0.1);
                                ImGui::SliderFloat("chromaticOffset_2_XY", (float*)&m_ChromaStylizedBuffer.chromaticOffset2.x, -0.1, 0.1);

                                m_ChromaStylizedBuffer.chromaticOffset1.y = m_ChromaStylizedBuffer.chromaticOffset1.x;
                                m_ChromaStylizedBuffer.chromaticOffset2.y = m_ChromaStylizedBuffer.chromaticOffset2.x;
                        }
                        else {
                                ImGui::SliderFloat("chromaticOffset1_X", (float*)&m_ChromaStylizedBuffer.chromaticOffset1.x, -0.1, 0.1);
                                ImGui::SliderFloat("chromaticOffset1_Y", (float*)&m_ChromaStylizedBuffer.chromaticOffset1.y, -0.1, 0.1);
                                ImGui::SliderFloat("chromaticOffset2_X", (float*)&m_ChromaStylizedBuffer.chromaticOffset2.x, -0.1, 0.1);
                                ImGui::SliderFloat("chromaticOffset2_Y", (float*)&m_ChromaStylizedBuffer.chromaticOffset2.y, -0.1, 0.1);
                        }
                        ImGui::TreePop();
                }
        }

        ImGui::TreePop();
}
ImGui::SetItemTooltip("Similar to true Chromatic aberation, this effect mimics the fringing whilst having the colours used customiseable giving a unique look.");
```

```
if (ImGui::TreeNode("colour Grading")) {

        startColor_CG[0] = { m_ColourGradBuffer.startColor.x };
        startColor_CG[1] = { m_ColourGradBuffer.startColor.y };
        startColor_CG[2] = { m_ColourGradBuffer.startColor.z };

        endColor_CG[0] = { m_ColourGradBuffer.endColor.x };
        endColor_CG[1] = { m_ColourGradBuffer.endColor.y };
        endColor_CG[2] = { m_ColourGradBuffer.endColor.z };

        ImGui::ColorEdit3("startColor ", startColor_CG);
        ImGui::SetItemTooltip("Start colour sets the baseline or beginning of a colour effect");
        //it's the colour at the "beginning" of the tonal range, often used to tint the shadows or define the base tone of the image.
        ImGui::ColorEdit3("endColor ", endColor_CG);
        ImGui::SetItemTooltip("The end colour replaces the brightest tones (highlights) in the image.");
        //The end colour replaces the brightest tones (highlights) in the image.
        //Start colour = what shadows become
        //End colour = what highlights become

        m_ColourGradBuffer.startColor = float3(startColor_CG[0], startColor_CG[1], startColor_CG[2]);
        m_ColourGradBuffer.endColor = float3(endColor_CG[0], endColor_CG[1], endColor_CG[2]);


        ImGui::TreePop();
}
ImGui::SetItemTooltip("Adjusts the colours and tones of an image to create a specific mood or aesthetic, enhancing visual storytelling through subtle shifts in hue, saturation, and contrast.");
```

```
if (ImGui::TreeNode("Embossed")) {

        bool greyScale = m_EmbossBuffer.greyScale;
        ImGui::Checkbox("greyScale", &greyScale);
        m_EmbossBuffer.greyScale = greyScale;

        ImGui::SliderFloat("Embossed width", (float*)&m_EmbossBuffer.width_, 0.9, 1);
        ImGui::SetItemTooltip("How far along the x-axis the textures that make up the embosed effects are");

        ImGui::SliderFloat("Embossed height", (float*)&m_EmbossBuffer.height_, 0.9, 1);
        ImGui::SetItemTooltip("How far along the u-axis the textures that make up the embosed effects are");

        ImGui::TreePop();
}
ImGui::SetItemTooltip("Creates the illusion of raised or recessed surfaces on an image, adding depth and texture by highlighting edges with a contrast of light and shadow.");
```

```
if (ImGui::TreeNode("Glass Panel")) {

        m_GlassBuffer;

        ImGui::Checkbox("is pixle height and width linked", &panelinked);
        ImGui::SetItemTooltip("This makes the individual 'pannels' squares with the hight equaling the width");

        if (panelinked == true) {
                ImGui::SliderFloat("pannel size", (float*)&m_GlassBuffer.panelSize.x, 0, 20);
                ImGui::SetItemTooltip("The higher the number the smaller the square texture of the 'glass'.");
                m_GlassBuffer.panelSize.y = m_GlassBuffer.panelSize.x;
        }
        else {

                if (ImGui::TreeNode("Square Texture size  more")) {

                        ImGui::SliderFloat("pannel width", (float*)&m_GlassBuffer.panelSize.x, 0, 20);
                        ImGui::SetItemTooltip("How wide the individual pannel textures are");
                        ImGui::SliderFloat("pannel height", (float*)&m_GlassBuffer.panelSize.y, 0, 20);
                        ImGui::SetItemTooltip("How high/tall the individual pannel textures are");
                        ImGui::TreePop();
                }
        }

        ImGui::TreePop();
}
ImGui::SetItemTooltip("simulates the appearance of looking through a panel of squared textured glass");
```

```
        if (ImGui::TreeNode("Pixel")) {


                m_PixelBuffer.pixelSize;

                ImGui::Checkbox("Is pixle height and width linked", &pixelinked);
                ImGui::SetItemTooltip("This makes the individual pixels squares with the hight equaling the width");

                if (pixelinked == true) {
                        ImGui::SliderFloat("pixelSize", (float*)&m_PixelBuffer.pixelSize.x, 50, 200);
                        //ImGui::SetItemTooltip("the higher the number the more pixles are 'added'");
                        ImGui::SetItemTooltip("The higher the number the smaller the pixles become or the higher the 'resolution' becomes");
                        m_PixelBuffer.pixelSize.y = m_PixelBuffer.pixelSize.x;
                }
                else {

                        if (ImGui::TreeNode("pixelSize more")) {

                                ImGui::SliderFloat("pixel width", (float*)&m_PixelBuffer.pixelSize.x, 50, 200);
                                ImGui::SetItemTooltip("How wide the individual pixles are");
                                ImGui::SliderFloat("pixel height", (float*)&m_PixelBuffer.pixelSize.y, 50, 200);
                                ImGui::SetItemTooltip("How high/tall the individual pixles are");

                                ImGui::TreePop();
                        }
                }

                ImGui::TreePop();
        }
        ImGui::SetItemTooltip("Turns given image into a pixelated version by //");
}
```

///

```
void PennyRender::PennyBoardIMGUI_savedEffectsTab()
{
	PennyBoardIMGUI_PreviewScreen();

	ImGui::Separator();

	ImGui::SeparatorText("effect list");
	ImGui::SetItemTooltip("where all saved / pre-built effects are stored");

	if (ImGui::TreeNode("pre - built effect list")) //not currently implemented
	{
		//ImGui::SeparatorText("pre-built effect list");

		ImGuiIO& io = ImGui::GetIO();
		ImTextureID my_tex_id = io.Fonts->TexID;

		float my_tex_w = (float)1366 / previewscreenSize;
		float my_tex_h = (float)768 / previewscreenSize;

		{
			static bool use_text_color_for_tint = false;

			ImVec2 uv_min = ImVec2(0.0f, 0.0f);                 // Top-left
			ImVec2 uv_max = ImVec2(1.0f, 1.0f);                 // Lower-right
			ImVec4 tint_col = use_text_color_for_tint ? ImGui::GetStyleColorVec4(ImGuiCol_Text) : ImVec4(1.0f, 1.0f, 1.0f, 1.0f);   // No tint
			ImVec4 border_col = ImGui::GetStyleColorVec4(ImGuiCol_Border);

			ImGui::NewLine(0);
			{
				ImGui::Image(my_tex_id, ImVec2(my_tex_w / 2, my_tex_h / 2), uv_min, uv_max, tint_col, border_col);
				ImGui::SetItemTooltip("image of the pre built effect, to show the user a preview of said effect. ");

				ImGui::SameLine();
				ImGui::Text("effect name \n" "list of effects active: \n  [] \n  []");
				ImGui::SetItemTooltip("this is where the name of the effect is displayed,\n and currently show what effects were used to make this effect");

				ImGui::SameLine();
				ImGui::Button("Apply");
				ImGui::SetItemTooltip("this will apply your saved effect to the proper screen not the preview screen");

				ImGui::SameLine();
				if (ImGui::TreeNode("more info")) {

					ImGui::SameLine();
					ImGui::Text("\n""info");

					ImGui::SameLine();
					ImGui::Text("\n""list of values: \n - [] \n  []");

					ImGui::TreePop();
				}
				ImGui::SetItemTooltip("Show you a more INDEPTH review of the effects and values used to produce the pre-built result");
			}
			ImGui::PopID();
```

```
            ImGui::PushID(1);
            {
                    ImGui::Image(my_tex_id, ImVec2(my_tex_w / 2, my_tex_h / 2), uv_min, uv_max, tint_col, border_col);
                    ImGui::SameLine();
                    ImGui::Text("Effect name \n" "list of effects active: \n - [] \n -[]");
                    ImGui::SameLine();
                    ImGui::Button("Apply ");

                    ImGui::SameLine();
                    if (ImGui::TreeNode("more info")) {

                            ImGui::SameLine();
                            ImGui::Text("\n""info");

                            ImGui::SameLine();
                            ImGui::Text("\n""list of values: \n - [] \n -[]");

                            ImGui::TreePop();
                    }
            }
            ImGui::PopID();

    }

    ImGui::TreePop();
}
ImGui::SetItemTooltip("where all pre-built effects are stored");
```

```
    if (ImGui::TreeNode("user saved effect list")) // saved stuff isnt currently working
    {
        //ImGui::SeparatorText("user saved effect list");
        {
            ImGui::Text("-Example Effect-");
            ImGui::SetItemTooltip("this is where the name of the effect you have save is displayed, the name basically act like an ID for the effect");
            ImGui::SameLine();
            ImGui::Button("APPLY");
            ImGui::SetItemTooltip("this will apply your saved effect to the proper screen not the preview screen");
            //ImGui::SameLine();//add if i have time
            //ImGui::Button("RENAME"); // make rename function
            //ImGui::SetItemTooltip("This will bring a pop-up that allows you to rename your effect");
            ImGui::SameLine();
            ImGui::Button("DELETE"); // make delete function
            ImGui::SetItemTooltip("This will delete currently stored effect, it is recoomended to only store X effects for memory and effecency sake");
            ImGui::Text("- -");
        }

        //ImGui::SeparatorText("user saved effect list");
        if (savedEffect_List.size() != 0) {
            for (int i = 0; i <= savedEffect_List.size(); i++)
            {
                ImGui::Text(savedEffect_List.data()->SavedName);
                ImGui::SetItemTooltip("this is where the name of the effect you have save is displayed, the name basically act like an ID for the effect");
                ImGui::SameLine();
                ImGui::PushID(i); {

                    if (ImGui::Button("APPLY")) {
                        //ApplySavedEffect(savedEffect_List[i]);//this breaks everything . not sure why, come back later if time permits

                    }
                    ImGui::SetItemTooltip("this will apply your saved effect to the proper screen not the preview screen");
                    //ImGui::SameLine();//add if i have time
                    //ImGui::Button("RENAME"); // make rename function
                    //ImGui::SetItemTooltip("This will bring a pop-up that allows you to rename your effect");
                    ImGui::SameLine();
                    if (ImGui::Button("DELETE")) {
                        removeEffect(savedEffect_List, i);
                    }
                    ImGui::SetItemTooltip("This will delete currently stored effect, it is recoomended to only store X effects for memory and effecency sake");
                    ImGui::Text("- -");


                    ImGui::PopID();
                }
            }

        }


        ImGui::TreePop();
    }
    ImGui::SetItemTooltip("where all user saved/made effects are stored");
}
```

```
void PennyRender::PennyBoardIMGUI_toolCustomTab()
{

        ImGui::ShowStyleEditor();

}
```

```
void PennyRender::PennyBoardIMGUI() {

        ImGui::Begin("Penny_board Graphics (shader toggle tool)");//creates new window

        ImGui::Separator();
        if (ImGui::BeginTabBar("##tabs", ImGuiTabBarFlags_None))
        {
                if (ImGui::BeginTabItem("Editor")) {

                        PennyBoardIMGUI_editorTab();

                        ImGui::EndTabItem();

                }
                if (ImGui::BeginTabItem("Saved Effects list")) {

                        PennyBoardIMGUI_savedEffectsTab();

                        ImGui::EndTabItem();

                }
                if (ImGui::BeginTabItem("Tool Cus")) {

                        PennyBoardIMGUI_toolCustomTab();

                        ImGui::EndTabItem();

                }

                ImGui::EndTabBar();
        }

        //ImGui::Separator();

        //ImGui::ShowDemoWindow();

        //ImGui::ShowStyleEditor();

        //ImGui::ShowStyleEditor(ImGuiStyle* ref);

        ImGui::End();//ends that window

}
```

Renderer Code

Popping open the engine/hood // under the hood

```
#pragma once

//#include "Skateboard/Renderer/GraphicsContext.h"
#include "Skateboard/Graphics/RHI/ResourceFactory.h"
#include "Skateboard/Graphics/RHI/RenderCommand.h"
#include "Skateboard/Renderers/Renderer.h"

//#include "Penny-Board/TestRenderer.h"
using namespace Skateboard;

namespace pen
{
```

Buffer stuff

Buffer Struct

```
struct ScreenBuffer //ScreenPara            //b2
{
float2 screenSize_;
float2 padding; //edit later
//ScreenBuffer() { padding = float2(1200, 800); screenSize_ = float2(1200, 800);}
ScreenBuffer() { padding = float2(1.f, 1.f); screenSize_ = float2(1366, 768); }
};
```

Screen buffer passes through the screen size to any potential shader if initialised

```
struct ScanlineBuffer//scanlinePara //b3 { float2 screenSize_; float LineThickness_; float
DimmedFactor_;

  float transferPower_;
    float vertical_;
    int PushData_instanceNo;
    float padding;


    ScanlineBuffer() { PushData_instanceNo = 0; screenSize_ = float2(1366, 768);
LineThickness_ = 3.f; DimmedFactor_ = 0.5; transferPower_ = 0; vertical_ = 0.5; }


    ScanlineBuffer(float linethickness) :
```

```
        ScanlineBuffer() {
        LineThickness_ = linethickness;
    }
    ScanlineBuffer(float linethickness, float dimmedFactor) :
        ScanlineBuffer(linethickness) {
        DimmedFactor_ = dimmedFactor;
    }
    ScanlineBuffer(float linethickness, float dimmedFactor, float transferPower) :
        ScanlineBuffer(linethickness, dimmedFactor) {
        transferPower_ = transferPower;
    }
    ScanlineBuffer(float linethickness, float dimmedFactor, float transferPower, float
vertical) :
        ScanlineBuffer(linethickness, dimmedFactor, transferPower) {
        vertical_ = vertical;
    }

};
```

```
struct VignettingBuffer//vignettingPara //b4 { float2 screenSize_; float2 padding;

  float innerRadius;
    float outerRadius;
    float opacity;
    int PushData_instanceNo;


    VignettingBuffer() { PushData_instanceNo = 0; screenSize_ = float2(1366, 768); innerRadius
= 0.f; outerRadius = 0.4; opacity = 0.8f; padding = float2(0,0); }
};
```

```
struct PosterizeBuffer//posterizePara        //b5
{
    float3 step_areas;
    int PushData_instanceNo;
    PosterizeBuffer() { PushData_instanceNo = 0; step_areas = float3(3.8f, 1.4f, 0.2f); }
```

```
};
```

```
struct BloomBuffer //bloomPara
{
int PushData_instanceNo;
int width;
float angleSteps;
float radiusSteps;

float ampFactor;
float3 padding;

BloomBuffer() { PushData_instanceNo = 0; width = 20; angleSteps = 20; radiusSteps = 20;
ampFactor = 1.25; }
};
```

```
struct Bloom2Buffer //bloom2Para
{
int PushData_instanceNo;
float ampFactor;
float threshold;
float padding0;

Bloom2Buffer() { PushData_instanceNo = 0; ampFactor = 1; threshold = 0.24; padding0 = 0; }
};
```

```
struct BlurGauBuffer //blurGauPara  //
{
float Luminance;
float2 screenSize_;
float padding0;

//float weight[5];
//float padding1[3]; // padding to align next array
//float texscrMultiplier[5];
//float padding2[3]; // padding for alignment
```

```
float weight[8]; //last values are padding
float texscrMultiplier[8];//last values ar padding


BlurGauBuffer() {
for (int i = 0; i < 8; i++) { weight[i] = 0.2; float placeholder = i; texscrMultiplier[i] =
placeholder / 4; }
padding0 = 0; screenSize_ = float2(1366, 768); Luminance = 0.4;
}
};
```

```
struct ChromaTrueBuffer //ChromaTruePara { int PushData_instanceNo; float2 screenSize_; float
aberrationFactor;

  float aberrationFactor_x;
    float aberrationFactor_y;
    float2 padding; // padding for alignment


    ChromaTrueBuffer() { PushData_instanceNo = 0; screenSize_ = float2(1366, 768); padding =
float2(0,0); aberrationFactor = 0.009; aberrationFactor_x = 0.5; aberrationFactor_y = 0.5; }
};
```

```
struct ChromaStylizedBuffer //ChromaStylizedPara
{                                                    //might not need arrays - matters how
customizable i want it

int PushData_instanceNo;
float3 padding0;

float4 chromaticWeights1;
float4 chromaticWeights2;
float4 chromaticOffset1;
float4 chromaticOffset2;

ChromaStylizedBuffer() {

//for (int i = 0; i < 1; i++)
```

```
{
chromaticOffset1 = float4(0.01, 0.01, 13, 0);
chromaticOffset2 = float4(-0.01, -0.01, 13, 0);
chromaticWeights1 = float4(0.2, 0.2f, 0.f, 0.25);
chromaticWeights2 = float4(0.2f, 0.f, 0.2, 0.25);
}
padding0 = float3(0,0,0);
PushData_instanceNo =0;
}
};
```

```
struct ColourGradBuffer //colourGradPara
{
int PushData_instanceNo;
float3 padding0;

float3 startColor;
float padding1;

float3 endColor;
float padding2;


ColourGradBuffer() { startColor = float3(1, 1, 1); endColor = float3(1.2, 0.01, 0.51);
PushData_instanceNo = 0;
}
};
```

```
struct EmbossBuffer //EmbossPara {

  int PushData_instanceNo;
    float2 screenSize_;
    float padding0;


    int greyScale;
    float width_;
    float height_;
    float padding1;
```

```
    EmbossBuffer() { PushData_instanceNo = 0; screenSize_ = float2(1366, 768); greyScale =
true; width_ = 400; height_ = 400; padding0 = 0; padding1 = 0; }
};
```

```
struct GlassBuffer //glassPara { int PushData_instanceNo; float2 screenSize_; float2
panelSize;

  float2 padding; // for alignment

    GlassBuffer() { PushData_instanceNo = 0; screenSize_ = float2(1366, 768); panelSize =
float2(10, 10); padding = float2(10, 10); }
};
```

```
struct PixelBuffer //pixelPara { int PushData_instanceNo; float2 screenSize_; float2
pixelSize;

  float2 padding; // for alignment

    PixelBuffer() { PushData_instanceNo = 0; screenSize_ = float2(1366, 768); pixelSize =
float2(200, 200); padding = float2(200, 200); }
};
```

```
struct SavedEffect {
//string SavedName;
const char* SavedName;
ShaderToggles EffectInUse;

ScreenBuffer screen_buff;
ScanlineBuffer scanline_buff;
VignettingBuffer vignet_buff;
PosterizeBuffer posterize_buff;
```

```
BloomBuffer bloom_buff;
Bloom2Buffer bloom2_buff;
BlurGauBuffer blurGauBuffer_buff;
ChromaTrueBuffer  chromaTrue_buff;
ChromaStylizedBuffer chromaStylized_buff;
ColourGradBuffer colourGrad_buff;
EmbossBuffer emboss_buff;
GlassBuffer glass_buff;
PixelBuffer pixel_buff;

};
```

```
struct Vertex
{
float3 Position;
float3 Colour;
float2 UV;
float3 Normal;

//default colour is white
Vertex() : Position(0, 0, 0), Colour(0, 0, 0), UV(0, 0), Normal(0, 0, 0) {}
Vertex(float3 pos) : Vertex() { Position = pos; }
Vertex(float3 pos, float3 col) : Vertex(pos) { Colour = col; }
Vertex(float3 pos, float3 col, float2 uv) : Vertex(pos, col) { UV = uv; }
Vertex(float3 pos, float3 col, float2 uv, float3 normal) : Vertex(pos, col, uv) { Normal = normal; }

static BufferLayout VertexLayout()
{
return {
{ POSITION,     ShaderDataType_::Float3 },
{ COLOUR,       ShaderDataType_::Float3 },
{ TEXCOORD,     ShaderDataType_::Float2 },
{ NORMAL,       ShaderDataType_::Float3 },
};
}
};
```

```
enum LightType : uint32_t
{
LightDirectional = 0,
```

```
LightPoint = 1,
LightSpot = 2,
};
```

```
struct InstanceData
{
matrix World;
float4 ColourScale;

int TextureIndex;

float4 SpecularColor;
float SpecularPower;
float SpecularWeight;

InstanceData() { World = glm::identity<glm::mat4x4>(); ColourScale = float4(1, 1, 1, 1); SpecularColor =
float4(1, 1, 1, 1);  TextureIndex = 1; SpecularPower = 1.f; SpecularWeight = 1; }
};
```

```
struct Light
{
public:
float4 DiffuseColour;
float4 Attenuation;

float3 LightPosition;
float InnerCone;

float3 LightDirection;
float OuterCone;

private:
float2 Padding;
public:
float FalloffPower;
uint32_t LightType;

static float4 AttenuationDefaults() { return { 0.05f, 0.01f, 0.001f, 100.f }; }
};
```

```
struct  FrameData
{
```

```
glm::mat4x4 ViewMatrix;
glm::mat4x4 ProjectionMatrix;
};
```

```
struct Frame  //change to be more like screen fram in raytracer
{
FrameData Matrices;

matrix CameraMatrix;
uint32_t LightCount;
float3 AmbientLight;
};
```

```
static SamplerDesc AnisotropicSampler(uint8_t anisotropy, SamplerMode_ U, SamplerMode_ V)
{
    return
    {
        .Filter = SamplerFilter_::SamplerFilter_Anisotropic,
        .ModeU = U,
        .ModeV = V,
        .ModeW = V,
        .MipMapLevelOffset = 0.f,
        .MipMapMinSampleLevel = 0.f,
        .MipMapMaxSampleLevel = 10.f,
        .MaxAnisotropy = anisotropy,    // Valid range 1 - 16 -> uint32_t cause padding anyways
        .ComparisonFunction = SamplerComparisonFunction_Less_Equal,
        .BorderColour = SamplerBorderColour_TransparentBlack,
        .Flags = 0,
    };
}
```

```
static SamplerDesc LinearSampler(SamplerMode_ U, SamplerMode_ V)
{
    auto sampler = SamplerDesc::InitAsDefaultTextureSampler();
    sampler.ModeU = U;
    sampler.ModeV = V;
    return sampler;
}
```

```
static SamplerDesc PointSampler(SamplerMode_ U, SamplerMode_ V)
{
    auto sampler = SamplerDesc::InitAsDefaultTextureSampler();
    sampler.Filter = SamplerFilter_::SamplerFilter_Comaprison_Min_Mag_Mip_Point;
    sampler.ModeU = U;
    sampler.ModeV = V;
    return sampler;
}
```

```
//post-processing pipelines + "toggles"
enum ShaderToggles : uint16_t
{
    //-- post-proccessing effects // shaders
    PIXELATE_ = 0,
    POSTERIZE_ = 1,
    SCANLINE_ = 2,
    VIGNETTE_ = 3,


    BLOOM_ = 4,
    CHROMABER_TRUE_ = 5,
    CHROMABER_STYLE_ = 6,


    COLOURGRAD_ = 7,
    GLASS_ = 8,
    EMBOS_ = 9,
    BLUR_GAU_ = 10,
    POSTPROSSESSING_ = 11
    //TEST_ = 1 << 10
    //--
};
```

```
ENUM_FLAG_OPERATORS(ShaderToggles);
```

```
constexpr static ShaderToggles DEFAULT_STATE_ = POSTPROSSESSING_;
constexpr static uint16_t EFFECT_PERMUTATIONS = 12U;//(PIXELATE_ | POSTERIZE_ | SCANLINE_ | VIGNETTE_ |
BLOOM_ | CHROMABER_TRUE_ | CHROMABER_STYLE_ | COLOURGRAD_ | GLASS_ | EMBOS_ | BLUR_GAU_ |
POSTPROSSESSING_) + 1;
```

Setting up the buffers

```
//normal rendering

   Skateboard::PipelineRef NormalVisualizer;

   size_t DynamicBufferSize = 64 * 1024;
    size_t InstanceDataBufferSize = 64 * 1024;


    Skateboard::MultiResource<Skateboard::BufferRef> TriangleDataBuffer;
    Skateboard::MultiResource<Skateboard::BufferRef> InstanceDataBuffer;
    Skateboard::MultiResource<Skateboard::BufferRef> LightDataBuffer;


    //--------------
    // post-processing buffers


    size_t pennyBufferSize = 64 * 1024; //


    //Buffer refs
    Skateboard::MultiResource<Skateboard::BufferRef> ScreenDataBuffer;
    Skateboard::MultiResource<Skateboard::BufferRef> ScanlineDataBuffer;
    Skateboard::MultiResource<Skateboard::BufferRef> VignettingDataBuffer;
    Skateboard::MultiResource<Skateboard::BufferRef> PosterizeDataBuffer;


    Skateboard::MultiResource<Skateboard::BufferRef> BloomDataBuffer;
    Skateboard::MultiResource<Skateboard::BufferRef> Bloom2DataBuffer;
    Skateboard::MultiResource<Skateboard::BufferRef> BlurGauDataBuffer;
    Skateboard::MultiResource<Skateboard::BufferRef> ChromaTrueDataBuffer;


    Skateboard::MultiResource<Skateboard::BufferRef> ChromaStylizedDataBuffer;
    Skateboard::MultiResource<Skateboard::BufferRef> ColourGradDataBuffer;
    Skateboard::MultiResource<Skateboard::BufferRef> EmbossDataBuffer;
    Skateboard::MultiResource<Skateboard::BufferRef> GlassDataBuffer;
```

```
    Skateboard::MultiResource<Skateboard::BufferRef> pixelDataBuffer;


    // desc
    BufferViewDesc screen_desc;
    BufferViewDesc scanline_desc;
    BufferViewDesc vignetting_desc;
    BufferViewDesc posterize_desc;


    BufferViewDesc bloom_desc;
    BufferViewDesc bloom2_desc;
    BufferViewDesc blurGau_desc;
    BufferViewDesc ChromaTrue_desc;


    BufferViewDesc ChromaStylized_desc;
    BufferViewDesc colourGrad_desc;
    BufferViewDesc Emboss_desc;
    BufferViewDesc glass_desc;


    BufferViewDesc pixel_desc;


    // init buffer structs
    ScreenBuffer m_ScreenBuffer = ScreenBuffer();
    ScanlineBuffer m_ScanlineBuffer = ScanlineBuffer();
    VignettingBuffer m_VignettingBuffer = VignettingBuffer();
    PosterizeBuffer m_PosterizeBuffer = PosterizeBuffer();


    BloomBuffer    m_BloomBuffer = BloomBuffer();
    Bloom2Buffer    m_Bloom2Buffer = Bloom2Buffer();
    BlurGauBuffer m_BlurGauBuffer = BlurGauBuffer();
    ChromaTrueBuffer m_ChromaTrueBuffer = ChromaTrueBuffer();


    ChromaStylizedBuffer m_ChromaStylizedBuffer = ChromaStylizedBuffer();
    ColourGradBuffer m_ColourGradBuffer = ColourGradBuffer();
    EmbossBuffer m_EmbossBuffer = EmbossBuffer();
    GlassBuffer m_GlassBuffer = GlassBuffer();


    PixelBuffer m_PixelBuffer = PixelBuffer();


    // Pipeline default sate inti
    ShaderToggles CURRENT_STATE_ = DEFAULT_STATE_;//
    //ShaderToggles CURRENT_STATE_Pass[EFFECT_PERMUTATIONS];// DEPTH_TEST;
    //ShaderToggles CURRENT_STATE_Pass[5] = { DEFAULT_STATE_ ,DEFAULT_STATE_ ,DEFAULT_STATE_
,DEFAULT_STATE_ ,DEFAULT_STATE_ };
```

```
    //--------------
    //inline view desc
    BufferViewDesc sbvdesc;
    BufferViewDesc lightsbvdesc;
    BufferViewDesc cbvdesc;


    //resterizer
    SamplerDesc StaticSamplerDesc = SamplerDesc::InitAsDefaultTextureSampler();


    //counts offsets for dynamic data uploaded every frame
    size_t m_Offset = ROUND_UP(sizeof(FrameData), GraphicsConstants::CONSTANT_BUFFER_ALIGNMENT);


    //forked on each call with a unique data pointer/ otherwise default instance data is used
    uint32_t m_InstanceDataForks = 0;


    Frame m_Frame{ .AmbientLight = { 0.1, 0.1, 0.1} };
    FrameData m_CameraData{};


    std::vector<Light> m_Lights;


    InstanceData m_DefaultInstanceData = InstanceData();


    uint32_t m_DefaultTextureIDX = 0;


    bool m_Pipeline_dirty;
    bool m_VisualiseNormals = false;// probs delete
```

Rendering & Post-Processing Pipeline Variables Documentation
This section documents the state and buffer declarations for a post-processing rendering pipeline built with the Skateboard engine framework. It includes buffers for scene rendering, post-processing effects, light data, and dynamic frame updates.

Normal Rendering Pipeline
Skateboard::PipelineRef NormalVisualizer;

Purpose: Pipeline reference for rendering scene geometry with standard or visual debug shaders (e.g., normal visualizations).

```
-------------------------------------------------------------------------------------------------------------
-------------

Dynamic & Instance Buffers
size_t DynamicBufferSize = 64 * 1024;
size_t InstanceDataBufferSize = 64 * 1024;

DynamicBufferSize: Memory allocated for frame-dependent dynamic data (e.g., camera info, light count,
etc.).
InstanceDataBufferSize: Storage for per-instance data such as transform matrices or material properties.



-------------------------------------------------------------------------------------------------------------
-------------

Scene Buffers
Skateboard::MultiResource<Skateboard::BufferRef> TriangleDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> InstanceDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> LightDataBuffer;

TriangleDataBuffer: Stores vertex/index data or triangle-level metadata for rendering.
InstanceDataBuffer: Holds data for each instance rendered this frame.
LightDataBuffer: Contains information on dynamic and static lights (position, color, intensity).



-------------------------------------------------------------------------------------------------------------
-------------

Post-Processing Buffers
size_t pennyBufferSize = 64 * 1024;

A general-purpose allocation size used for each post-processing buffer

ach effect has its own BufferRef stored in a MultiResource (multi-frame resource safe for double/triple
buffering):



Skateboard::MultiResource<Skateboard::BufferRef> ScreenDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> ScanlineDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> VignettingDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> PosterizeDataBuffer;

//Bloom and blur type effects
Skateboard::MultiResource<Skateboard::BufferRef> BloomDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> Bloom2DataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> BlurGauDataBuffer;


// Chromatic effects
```

```
Skateboard::MultiResource<Skateboard::BufferRef> ChromaTrueDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> ChromaStylizedDataBuffer;


Skateboard::MultiResource<Skateboard::BufferRef> ColourGradDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> EmbossDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> GlassDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> pixelDataBuffer;

-------------------------------------------------------------------------------------------------------
-------------
Buffer View Descriptions
BufferViewDesc screen_desc;
BufferViewDesc scanline_desc;
...
BufferViewDesc pixel_desc;

Purpose: Describe how each buffer is viewed by the GPU (CBV, SRV, UAV, etc.).
Each effect has a corresponding descriptor that links the CPU-side buffer to a GPU-side resource binding.



-------------------------------------------------------------------------------------------------------
-------------
Buffer Struct Instances
ScreenBuffer m_ScreenBuffer;
ScanlineBuffer m_ScanlineBuffer;
...
PixelBuffer m_PixelBuffer;

These structs hold initialized per-effect data, ready to be copied to their corresponding GPU buffer
before a draw or dispatch.
Initialized once here, but typically updated each frame with relevant scene-dependent values (resolution,
animation, toggles).



-------------------------------------------------------------------------------------------------------
-------------
Pipeline State Tracking
ShaderToggles CURRENT_STATE_ = DEFAULT_STATE_;

Tracks the active shader features or combinations (toggle flags).
Could include toggles like USE_VIGNETTE, ENABLE_BLOOM, or VISUALIZE_NORMALS.



-------------------------------------------------------------------------------------------------------
-------------
Global Buffer Descriptors
BufferViewDesc sbvdesc;       // Screen-space descriptor (scene buffer)
BufferViewDesc lightsbvdesc;  // Light buffer descriptor
```

```
BufferViewDesc cbvdesc;       // Constant buffer descriptor (camera/frame)

Purpose: Low-level descriptor used when binding data to GPU pipeline.
Abstracts away root signature or descriptor table details.



-------------------------------------------------------------------------------------------------------
-------------
Sampler State
SamplerDesc StaticSamplerDesc = SamplerDesc::InitAsDefaultTextureSampler();

Default sampler used in pixel shaders.
Typically includes linear filtering and clamp addressing.

-------------------------------------------------------------------------------------------------------
-------------
Frame-Dependent Data
size_t m_Offset = ROUND_UP(sizeof(FrameData), GraphicsConstants::CONSTANT_BUFFER_ALIGNMENT);

Computes aligned offset for frame-constant buffers.
Ensures safe and efficient memory writes per-frame.

-------------------------------------------------------------------------------------------------------
-------------
Instance Forking (Instancing)
uint32_t m_InstanceDataForks = 0;

Tracks the number of times instance data has been duplicated this frame (e.g., for branching visual
states, special effects).

-------------------------------------------------------------------------------------------------------
-------------
Scene Lighting and Frame Metadata
Frame m_Frame{ .AmbientLight = { 0.1, 0.1, 0.1} };
FrameData m_CameraData{};
std::vector<Light> m_Lights;
InstanceData m_DefaultInstanceData = InstanceData();

m_Frame: Global frame constants, including ambient lighting.
m_CameraData: Per-frame camera matrices and frustum data.
m_Lights: Collection of point/spot/directional lights in the scene.
m_DefaultInstanceData: Used when no override is supplied during rendering.

-------------------------------------------------------------------------------------------------------
-------------
Texture Binding Default
uint32_t m_DefaultTextureIDX = 0;

Fallback or default texture index used when no texture is explicitly bound to a material or instance.
```

```
-------------------------------------------------------------------------------------------------------------
-------------
Render State Dirty Flag
bool m_Pipeline_dirty;

If true, the rendering pipeline or shader bindings need to be recompiled or rebound before the next
frame.

-------------------------------------------------------------------------------------------------------------
-------------
Debug Toggle for Normals

bool m_VisualiseNormals = false;

When enabled, shaders visualize surface normals—useful for debugging lighting or tangent-space
calculations.
```

```
 void InitBuffers(BufferDesc bufferdesc) {

        //ScreenDataBuffer Buffer
        bufferdesc.Init(pennyBufferSize, ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead);
        ScreenDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer(bufferdesc);
});

        //ScanlineDataBuffer Buffer
        ScanlineDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer({
.AccessFlags = ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead, .Size = sizeof(ScanlineBuffer)
}); });

        //VignettingDataBuffer Buffer
        VignettingDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer({
.AccessFlags = ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead, .Size = sizeof(VignettingBuffer)
}); });

        //PosterizeDataBuffer Buffer
        PosterizeDataBuffer.ForEach([&](BufferRef& ref) {ref =
ResourceFactory::CreateBuffer({.AccessFlags = ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead,
.Size = sizeof(PosterizeBuffer)}); });

        BloomDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer({ .AccessFlags =
ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead, .Size = sizeof(BloomBuffer) }); });
        Bloom2DataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer({ .AccessFlags
= ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead, .Size = sizeof(Bloom2Buffer) }); });
```

```
        BlurGauDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer({ .AccessFlags
= ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead, .Size = sizeof(BlurGauBuffer) }); });
        ChromaTrueDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer({
.AccessFlags = ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead, .Size = sizeof(ChromaTrueBuffer)
}); });
        ChromaStylizedDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer({
.AccessFlags = ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead, .Size =
sizeof(ChromaStylizedBuffer) }); });
        ColourGradDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer({
.AccessFlags = ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead, .Size = sizeof(ColourGradBuffer)
}); });
        EmbossDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer({ .AccessFlags
= ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead, .Size = sizeof(EmbossBuffer) }); });
        GlassDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer({ .AccessFlags =
ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead, .Size = sizeof(GlassBuffer) }); });
        pixelDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer({ .AccessFlags =
ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead, .Size = sizeof(PixelBuffer) }); });


    }
```

InitBuffers(BufferDesc bufferdesc) — Buffer Initialization Function

---

Purpose

This function allocates and initializes all CPU-writeable / GPU-readable buffers required by the post-processing system. Each buffer corresponds to a specific effect like scanlines, bloom, chromatic aberration, emboss, etc.
These buffers will later be updated with their respective data structs (like ScanlineBuffer, BloomBuffer, etc.) before being uploaded to the GPU.

Parameters

BufferDesc bufferdesc

A reusable buffer description object passed in from outside.
The Init method is used to set:
Size: e.g. pennyBufferSize (64 KB default)
AccessFlags: CPU write access and GPU read access, suitable for constant buffers.

Function Body Walkthrough

```
// ScreenDataBuffer
bufferdesc.Init(pennyBufferSize, ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead);
ScreenDataBuffer.ForEach([&](BufferRef& ref) {
    ref = ResourceFactory::CreateBuffer(bufferdesc);
});

Buffer: ScreenDataBuffer
Usage: Holds general screen-space rendering info like screen resolution.
Access: CPU-writable for updates each frame, GPU-readable in shaders.

-----------------------------------------------------------------------------------------

Per-Effect Buffers

Each of the following follows this template:
Buffer.ForEach([&](BufferRef& ref) {
    ref = ResourceFactory::CreateBuffer({
        .AccessFlags = ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead,
        .Size = sizeof(CorrespondingBufferStruct)
    });
});

List of Buffers and Their Purpose:
Buffer, Struct, Purpose
ScanlineDataBuffer, ScanlineBuffer, CRT scanline simulation
VignettingDataBuffer, VignettingBuffer, Darkens screen edges to focus viewer's attention
PosterizeDataBuffer, PosterizeBuffer, Color quantization for stylized/retro effects
BloomDataBuffer, BloomBuffer, Radial light bleeding (intense light bloom)
Bloom2DataBuffer, Bloom2Buffer, Simpler threshold-based bloom effect
BlurGauDataBuffer, BlurGauBuffer, Configurable Gaussian blur filter
ChromaTrueDataBuffer, ChromaTrueBuffer, Basic chromatic aberration (RGB channel offset)
ChromaStylizedDataBuffer, ChromaStylizedBuffer, Advanced multi-channel chromatic distortions
ColourGradDataBuffer, ColourGradBuffer, Gradient overlays between two colors
EmbossDataBuffer, EmbossBuffer, Screen-space bump/edge highlighting for stylized visuals
GlassDataBuffer, GlassBuffer, Simulates refraction through glass panels or water
pixelDataBuffer, PixelBuffer, Low-resolution pixelation filter

-----------------------------------------------------------------------------------------

Implementation Notes

MultiResource Pattern: Each buffer is wrapped in a MultiResource object, enabling use in multi-frame
rendering systems (e.g., triple buffering or per-frame resource sets).
Access Flags: All buffers are marked with:
CpuWrite: For CPU-side updates each frame.
GpuRead: Allows shaders to access the data in rendering or post-processing passes.
Size Calculation: For most buffers, sizeof(Struct) ensures only the required space is reserved.
```

```
void InitData(uint32_t InstanceDataBufferSize) {

int offset = 0; //nothing to offset by /-change cmment layter

//ScreenDATA
screen_desc.InitAsStructuredBuffer<ScreenBuffer>(offset, InstanceDataBufferSize / sizeof(ScreenBuffer));

//ScanlineDATA
scanline_desc.InitAsStructuredBuffer<ScanlineBuffer>(offset, InstanceDataBufferSize /
sizeof(ScanlineBuffer));

//vignettingDATA
vignetting_desc.InitAsStructuredBuffer<VignettingBuffer>(offset, InstanceDataBufferSize /
sizeof(VignettingBuffer));

//posterizeDATA
posterize_desc.InitAsStructuredBuffer<PosterizeBuffer>(offset, InstanceDataBufferSize /
sizeof(PosterizeBuffer));

//bloomDATA
bloom_desc.InitAsStructuredBuffer<BloomBuffer>(offset, InstanceDataBufferSize / sizeof(BloomBuffer));

//bloom2DATA
bloom2_desc.InitAsStructuredBuffer<Bloom2Buffer>(offset, InstanceDataBufferSize / sizeof(Bloom2Buffer));

//blurGauDATA
blurGau_desc.InitAsStructuredBuffer<BlurGauBuffer>(offset, InstanceDataBufferSize /
sizeof(BlurGauBuffer));

//ChromaTrueDATA
ChromaTrue_desc.InitAsStructuredBuffer<ChromaTrueBuffer>(offset, InstanceDataBufferSize /
sizeof(ChromaTrueBuffer));

//ChromaStylizedDATA
ChromaStylized_desc.InitAsStructuredBuffer<ChromaStylizedBuffer>(offset, InstanceDataBufferSize /
sizeof(ChromaStylizedBuffer));

//colourGradDATA
colourGrad_desc.InitAsStructuredBuffer<ColourGradBuffer>(offset, InstanceDataBufferSize /
sizeof(ColourGradBuffer));

//embossDATA
Emboss_desc.InitAsStructuredBuffer<EmbossBuffer>(offset, InstanceDataBufferSize / sizeof(EmbossBuffer));

//glassDATA
glass_desc.InitAsStructuredBuffer<GlassBuffer>(offset, InstanceDataBufferSize / sizeof(GlassBuffer));

//pixelDATA
pixel_desc.InitAsStructuredBuffer<PixelBuffer>(offset, InstanceDataBufferSize / sizeof(PixelBuffer));
```

```
}
```

InitData(uint32_t InstanceDataBufferSize) — Descriptor Initialization Function

---

Purpose

This function initializes GPU buffer view descriptors (of type BufferViewDesc) for each post-processing effect buffer. These descriptors define how shaders read structured data (e.g., uniform-like structures) from memory.
Each view maps a specific type (e.g., ScanlineBuffer, BloomBuffer) to a region of a buffer, enabling structured access on the GPU side.

Parameters

uint32_t InstanceDataBufferSize

Total memory size available for structured data (in bytes).
Used to compute the number of struct instances per buffer.
Ensures consistent allocation across different buffers.

Function Body Walkthrough

int offset = 0; // nothing to offset by / - change comment later

Offset into the buffer, currently set to 0 for all views (no subregioning yet).
Placeholder for future support for buffer suballocations (i.e., offset-based binding).

Descriptor Initialization Pattern

Each line follows the same structure:

```
descriptor.InitAsStructuredBuffer<StructType>(
    offset,
    InstanceDataBufferSize / sizeof(StructType)
);
```

StructType: The buffer structure to be read by the GPU (e.g., PosterizeBuffer).
offset: Offset into the buffer (currently always 0).
count: Number of struct elements that can fit in the allocated buffer.

Post-Processing Buffer View Descriptors

Descriptor, Struct Type, Purpose

screen_desc, ScreenBuffer, Holds screen dimensions and padding
scanline_desc, ScanlineBuffer, Defines scanline strength, direction, dimming
vignetting_desc, VignettingBuffer, Holds inner/outer radius and opacity for vignette effect
posterize_desc, PosterizeBuffer, Describes color step sizes for posterization
bloom_desc, BloomBuffer, Describes bloom radii and amplification
bloom2_desc, Bloom2Buffer, Alternate bloom method using threshold/amp
blurGau_desc, BlurGauBuffer, Gaussian blur parameters and sampling weights
ChromaTrue_desc, ChromaTrueBuffer, Simple chromatic aberration offsets
ChromaStylized_desc, ChromaStylizedBuffer, Advanced chromatic offsets and color weighting
colourGrad_desc, ColourGradBuffer, Defines gradient transition between two colors
Emboss_desc, EmbossBuffer, Emboss width, height, and greyscale toggle
glass_desc, GlassBuffer, Glass pane size, used for distortion effects
pixel_desc, PixelBuffer, Pixel size data for pixelation effect

---

Why This Step Matters

These descriptors are what allow the GPU to see the buffer contents correctly.
Without this, data may be interpreted incorrectly or misaligned.
The structured format also gives type safety on the shader side, aligning with constant buffer layout rules.

```
 void BeginIncrementCounter() {
        //m_ScanlineBuffer = ScanlineBuffer(LineThickness_, DimmedFactor_, transferPower_, vertical_);
        ScreenDataBuffer.IncrementCounter();
        ScanlineDataBuffer.IncrementCounter();
        VignettingDataBuffer.IncrementCounter();
        PosterizeDataBuffer.IncrementCounter();


        BloomDataBuffer.IncrementCounter();
        Bloom2DataBuffer.IncrementCounter();
        BlurGauDataBuffer.IncrementCounter();
        ChromaTrueDataBuffer.IncrementCounter();


        ChromaStylizedDataBuffer.IncrementCounter();
        ColourGradDataBuffer.IncrementCounter();
        EmbossDataBuffer.IncrementCounter();
        GlassDataBuffer.IncrementCounter();


        pixelDataBuffer.IncrementCounter();


    }
```

BeginIncrementCounter() – Advance Frame Resource Counters

```
---------------------------------------------------------------------------------------------
Purpose

This function increments the internal resource counters for all post-processing MultiResource buffers,
enabling them to switch to the next buffer instance for the current frame. It’s designed to be called
once per frame (typically at the start of the frame update or rendering pass).

---------------------------------------------------------------------------------------------
Context: MultiResource Buffering

MultiResource<T> is a wrapper that contains a buffer of GPU buffer references.
Internally, IncrementCounter() moves to the next available buffer slot.

Allowing the CPU to prepare data while the GPU is still reading previous frame resources.

---------------------------------------------------------------------------------------------
Affected Buffers

The following buffers have their internal index advanced:
Buffer, Purpose / Contents
ScreenDataBuffer, Contains screen resolution and padding
ScanlineDataBuffer, Controls scanline rendering parameters
VignettingDataBuffer, Stores vignette falloff and opacity data
PosterizeDataBuffer, Contains parameters for posterization effect
BloomDataBuffer, Parameters for radial bloom/glow
Bloom2DataBuffer, Alternate bloom with amp/threshold settings
BlurGauDataBuffer, Gaussian blur weight and sampling configuration
ChromaTrueDataBuffer, Basic chromatic aberration settings
ChromaStylizedDataBuffer, Advanced chromatic offset/weight combinations
ColourGradDataBuffer, Gradient colour data (start/end)
EmbossDataBuffer, Emboss intensity, greyscale toggle, dimensions
GlassDataBuffer, Simulates panel-like refraction or distortion
pixelDataBuffer, Pixelation strength and pixel block size



---------------------------------------------------------------------------------------------
Implementation Notes

Safe Resource Management: Ensures each frame’s data write doesn’t overwrite data in use by the GPU.
Temporal Effects: If any of these buffers support motion-aware or frame-history-based effects, this
system enables it.
Expandability: Adding new buffers to this system simply requires calling IncrementCounter() on the new
MultiResource.
```

```
  void EndCopyToBuffer() {


        int offset = 0;
```

```
            //GraphicsContext::CopyDataToBuffer(PosterizeDataBuffer.Get().get(), offset, sizeof(Frame),
&m_Frame);
            //GraphicsContext::CopyDataToBuffer(PosterizeDataBuffer.Get().get(), offset,
sizeof(PosterizeBuffer), &m_PosterizeBuffer);



            //switch (EFFECT_PERMUTATIONS)
            switch (CURRENT_STATE_)
            //switch (ShaderToggles)
            {
            case PIXELATE_:
                GraphicsContext::CopyDataToBuffer(pixelDataBuffer.Get().get(), offset,
sizeof(PixelBuffer), &m_PixelBuffer);
                break;
            case POSTERIZE_:
                GraphicsContext::CopyDataToBuffer(PosterizeDataBuffer.Get().get(), offset,
sizeof(PosterizeBuffer), &m_PosterizeBuffer);
                break;
            case SCANLINE_:
                GraphicsContext::CopyDataToBuffer(ScanlineDataBuffer.Get().get(), offset,
sizeof(ScanlineBuffer), &m_ScanlineBuffer);
                break;
            case VIGNETTE_:
                GraphicsContext::CopyDataToBuffer(VignettingDataBuffer.Get().get(), offset,
sizeof(VignettingBuffer), &m_VignettingBuffer);
                break;
            case BLOOM_:
                //RSPD.SetPixelShader(L"bloom_ps");
                GraphicsContext::CopyDataToBuffer(Bloom2DataBuffer.Get().get(), offset,
sizeof(Bloom2Buffer), &m_Bloom2Buffer);
                break;
            case CHROMABER_TRUE_:
                GraphicsContext::CopyDataToBuffer(ChromaTrueDataBuffer.Get().get(), offset,
sizeof(ChromaTrueBuffer), &m_ChromaTrueBuffer);
                break;
            case CHROMABER_STYLE_:
                GraphicsContext::CopyDataToBuffer(ChromaStylizedDataBuffer.Get().get(), offset,
sizeof(ChromaStylizedBuffer), &m_ChromaStylizedBuffer);
                break;
            case COLOURGRAD_:
                GraphicsContext::CopyDataToBuffer(ColourGradDataBuffer.Get().get(), offset,
sizeof(ColourGradBuffer), &m_ColourGradBuffer);
                break;
            case GLASS_:
                GraphicsContext::CopyDataToBuffer(GlassDataBuffer.Get().get(), offset,
sizeof(GlassBuffer), &m_GlassBuffer);
                break;
```

```
            case EMBOS_:
                GraphicsContext::CopyDataToBuffer(EmbossDataBuffer.Get().get(), offset,
sizeof(EmbossBuffer), &m_EmbossBuffer);
                break;
            case BLUR_GAU_:
                GraphicsContext::CopyDataToBuffer(BlurGauDataBuffer.Get().get(), offset,
sizeof(BlurGauBuffer), &m_BlurGauBuffer);
                break;


            default:
                GraphicsContext::CopyDataToBuffer(ScreenDataBuffer.Get().get(), offset,
sizeof(ScreenBuffer), &m_ScreenBuffer);


                break;
            }


    }
```

EndCopyToBuffer() — Upload Active Post-Processing Buffer to GPU

Purpose

This function finalizes the frame setup by copying CPU-side post-processing effect data (e.g., scanline strength, vignette radius, etc.) into the currently active GPU buffer associated with the enabled effect. It is usually called at the end of frame preparation, just before rendering.

How It Works

Offset is set to zero: All data is copied to the beginning of each buffer.
`int offset = 0;`

The function then selects the active post-processing effect via the CURRENT_STATE_ switch:
`switch (CURRENT_STATE_)`

This is based on a shader state enum (ShaderToggles) such as PIXELATE_, BLOOM_, etc.
For the active effect, the corresponding CPU-side buffer (e.g., m_PixelBuffer, m_VignettingBuffer) is copied to its associated MultiResource GPU buffer via:
`GraphicsContext::CopyDataToBuffer(buffer, offset, size, &source);`

If none of the predefined post-processing states match, it defaults to uploading the ScreenBuffer.

supported Shader States and Corresponding Buffers
Shader Toggle, GPU Buffer, CPU Struct
PIXELATE_, pixelDataBuffer, m_PixelBuffer
POSTERIZE_, PosterizeDataBuffer, m_PosterizeBuffer

```
SCANLINE_, ScanlineDataBuffer, m_ScanlineBuffer
VIGNETTE_, VignettingDataBuffer, m_VignettingBuffer
BLOOM_, Bloom2DataBuffer, m_Bloom2Buffer
CHROMABER_TRUE_, ChromaTrueDataBuffer, m_ChromaTrueBuffer
CHROMABER_STYLE_, ChromaStylizedDataBuffer, m_ChromaStylizedBuffer
COLOURGRAD_, ColourGradDataBuffer, m_ColourGradBuffer
GLASS_, GlassDataBuffer, m_GlassBuffer
EMBOS_, EmbossDataBuffer, m_EmbossBuffer
BLUR_GAU_, BlurGauDataBuffer, m_BlurGauBuffer
(default), ScreenDataBuffer, m_ScreenBuffer
```

```
  void DrawVBIB_SetInlineResourceViewGraphics() {
        //m_ScanlineBuffer = ScanlineBuffer(LineThickness_, DimmedFactor_, transferPower_, vertical_);


            //RenderCommand::SetInlineResourceViewGraphics(4, PosterizeDataBuffer.Get().get(),
posterize_desc, ViewAccessType_ConstantBuffer);


            ////switch (EFFECT_PERMUTATIONS)
            switch (CURRENT_STATE_)
            {
            case PIXELATE_:
                RenderCommand::SetInlineResourceViewGraphics(4, pixelDataBuffer.Get().get(), pixel_desc,
ViewAccessType_ConstantBuffer);
                break;
            case POSTERIZE_:
                RenderCommand::SetInlineResourceViewGraphics(4, PosterizeDataBuffer.Get().get(),
posterize_desc, ViewAccessType_ConstantBuffer);
                break;
            case SCANLINE_:
                RenderCommand::SetInlineResourceViewGraphics(4, ScanlineDataBuffer.Get().get(),
scanline_desc, ViewAccessType_ConstantBuffer);
                break;
            case VIGNETTE_:
                RenderCommand::SetInlineResourceViewGraphics(4, VignettingDataBuffer.Get().get(),
vignetting_desc, ViewAccessType_ConstantBuffer);
                break;
            case BLOOM_:
                //RenderCommand::SetInlineResourceViewGraphics(4, BloomDataBuffer.Get().get(),
bloom_desc, ViewAccessType_ConstantBuffer);
                RenderCommand::SetInlineResourceViewGraphics(4, Bloom2DataBuffer.Get().get(),
bloom2_desc, ViewAccessType_ConstantBuffer);
                break;
            case CHROMABER_TRUE_:
                RenderCommand::SetInlineResourceViewGraphics(4, ChromaTrueDataBuffer.Get().get(),
ChromaTrue_desc, ViewAccessType_ConstantBuffer);
                break;
            case CHROMABER_STYLE_:
```

```
                RenderCommand::SetInlineResourceViewGraphics(4, ChromaStylizedDataBuffer.Get().get(),
ChromaStylized_desc, ViewAccessType_ConstantBuffer);
                break;
            case COLOURGRAD_:
                RenderCommand::SetInlineResourceViewGraphics(4, ColourGradDataBuffer.Get().get(),
colourGrad_desc, ViewAccessType_ConstantBuffer);
                break;
            case GLASS_:
                RenderCommand::SetInlineResourceViewGraphics(4, GlassDataBuffer.Get().get(), glass_desc,
ViewAccessType_ConstantBuffer);
                break;
            case EMBOS_:
                RenderCommand::SetInlineResourceViewGraphics(4, EmbossDataBuffer.Get().get(),
Emboss_desc, ViewAccessType_ConstantBuffer);
                break;
            case BLUR_GAU_:
                RenderCommand::SetInlineResourceViewGraphics(4, BlurGauDataBuffer.Get().get(),
blurGau_desc, ViewAccessType_ConstantBuffer);
                break;


            default:
                RenderCommand::SetInlineResourceViewGraphics(4, ScreenDataBuffer.Get().get(),
screen_desc, ViewAccessType_ConstantBuffer);


                break;
            }


    }
```

DrawVBIB_SetInlineResourceViewGraphics() — Bind Per-Effect GPU Buffer as Inline Constant Buffer

---

Purpose

This function selects and binds the appropriate GPU-side constant buffer for the currently enabled post-processing effect, using a shader slot (register) during the graphics pipeline draw call.
It acts as a resource view dispatcher, driven by the current shader state (CURRENT_STATE_).

Binding Mechanism

Each effect has a buffer and a buffer view (BufferViewDesc) prepared during initialization. This function uses:

```
RenderCommand::SetInlineResourceViewGraphics(
    slotIndex,           // e.g. 4
    bufferRef,           // GPU buffer (e.g. ScanlineDataBuffer)
```

```
    bufferViewDesc,      // Predefined view describing buffer layout
    ViewAccessType       // ViewAccessType_ConstantBuffer
);

This binds a GPU buffer as a constant buffer at graphics shader register slot 4.

---------------------------------------------------------------------------------------------------------
Supported Effects and Bindings

Shader Toggle, GPU Buffer, View Descriptor
PIXELATE_, pixelDataBuffer, pixel_desc
POSTERIZE_, PosterizeDataBuffer, posterize_desc
SCANLINE_, ScanlineDataBuffer, scanline_desc
VIGNETTE_, VignettingDataBuffer, vignetting_desc
BLOOM_, Bloom2DataBuffer, bloom2_desc
CHROMABER_TRUE_, ChromaTrueDataBuffer, ChromaTrue_desc
CHROMABER_STYLE_, ChromaStylizedDataBuffer, ChromaStylized_desc
COLOURGRAD_, ColourGradDataBuffer, colourGrad_desc
GLASS_, GlassDataBuffer, glass_desc
EMBOS_, EmbossDataBuffer, Emboss_desc
BLUR_GAU_, BlurGauDataBuffer, blurGau_desc
(default fallback), ScreenDataBuffer, screen_desc
Each case ensures that the appropriate constant buffer view is bound, matching the currently active
shader.


---------------------------------------------------------------------------------------------------------
Notes

Uses slot 4 uniformly for all effects – make sure shaders expect the constant buffer at this slot.
Falls back to ScreenDataBuffer if no specific effect is active.
Commented out Bloom1 this was accidental – the shader works just ran out of time to put it in
```

Penny Values

```
Skateboard::ShaderInputLayoutRef RootSig;

//pipeline permutations
std::array<Skateboard::PipelineRef, EFFECT_PERMUTATIONS> Pipeline

//normal rendering
Skateboard::PipelineRef NormalVisualizer;
```

```
Skateboard::MultiResource<Skateboard::BufferRef> TriangleDataBu
Skateboard::MultiResource<Skateboard::BufferRef> InstanceDataBu
Skateboard::MultiResource<Skateboard::BufferRef> LightDataBuffe
```

```
// post-processing buffers

size_t pennyBufferSize = 64 * 1024; //

//Buffer refs
Skateboard::MultiResource<Skateboard::BufferRef> ScreenDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> ScanlineDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> VignettingDataBuffe
Skateboard::MultiResource<Skateboard::BufferRef> PosterizeDataBuffer

Skateboard::MultiResource<Skateboard::BufferRef> BloomDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> Bloom2DataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> BlurGauDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> ChromaTrueDataBuffe

Skateboard::MultiResource<Skateboard::BufferRef> ChromaStylizedDataB
Skateboard::MultiResource<Skateboard::BufferRef> ColourGradDataBuffe
Skateboard::MultiResource<Skateboard::BufferRef> EmbossDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> GlassDataBuffer;

Skateboard::MultiResource<Skateboard::BufferRef> pixelDataBuffer;
```

```
// desc
BufferViewDesc screen_desc;
BufferViewDesc scanline_desc;
BufferViewDesc vignetting_desc;
BufferViewDesc posterize_desc;

BufferViewDesc bloom_desc;
BufferViewDesc bloom2_desc;
BufferViewDesc blurGau_desc;
BufferViewDesc ChromaTrue_desc;

BufferViewDesc ChromaStylized_desc;
BufferViewDesc colourGrad_desc;
BufferViewDesc Emboss_desc;
BufferViewDesc glass_desc;

BufferViewDesc pixel_desc;
```

```
// init buffer structs
ScreenBuffer m_ScreenBuffer = ScreenBuffer();
ScanlineBuffer m_ScanlineBuffer = ScanlineBuffer();
VignettingBuffer m_VignettingBuffer = VignettingBuffer();
PosterizeBuffer m_PosterizeBuffer = PosterizeBuffer();

BloomBuffer     m_BloomBuffer = BloomBuffer();
Bloom2Buffer    m_Bloom2Buffer = Bloom2Buffer();
BlurGauBuffer m_BlurGauBuffer = BlurGauBuffer();
ChromaTrueBuffer m_ChromaTrueBuffer = ChromaTrueBuffer();

ChromaStylizedBuffer m_ChromaStylizedBuffer = ChromaStylizedBuf
ColourGradBuffer m_ColourGradBuffer = ColourGradBuffer();
EmbossBuffer m_EmbossBuffer = EmbossBuffer();
GlassBuffer m_GlassBuffer = GlassBuffer();

PixelBuffer m_PixelBuffer = PixelBuffer();
```

```
// Pipeline default sate inti
ShaderToggles CURRENT_STATE_ = DEFAULT_STATE_
```

```
//--------------
//inline view desc
BufferViewDesc sbvdesc;
BufferViewDesc lightsbvdesc;
BufferViewDesc cbvdesc;
```

```
//resterizer
SamplerDesc StaticSamplerDesc = SamplerDesc::InitAsDefaultTextureSamp
```

```
//counts offsets for dynamic data uploaded every frame
size_t m_Offset = ROUND_UP(sizeof(FrameData), GraphicsConstants::CONSTANT_BUFFER_ALIG
```

```
//forked on each call with a unique data pointer/ otherwise default instance data
uint32_t m_InstanceDataForks = 0;
```

```
Frame m_Frame{ .AmbientLight = { 0.1, 0.1, 0.1}
FrameData m_CameraData{};
```

```
std::vector<Light> m_Lights;
```

```
InstanceData m_DefaultInstanceData = InstanceDat
```

```
uint32_t m_DefaultTextureIDX = 0;
```

```
bool m_Pipeline_dirty;
bool m_VisualiseNormals = false;// probs del
```

```
//effects - list stuff
std::vector < SavedEffect> savedEffect_List; // max4 for n
```

```
//Output Texture
MultiResource<TextureBufferRef> OutputTexture;
MultiResource<RenderTargetViewRef> OutputRTV;
MultiResource<TextureSRVRef>  OutputSRV;

MultiResource<TextureBufferRef> OutputTextures_0
MultiResource<RenderTargetViewRef> OutputRTVs_0;
MultiResource<TextureSRVRef>  OutputSRVs_0;

MultiResource<TextureBufferRef> OutputTextures_[
MultiResource<RenderTargetViewRef> OutputRTVs_[2
MultiResource<TextureSRVRef>  OutputSRVs_[2];
```

```
//test values

InstanceData current_instancedata;

RasterizationPipelineDesc Raster{};
```

```
int previewscreenSize = 2;
```

```
float aberrationFactor_XY = 0.5;
bool link_XY = true;
```

```
float colC[4] = {0.8,0.5,0.9,1};
```

```
float chromaticWeights1[4] = {0.8,0.5,0,1}
float chromaticWeights2[4] = {0.8,0.5,0,1}
```

```
float startColor_CG[3] = { 1,1,1};
float endColor_CG[3] = { 1.2, 0.01, 0.51 }
```

```
float4 chromaticOffset1;
float4 chromaticOffset2;
float4 chromaticOffset_XY;
bool offsetlinked = true;
bool XYlinked = true;
```

```
bool panelinked = true;
bool pixelinked = true;
```

```
float weight_all;
float multi_all;
```

```
bool link_weight;
bool link_Multiplier;
```

```
float4 SpecularC = float4(1, 1, 1, 1);
float4 DiffuseC = float4(1, 1, 1, 1);
float SpecularWeight = 1;
float SpecularPower = 200;
```

```
std::vector<pos::Vertex> vertices_screen =
{
    { float3( 0.5, 0.5,0) ,float3(1,0,0)        ,float2(0,1)   float3( 0.05,0.05)     , float
    { float3(0.5,-0.5,0)  ,float3(0,1,0)        ,float2(1,1)   float3( 0.05,0.05)     , float
    { float3(0,0.5,0)     ,float3(0,0,1)        ,float2(0.5, 0) - float2(-0.05,0.05), float3(0,
};
```

The Renderer

```
//renderer + pipeline stuff

void PennyRender::Init() {

    SKTBD_APP_INFO("Renderer test 2 Init")

   //Layout
     ShaderInputLayoutDesc Layout{};


     //instance index //0- slot
     Layout.AddRootConstant(1, 0);


     //frame data // 1- slot
     Layout.AddConstantBufferView(1);


     //instance data // 2- slot
     Layout.AddShaderResourceView(0);


     //light data // 3- slot
     Layout.AddShaderResourceView(1);


     //--


     //all post-processing buffers data // 4 - slot
     Layout.AddConstantBufferView(2);


     Layout.AddStaticSampler(StaticSamplerDesc, 0);


     Layout.DescriptorsDirctlyAddresssed = true;
     Layout.SamplersDirectlyAddressed = true;
     Layout.CanUseInputAssembler = true;


     Init_RasterPipeline(Layout);//


     renderToScreen_init();//// initilising render to target
     renderToScreen_init_pass(OutputTextures_0, OutputRTVs_0, OutputSRVs_0);// - not in use -- suppost to
be for multi- render passes


}
```

PennyRender::Init() — Pipeline and Renderer Initialization

Purpose

This function initializes the shader input layout, post-processing pipeline, and render targets for the rendering system used by the PennyRender renderer.
It sets up all required resource bindings, including constant buffers, samplers, and resource views, and then calls initialization routines for the graphics raster pipeline and render-to-screen pipeline.

Function Breakdown

Logging
SKTBD_APP_INFO("Renderer test 2 Init")

Logs the initialization event for debugging or performance tracking.

Shader Input Layout Setup

ShaderInputLayoutDesc Layout{};

A layout descriptor is created to define how shaders will receive data (CBVs, SRVs, root constants, etc.).

Resource Slot Bindings:
Shader Slot, Type, Resource Use
0, RootConstant, Instance index
1, ConstantBufferView, Per-frame data (e.g., camera)
2, ShaderResourceView, Instance data buffer
3, ShaderResourceView, Light data buffer
4, ConstantBufferView, Post-processing data buffer
0 (Sampler), StaticSampler, Texture sampling

```
Layout.AddRootConstant(1, 0);
Layout.AddConstantBufferView(1);  // Frame data (e.g., camera)
Layout.AddShaderResourceView(0);  // Instance data (e.g., transform matrices)
Layout.AddShaderResourceView(1);  // Light data
Layout.AddConstantBufferView(2);  // Post-processing buffer (e.g., bloom, vignette)
Layout.AddStaticSampler(StaticSamplerDesc, 0);
```

Shader Binding Model Configuration

```
Layout.DescriptorsDirctlyAddresssed = true;
Layout.SamplersDirectlyAddressed = true;
Layout.CanUseInputAssembler = true;
```

These flags determine how shaders access their resources:
Direct addressing of descriptors (not through descriptor tables).
Input assembler is enabled, allowing use of vertex/index buffers.

Pipeline Initialization

Init_RasterPipeline(Layout);

Initializes the rasterization pipeline with the defined shader layout. This configures the graphics pipeline (shaders, buffers, states) to prepare for rendering.

Render Target Initialization

renderToScreen_init(); // Initializes base render-to-texture resources

renderToScreen_init_pass(OutputTextures_0, OutputRTVs_0, OutputSRVs_0);

Sets up render targets:
OutputTextures_0: The textures to render into.
OutputRTVs_0: Render Target Views.
OutputSRVs_0: Shader Resource Views for sampling the output in post-processing.
Note: The second call is currently marked as "not in use" — originally intended for multi-pass post-processing setups.

Conceptual Summary

This function configures the graphics pipeline to accept structured and consistent input from CPU-side buffers and sets up output targets for screen rendering. It's a crucial part of the initialization flow that ensures shaders receive the right data in the right slots and that the rendering output can be displayed or post-processed.

Key Features Enabled

Dynamic instance rendering (via root constants + instance SRVs).
Per-frame camera and lighting data.
Slot for post-processing constant buffers (e.g., bloom, vignette).
Support for static samplers.
Input assembler support (for vertex/indexed geometry).
Configurable render targets.

```
void PennyRender::Init_RasterPipeline(ShaderInputLayoutDesc Layout) {
```

```
RootSig = ResourceFactory::CreateShaderInputLayout(Layout);


//RasterizationPipelineDesc Raster{};
//Raster{};
Raster.Rasterizer = RasterizerConfig::Default();
Raster.Blend.AlphaToCoverage = false;
Raster.Blend.IndependentBlendEnable = false;
Raster.Blend.RTBlendConfigs[0].BlendEnable = false;
Raster.Blend.RTBlendConfigs[0].RenderTargetWriteMask = 0xF;
Raster.DepthStencil.DepthEnable = false;
Raster.DepthStencil.BackFace.StencilDepthFailOp = SKTBD_StencilOp_KEEP;
Raster.DepthStencil.BackFace.StencilFailOp = SKTBD_StencilOp_KEEP;
Raster.DepthStencil.BackFace.StencilFunc = SKTBD_CompareOp_ALWAYS;
Raster.DepthStencil.BackFace.StencilPassOp = SKTBD_StencilOp_KEEP;
Raster.DepthStencil.DepthFunc = SKTBD_CompareOp_LESS;
Raster.DepthStencil.DepthWriteAll = true;
Raster.DepthStencil.FrontFace = Raster.DepthStencil.BackFace;
Raster.DepthStencil.StencilEnable = false;
Raster.DepthstencilTargetFormat = DataFormat_DEFAULT_DEPTHSTENCIL;
Raster.InputPrimitiveType = PrimitiveTopologyType_Triangle;
Raster.RenderTargetCount = 1;
Raster.RenderTargetDataFormats[0] = DataFormat_DEFAULT_BACKBUFFER;
Raster.InputVertexLayout = Vertex::VertexLayout();
Raster.SampleCount = 1;
Raster.SampleQuality = 0;
Raster.SampleMask = 1;


//initilising all shaders
Raster.SetVertexShader(L"CMP203_VertexShader");
Raster.SetPixelShader(L"CMP203_PixelShader_Unlit");
//Raster.SetPixelShader(L"test_ps");


//piepline permutations
Init_ShaderPermutation(Raster);


//NormalPipeline


Raster.DepthStencil.DepthEnable = true;
Raster.SetGeometryShader(L"CMP203_NormalGS_GS");
Raster.SetVertexShader(L"CMP203_NormalGS_VS");
Raster.SetPixelShader(L"CMP203_NormalGS_PS");



//creating PipelineDesc -
PipelineDesc PiplDesc;
```

```
    PiplDesc.GlobalLayoutSignature = RootSig;
    PiplDesc.Type = PipelineType_Graphics;
    PiplDesc.TypeDesc = &Raster;


    NormalVisualizer = ResourceFactory::CreatePipelineState(PiplDesc);


    //create Buffers
    //buffer desc
    BufferDesc bufferdesc{};


    //Triangle Buffer
    bufferdesc.Init(DynamicBufferSize, ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead);
    TriangleDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer(bufferdesc); });


    //InstanceData Buffer
    bufferdesc.Init(InstanceDataBufferSize, ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead);
    InstanceDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer(bufferdesc); });


    //LightDataBuffer
    LightDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer(bufferdesc); });


    //-- postprocessing buffers
    InitBuffers(bufferdesc);


    //Create Views
    //Camera data
    cbvdesc.InitAsConstantBuffer<Frame>(0);


    //InstanceData
    sbvdesc.InitAsStructuredBuffer<InstanceData>(0, InstanceDataBufferSize / sizeof(InstanceData));


    //lightdata
    lightsbvdesc.InitAsStructuredBuffer<Light>(0, InstanceDataBufferSize / sizeof(Light));


    //-- postprocessing data desc
    InitData(InstanceDataBufferSize);


    //compute default camera
    auto aspect = GraphicsContext::GetClientAspectRatio();
    m_CameraData.ProjectionMatrix = glm::perspectiveLH(glm::radians(90.f), aspect, 0.01f, 100.f);
    m_CameraData.ViewMatrix = glm::lookAtLH(glm::vec3(0, 0, -10), glm::vec3(0, 0, 0), glm::vec3(0, 1,
0));


    m_DefaultInstanceData.TextureIndex = m_DefaultTextureIDX;
```

```
    //default Instance data being past to buffer
    GraphicsContext::CopyDataToBuffer(InstanceDataBuffer[0].get(), 0, sizeof(InstanceData),
&m_DefaultInstanceData);
    GraphicsContext::CopyDataToBuffer(InstanceDataBuffer[1].get(), 0, sizeof(InstanceData),
&m_DefaultInstanceData);
    GraphicsContext::CopyDataToBuffer(InstanceDataBuffer[2].get(), 0, sizeof(InstanceData),
&m_DefaultInstanceData);

}
```

```
PennyRender::Init_RasterPipeline(ShaderInputLayoutDesc Layout)
--------------------------------------------------------------------------------------------------------
--------
Purpose

This function configures and initializes the graphics pipeline, including the shader input layout,
rasterizer settings, render target formats, and key GPU buffers required for rendering and post-
processing effects. It also sets up a dedicated normal visualization pipeline and initializes associated
structured and constant buffers.
--------------------------------------------------------------------------------------------------------
--------
Function Breakdown
--------------------------------------------------------------------------------------------------------
--------

1. Root Signature Creation
-----------------------------------------------------------------
RootSig = ResourceFactory::CreateShaderInputLayout(Layout);

Creates the shader root signature from the layout passed by PennyRender::Init() – defining how data is
bound to the shaders.

-----------------------------------------------------------------
2. Rasterization Pipeline Configuration
-----------------------------------------------------------------

Raster.Rasterizer = RasterizerConfig::Default();
...
Raster.InputVertexLayout = Vertex::VertexLayout();

Sets up the rasterization pipeline with the following configurations:
No blending, no depth stencil, default backbuffer format.
Triangle-based topology.
Uses a default vertex layout for geometry rendering.
Render output is single-sample (no MSAA).
Shaders assigned:
```

```
Vertex Shader: CMP203_VertexShader
Pixel Shader: CMP203_PixelShader_Unlit
Raster.SetVertexShader(...);
Raster.SetPixelShader(...);


---------------------------------------------------------------------
3. Pipeline Shader Permutations
---------------------------------------------------------------------

Init_ShaderPermutation(Raster);

Handles setup for various shader configurations (e.g., toggling between effects or shader versions).

---------------------------------------------------------------------
4. Normal Visualization Pipeline
---------------------------------------------------------------------

Raster.DepthStencil.DepthEnable = true;
Raster.SetGeometryShader(L"CMP203_NormalGS_GS");
Raster.SetVertexShader(L"CMP203_NormalGS_VS");
Raster.SetPixelShader(L"CMP203_NormalGS_PS");

This alternate configuration enables depth testing and sets geometry shaders for visualizing vertex
normals in 3D.

---------------------------------------------------------------------
5. Pipeline State Creation
---------------------------------------------------------------------

PipelineDesc PiplDesc;
...
NormalVisualizer = ResourceFactory::CreatePipelineState(PiplDesc);

Creates the finalized pipeline state object (PSO) with the configured raster pipeline and root signature.
This is stored in NormalVisualizer.

---------------------------------------------------------------------
6. Buffer Initialization
---------------------------------------------------------------------

Initializes GPU-side structured buffers for rendering geometry and lights:
TriangleDataBuffer
InstanceDataBuffer
LightDataBuffer
bufferdesc.Init(...);
TriangleDataBuffer.ForEach(...);

All buffers are marked as CpuWrite | GpuRead for frequent updates from the CPU side.
```

```
---------------------------------------------------------------------
7. Post-Processing Buffers
---------------------------------------------------------------------

InitBuffers(bufferdesc);

Initializes all post-processing effect buffers (e.g., vignette, posterize, bloom), using the same access
pattern as above.

---------------------------------------------------------------------
8. Buffer View Descriptors
---------------------------------------------------------------------

Sets up views for constant/structured buffers, describing how GPU shaders access each buffer:
cbvdesc.InitAsConstantBuffer<Frame>(0);
sbvdesc.InitAsStructuredBuffer<InstanceData>(...);
lightsbvdesc.InitAsStructuredBuffer<Light>(...);
InitData(...); // Additional post-process buffer view init

Each descriptor (CBV/SBV) defines:
The buffer type.
The offset and size (based on buffer capacity).
How it's accessed by the GPU.

---------------------------------------------------------------------
9. Default Camera Setup
---------------------------------------------------------------------

m_CameraData.ProjectionMatrix = glm::perspectiveLH(...);
m_CameraData.ViewMatrix = glm::lookAtLH(...);

Creates a default view-projection matrix:
Perspective projection with 90° FOV.
Camera located at (0, 0, -10) looking at origin.

---------------------------------------------------------------------
10. Default Instance Data
---------------------------------------------------------------------

m_DefaultInstanceData.TextureIndex = m_DefaultTextureIDX;

Initializes a fallback InstanceData structure, assigning the default texture index for all instances.
GraphicsContext::CopyDataToBuffer(...);

Copies default instance data into three GPU buffers — triple-buffered for use across frames.

---------------------------------------------------------------------
Conceptual Summary
```

Init_RasterPipeline fully configures the rendering pipeline:
Establishes GPU-side shader data bindings.
Builds pipelines for both base rendering and normal visualization.
Allocates and binds buffers for all geometry, lighting, and post-processing data.
Sets up initial camera and rendering state.
It is one of the core setup functions responsible for getting the renderer ready to draw frames efficiently and flexibly.

```
void PennyRender::Init_ShaderPermutation(RasterizationPipelineDesc Raster) {


    //piepline permutations

    for (uint16_t p = 0; p < EFFECT_PERMUTATIONS; p++)
    {
        auto RSPD = Raster;
        RSPD.DepthStencil.DepthEnable = true;

        ////pipeling permutaions

        //switch (CURRENT_STATE_Pass[p])
        switch ((ShaderToggles)p)
        {
        case PIXELATE_:
            RSPD.SetPixelShader(L"pixelate_ps");
            break;
        case POSTERIZE_:
            RSPD.SetPixelShader(L"posterize_ps");
            break;
        case SCANLINE_:
            RSPD.SetPixelShader(L"scanline_ps");
            break;
        case VIGNETTE_:
            RSPD.SetPixelShader(L"vignetting_ps");
            break;
        case BLOOM_:
            //RSPD.SetPixelShader(L"bloom_ps");
            RSPD.SetPixelShader(L"bloom2_ps");// needs to be passed through the blur shader
            break;
        case CHROMABER_TRUE_:
            RSPD.SetPixelShader(L"ChromaAber_True_ps");
            break;
        case CHROMABER_STYLE_:
            RSPD.SetPixelShader(L"ChromaAber_Stylized_ps");
            break;
        case COLOURGRAD_:
            RSPD.SetPixelShader(L"colourGrading_ps");
            break;
        case GLASS_:
            RSPD.SetPixelShader(L"glass_ps");
            break;
        case EMBOS_:
            RSPD.SetPixelShader(L"embossed_ps");
            break;
        case BLUR_GAU_:
            RSPD.SetPixelShader(L"blur_gaussian_ps");
            break;
        case POSTPROSSESSING_:
            RSPD.SetPixelShader(L"Defualt_ps");
            break;
        default:
            RSPD.SetPixelShader(L"Defualt_ps");
            //RSPD.SetPixelShader(L"CMP203_NormalGS_PS");
            break;
        }
```

```
        PipelineDesc PiplDesc;
        PiplDesc.GlobalLayoutSignature = RootSig;
        PiplDesc.Type = PipelineType_Graphics;
        PiplDesc.TypeDesc = &RSPD;
        Pipelines[p] = ResourceFactory::CreatePipelineState(PiplDesc);
        //Pipelines[0] = ResourceFactory::CreatePipelineState(PiplDesc);

    }

}
```

PennyRender::Init_ShaderPermutation(RasterizationPipelineDesc Raster)

Purpose

Initializes and stores a set of graphics pipeline permutations, each using a different pixel shader corresponding to a specific post-processing effect. This system allows fast switching between visual effects at runtime by pre-building the necessary pipeline state objects (PSOs).

🦴 Function Overview

Loop Through Effect Permutations

for (uint16_t p = 0; p < EFFECT_PERMUTATIONS; p++)

Iterates over each defined effect in EFFECT_PERMUTATIONS — a constant that represents the number of supported post-processing effects or visual shader modes.

Base Raster Configuration

auto RSPD = Raster;
RSPD.DepthStencil.DepthEnable = true;

Creates a copy of the input raster pipeline configuration. Then, enables depth testing for each effect permutation (useful for z-ordering or 3D-aware effects).

Set Pixel Shader Based on Effect

A switch is used to assign the correct pixel shader for each permutation index p, interpreted as a ShaderToggles enum.
Example mappings:
Shader Toggle Enum, Assigned Pixel Shader

PIXELATE_, pixelate_ps

POSTERIZE_, posterize_ps

SCANLINE_, scanline_ps

BLOOM_, bloom2_ps

CHROMABER_TRUE_, ChromaAber_True_ps

CHROMABER_STYLE_, ChromaAber_Stylized_ps

COLOURGRAD_, colourGrading_ps

GLASS_, glass_ps

EMBOS_, embossed_ps

BLUR_GAU_, blur_gaussian_ps

POSTPROSSESSING_/other, Defualt_ps

This allows each visual effect to have its dedicated rendering pipeline with appropriate shading logic.

Create Pipeline State for Each Shader

```
PipelineDesc PiplDesc;
PiplDesc.GlobalLayoutSignature = RootSig;
PiplDesc.Type = PipelineType_Graphics;
PiplDesc.TypeDesc = &RSPD;
Pipelines[p] = ResourceFactory::CreatePipelineState(PiplDesc);
```

For each shader permutation:

Constructs a PipelineDesc structure referencing the layout (RootSig) and configured raster data (RSPD).

Calls the GPU abstraction layer to create a pipeline state object (PSO).

Stores the resulting PSO in the Pipelines array indexed by effect type.

Output

The result is a populated Pipelines[] array with prebuilt PSOs, each optimized for a specific post-processing effect. This avoids runtime compilation and enables rapid switching between visual modes during rendering.

Conceptual Summary

This function is a core part of the effect system, handling:

Initialization of shader permutations.

Binding of pixel shaders specific to visual styles.

Creation of PSO objects for all valid rendering paths.

This enables PennyRender to dynamically switch effects via CURRENT_STATE_ without incurring pipeline creation cost during frame rendering.

```
void PennyRender::Begin()
{
    //Move Onto Next frame In Buffer
    TriangleDataBuffer.IncrementCounter();


    InstanceDataBuffer.IncrementCounter();


    LightDataBuffer.IncrementCounter();


    BeginIncrementCounter();//


    //reset the buffer Offset
    m_Offset = ROUND_UP(sizeof(Frame), GraphicsConstants::CONSTANT_BUFFER_ALIGNMENT);



    m_InstanceDataForks = 1;


    RenderCommand::SetInputLayoutGraphics(RootSig.get());
    //RenderCommand::SetPipelineState(Pipelines[0].get());
    //RenderCommand::SetPipelineState(Pipelines[m_PipelineSelection].get());
    RenderCommand::SetPipelineState(Pipelines[CURRENT_STATE_].get());
}
```

```
PennyRender::Begin()
-----------------------------------------------------------------------------------------------
Purpose

The Begin() function prepares the renderer for the next frame by:
Incrementing counters to move to the next frame.
Setting up the necessary buffers and resources.
Configuring the graphics pipeline state for the current frame.


-----------------------------------------------------------------------------------------------
Function Overview
-----------------------------------------------------------------------------------------------
Incrementing Data Buffers

TriangleDataBuffer.IncrementCounter();
InstanceDataBuffer.IncrementCounter();
LightDataBuffer.IncrementCounter();
BeginIncrementCounter();
```

TriangleDataBuffer, InstanceDataBuffer, and LightDataBuffer are incremented to move to the next frame in their respective buffers. This ensures the renderer works with the latest data during rendering.
BeginIncrementCounter() is called to handle additional internal counters (for other buffers related to rendering).

Resetting the Buffer Offset

```
m_Offset = ROUND_UP(sizeof(Frame), GraphicsConstants::CONSTANT_BUFFER_ALIGNMENT);
```

The offset is reset to ensure that the frame data is correctly aligned to the GPU's constant buffer alignment requirements. This ensures that when new data is copied into the buffer, it aligns with the expected memory boundaries.

Setting the Instance Data Forks

```
m_InstanceDataForks = 1;
```

m_InstanceDataForks is set to 1. This manages how many copies (or "forks") of the instance data will be used, typically one for each object in a scene.

Setting the Graphics Pipeline Input Layout

```
RenderCommand::SetInputLayoutGraphics(RootSig.get());
```

Input layout is configured using the Root Signature (RootSig), which is a descriptor of the pipeline’s input resources (constant buffers, resources, etc.).
This function ensures that the GPU knows the layout of incoming vertex data.

Setting the Graphics Pipeline State

```
RenderCommand::SetPipelineState(Pipelines[CURRENT_STATE_].get());
```

The pipeline state for the current frame is set using the Pipelines[CURRENT_STATE_].
The CURRENT_STATE_ is an enum variable that selects the current rendering effect, enabling the appropriate shader/pipeline configuration.

Output
The function does not return anything but prepares the renderer for the upcoming frame by:
Ensuring buffers are properly incremented and prepared.
Setting the input layout and pipeline state to match the current rendering configuration.

Conceptual Summary

The Begin() function is responsible for preparing the renderer for the next frame, ensuring that:
Data buffers are updated for the next frame.
The GPU has the necessary information on how to interpret vertex and constant data.
The correct pipeline state is set to handle the current rendering state, as defined by CURRENT_STATE_.

```
void PennyRender::End()
{
    m_Frame.Matrices = m_CameraData;
    m_Frame.LightCount = m_Lights.size();
    m_Frame.CameraMatrix = glm::inverse(m_CameraData.ViewMatrix);

    //update the frame data buffer section
    GraphicsContext::CopyDataToBuffer(TriangleDataBuffer.Get().get(), 0, sizeof(Frame), &m_Frame);

    //send lightdata to gpu
    GraphicsContext::CopyDataToBuffer(LightDataBuffer.Get().get(), 0, sizeof(Light) * m_Frame.LightCount,
m_Lights.data());

    EndCopyToBuffer();

}
```

```
PennyRender::End()
-----------------------------------------------------------------------------------------------
Purpose

The End() function finalizes the frame rendering process by:
Updating the frame data with the camera and lighting information.
Copying updated frame data and light data to the GPU.
Completing any necessary buffer copy operations for the current frame.

-----------------------------------------------------------------------------------------------
Function Overview
-----------------------------------------------------------------------------------------------
Updating Frame Data

m_Frame.Matrices = m_CameraData;
m_Frame.LightCount = m_Lights.size();
m_Frame.CameraMatrix = glm::inverse(m_CameraData.ViewMatrix);

 -----------------------------------------------------------------------------------------------
Camera and Lighting Data:
```

m_Frame.Matrices is updated with the camera data (m_CameraData), ensuring the frame has the latest view and projection matrix information.
m_Frame.LightCount is updated to reflect the number of lights in the scene (m_Lights.size()), allowing the shader to know how many lights to process.
m_Frame.CameraMatrix stores the inverse view matrix (glm::inverse(m_CameraData.ViewMatrix)), which will be used in shaders to transform world space coordinates into camera space.

Copying Updated Frame Data to GPU

```
GraphicsContext::CopyDataToBuffer(TriangleDataBuffer.Get().get(), 0, sizeof(Frame), &m_Frame);
```

The updated frame data (m_Frame) is copied to the GPU using the TriangleDataBuffer. This will allow the GPU to access the latest camera and lighting data during rendering.
The data is copied starting from offset 0, and the size of the data being copied is the size of the Frame structure.

Sending Light Data to GPU

```
GraphicsContext::CopyDataToBuffer(LightDataBuffer.Get().get(), 0, sizeof(Light) * m_Frame.LightCount, m_Lights.data());
```

Light data is copied to the GPU using the LightDataBuffer. The light data is passed to the shader, and the number of lights (m_Frame.LightCount) determines how much light data will be copied.
The light data is represented as an array (m_Lights.data()), and the size of the data copied is sizeof(Light) * m_Frame.LightCount.

Finalizing Buffer Copy Operations

```
EndCopyToBuffer();
```

EndCopyToBuffer() is called to perform any additional copying of buffer data necessary at the end of the frame.
This function handles any post-processing or final buffer updates required for the GPU.

Output

The function does not return anything but ensures that:
The frame data is updated and available for the next stage in the graphics pipeline.
The light data is sent to the GPU and will be used in shaders for lighting calculations.
The final buffer copy operations are completed to ensure the GPU has all necessary data.

Conceptual Summary
The End() function finalizes the rendering process by:
Updating the frame with the latest camera and light information.

```
Copying updated frame data and light data to GPU buffers.
Finalizing buffer copy operations to ensure the GPU is prepared for the next frame.
```

```
void PennyRender::renderToScreen_end(MultiResource<RenderTargetViewRef> OutputRTV, MultiResource<TextureSRVRef>
OutputSRV) {
////next frame
++OutputSRV;
++OutputRTV;
}
```

PennyRender::renderToScreen_end()

Purpose

The renderToScreen_end() function is used to finalize the process of rendering to the screen by updating the render target and shader resource view (SRV) for the next frame.

Function Overview

Incrementing the Render Target View (RTV)

```
++OutputRTV;
```

Purpose: This operation increments the Render Target View (RTV) to point to the next render target in the sequence.
Effect: This essentially prepares the system to render the next frame to a new target, moving forward in the sequence of render targets.

Incrementing the Shader Resource View (SRV)

```
++OutputSRV;
```

Purpose: Similar to the RTV, this operation increments the Shader Resource View (SRV) to the next texture in the sequence.
Effect: It ensures that the next texture resource (for example, a texture containing the result of the current frame's rendering) is used in subsequent shader operations, typically for post-processing or any other shader-based effects.

Conceptual Summary

The renderToScreen_end() function serves as a mechanism to increment the render target and shader resource view pointers, effectively preparing them for the next frame. This allows the system to move from one frame's output to the next frame's input seamlessly, ensuring continuous rendering.

```
void PennyRender::SetFrameData(FrameData newData)
{
```

m_CameraData = newData;
}

PennyRender::SetFrameData()

-
Purpose

The SetFrameData() function is used to update the camera data for the current frame. It takes a FrameData object as input and assigns it to the m_CameraData member variable.

-
Function Overview

-
Updating the Camera Data

m_CameraData = newData;

Purpose: This line of code assigns the provided newData (of type FrameData) to the m_CameraData member variable.
Effect: The camera data for the current frame is updated. This includes information such as the view and projection matrices, which are typically used for rendering the scene from the camera's perspective.

Conceptual Summary

The SetFrameData() function is a simple setter that allows external code to update the camera data for the current frame. This is important for scenarios where the camera might change during the course of the program, and new camera data needs to be used for rendering.

```
void PennyRender::DrawVertices(Vertex* vertexBuffer, size_t vertexcount, uint32_t* indexbuffer, size_t
indexcount, InstanceData* data, int CurrentRenderPass_in)
{
    size_t VBSize = vertexcount * Vertex::VertexLayout().GetStride();
    size_t IBSize = sizeof(int32_t) * indexcount;

    GraphicsContext::CopyDataToBuffer(TriangleDataBuffer.Get().get(), m_Offset, VBSize, vertexBuffer);
    GraphicsContext::CopyDataToBuffer(TriangleDataBuffer.Get().get(), m_Offset + VBSize, IBSize,
indexbuffer);

    Skateboard::VertexBufferView vbv{};
    Skateboard::IndexBufferView ibv{};
```

```
    vbv.m_Offset = m_Offset;
    vbv.m_ParentResource = TriangleDataBuffer.Get();
    vbv.m_VertexCount = vertexcount;
    vbv.m_VertexStride = Vertex::VertexLayout().GetStride();


    ibv.m_Offset = m_Offset + VBSize;
    ibv.m_Format = IndexFormat::bit32;
    ibv.m_ParentResource = TriangleDataBuffer.Get();
    ibv.m_IndexCount = indexcount;


    m_Offset += VBSize + IBSize;


    DrawVBIB(&vbv, &ibv, data, CurrentRenderPass_in);
}
```

PennyRender::DrawVertices()

Purpose

The DrawVertices() function handles the process of copying vertex and index data to a buffer and then initiating the drawing process. It takes a vertex buffer, index buffer, instance data, and the current render pass as inputs. This function is primarily used to render geometry based on the provided data.

🦴 Function Overview

Calculating Buffer Sizes

```
size_t VBSize = vertexcount * Vertex::VertexLayout().GetStride();
size_t IBSize = sizeof(int32_t) * indexcount;
```

Purpose:

The size of the vertex buffer (VBSize) is calculated by multiplying the number of vertices (vertexcount) by the stride of each vertex (which can be obtained via Vertex::VertexLayout().GetStride()).
The size of the index buffer (IBSize) is calculated by multiplying the number of indices (indexcount) by the size of an index (assumed to be sizeof(int32_t)).

Copying Data to Triangle Data Buffer

```
GraphicsContext::CopyDataToBuffer(TriangleDataBuffer.Get().get(), m_Offset, VBSize, vertexBuffer);
GraphicsContext::CopyDataToBuffer(TriangleDataBuffer.Get().get(), m_Offset + VBSize, IBSize,
indexbuffer);
```

```
Purpose:
Copies the vertex data to the TriangleDataBuffer starting from the offset m_Offset.
Then, copies the index data to the buffer at the new offset, which is m_Offset + VBSize.
Effect: The vertex and index data are placed into a GPU buffer for rendering.

-------------------------------------------------------------------------------------------
Setting Up Vertex and Index Buffer Views

Skateboard::VertexBufferView vbv{};
Skateboard::IndexBufferView ibv{};

Purpose:
Initializes VertexBufferView (vbv) and IndexBufferView (ibv), which describe how to access the vertex and
index data in the buffer.

-------------------------------------------------------------------------------------------
Configuring Vertex Buffer View

vbv.m_Offset = m_Offset;
vbv.m_ParentResource = TriangleDataBuffer.Get();
vbv.m_VertexCount = vertexcount;
vbv.m_VertexStride = Vertex::VertexLayout().GetStride();

Purpose:
The VertexBufferView (vbv) is configured with the following:
m_Offset: The offset from where the vertex data starts in the buffer.
m_ParentResource: The buffer resource that holds the vertex data.
m_VertexCount: The total number of vertices.
m_VertexStride: The stride (size) of each vertex.

-------------------------------------------------------------------------------------------
Configuring Index Buffer View

ibv.m_Offset = m_Offset + VBSize;
ibv.m_Format = IndexFormat::bit32;
ibv.m_ParentResource = TriangleDataBuffer.Get();
ibv.m_IndexCount = indexcount;

Purpose:
The IndexBufferView (ibv) is configured with the following:
m_Offset: The offset from where the index data starts in the buffer (m_Offset + VBSize).
m_Format: The format of the indices (bit32 indicates 32-bit indices).
m_ParentResource: The buffer resource that holds the index data.
m_IndexCount: The total number of indices.

-------------------------------------------------------------------------------------------
Updating the Offset
```

m_Offset += VBSize + IBSize;

Purpose: The offset (m_Offset) is updated by adding the sizes of the vertex and index buffers. This ensures that the next time vertex or index data is copied, it will be placed in the correct location in the buffer.

Calling Draw Function

DrawVBIB(&vbv, &ibv, data, CurrentRenderPass_in);

Purpose: The DrawVBIB() function is called to initiate the drawing process with the configured vertex and index buffer views (vbv and ibv), the instance data (data), and the current render pass (CurrentRenderPass_in).

Output

Effect: This function initiates the drawing process by sending the vertex and index data to the GPU, using the TriangleDataBuffer. It sets up the buffers and calls the DrawVBIB() function to actually issue the draw command.
No return value.

Conceptual Summary

The DrawVertices() function is responsible for copying vertex and index data to a buffer in the GPU, configuring the necessary buffer views, updating the offset, and then triggering the drawing process. This is a typical function used in rendering pipelines to handle geometry rendering for a frame, with support for vertex and index data, as well as handling multiple render passes and instance data.

```
void PennyRender::DrawVBIB(VertexBufferView* vb, IndexBufferView* ib, InstanceData* data, int
CurrentRenderPass_in)
{
    if (m_Pipeline_dirty)
    {
        m_Pipeline_dirty = false;
        //RenderCommand::SetPipelineState(Pipelines[0].get());
        //RenderCommand::SetPipelineState(Pipelines[m_PipelineSelection].get());
        RenderCommand::SetPipelineState(Pipelines[CURRENT_STATE_].get());
    }



    if (data)
    {
        GraphicsContext::CopyDataToBuffer(InstanceDataBuffer.Get().get(), sizeof(InstanceData) *
m_InstanceDataForks, sizeof(InstanceData), data);
        RenderCommand::SetInline32bitDataGraphics(0, &m_InstanceDataForks, 1);
```

```
        ++m_InstanceDataForks;
    }
    else
    {
        //default value
        int d = 0;
        RenderCommand::SetInline32bitDataGraphics(0, &d, 1);
    }


    RenderCommand::SetInlineResourceViewGraphics(1, TriangleDataBuffer.Get().get(), cbvdesc,
ViewAccessType_ConstantBuffer);
    RenderCommand::SetInlineResourceViewGraphics(2, InstanceDataBuffer.Get().get(), sbvdesc,
ViewAccessType_GpuRead);
    RenderCommand::SetInlineResourceViewGraphics(3, LightDataBuffer.Get().get(), lightsbvdesc,
ViewAccessType_GpuRead);


    DrawVBIB_SetInlineResourceViewGraphics();//


    RenderCommand::SetIndexBuffer(ib);
    RenderCommand::SetVertexBuffer(vb, 1);


    RenderCommand::SetPrimitiveTopology(Topology);


    RenderCommand::DrawIndexed(0, 0, ib->m_IndexCount);


    if (m_VisualiseNormals)//
    {
        RenderCommand::SetPipelineState(NormalVisualizer.get());
        RenderCommand::DrawIndexed(0, 0, ib->m_IndexCount);
        RenderCommand::SetPipelineState(Pipelines[CURRENT_STATE_].get());
    }



}
```

PennyRender::DrawVBIB()

Purpose

The DrawVBIB() function is responsible for managing the drawing process for a given set of vertex and index buffers. It ensures that pipeline states, instance data, and resources are properly set before issuing a draw call. This function handles both regular and instance-based rendering, allowing for dynamic and efficient drawing commands to be issued to the GPU.

Function Overview

Pipeline State Check

```
if (m_Pipeline_dirty)
{
    m_Pipeline_dirty = false;
    RenderCommand::SetPipelineState(Pipelines[CURRENT_STATE_].get());
}
```

The function first checks if the pipeline state is dirty (i.e., needs updating). If so, it sets the appropriate pipeline state using the current state from Pipelines[CURRENT_STATE_]. This ensures that any changes in the pipeline configuration are reflected before drawing.

Handling Instance Data

```
if (data)
{
    GraphicsContext::CopyDataToBuffer(InstanceDataBuffer.Get().get(), sizeof(InstanceData) *
m_InstanceDataForks, sizeof(InstanceData), data);
    RenderCommand::SetInline32bitDataGraphics(0, &m_InstanceDataForks, 1);
    ++m_InstanceDataForks;
}
else
{
    int d = 0;
    RenderCommand::SetInline32bitDataGraphics(0, &d, 1);
}
```

If instance data (data) is provided, it copies the instance data to the InstanceDataBuffer at an offset determined by m_InstanceDataForks. It also increments the m_InstanceDataForks counter to track how many instance data blocks have been processed.
If no instance data is provided, it sets a default value (0) to indicate no specific instance data for the current draw call.

Setting Inline Resource Views

```
RenderCommand::SetInlineResourceViewGraphics(1, TriangleDataBuffer.Get().get(), cbvdesc,
ViewAccessType_ConstantBuffer);
RenderCommand::SetInlineResourceViewGraphics(2, InstanceDataBuffer.Get().get(), sbvdesc,
ViewAccessType_GpuRead);
RenderCommand::SetInlineResourceViewGraphics(3, LightDataBuffer.Get().get(), lightsbvdesc,
ViewAccessType_GpuRead);
```

These lines set up the resource views for different buffers:

TriangleDataBuffer: The vertex and index data.
InstanceDataBuffer: The instance-specific data.
LightDataBuffer: The light data used for shading or lighting effects.

The data is passed with different access types, such as ConstantBuffer and GpuRead, which define how the GPU will access these resources.

Setting Additional Resource Views

```
DrawVBIB_SetInlineResourceViewGraphics();
```

This function call is used to handle additional resource views (related to post-processing stages of the pipeline).

Setting Buffers and Primitive Topology

```
RenderCommand::SetIndexBuffer(ib);
RenderCommand::SetVertexBuffer(vb, 1);
RenderCommand::SetPrimitiveTopology(Topology);
```

Sets the index buffer (ib), vertex buffer (vb), and the primitive topology (Topology). The topology defines the way vertices are interpreted, such as TriangleList, LineStrip, etc.

Draw Indexed Command

```
RenderCommand::DrawIndexed(0, 0, ib->m_IndexCount);
```

This is the main draw command that issues a draw call using indexed geometry. It uses the index count from the provided IndexBufferView (ib) to determine how many indices to process.

Visualizing Normals

```
if (m_VisualiseNormals)
{
    RenderCommand::SetPipelineState(NormalVisualizer.get());
    RenderCommand::DrawIndexed(0, 0, ib->m_IndexCount);
    RenderCommand::SetPipelineState(Pipelines[CURRENT_STATE_].get());
}
```

If normal visualization is enabled (m_VisualiseNormals), it switches the pipeline state to a "normal visualizer" state and issues another draw call. This is often used in debugging to visualize how normals are applied to the geometry.
After visualizing the normals, the pipeline state is reset to the current state to continue with the regular rendering process.

Output

This function sets up the necessary resources, instance data, and pipeline states to render geometry using vertex and index buffers. It also optionally handles normal visualization for debugging or visualization purposes.
The function does not return any value but directly interacts with the GPU to manage rendering.

Conceptual Summary
The DrawVBIB() function is a critical part of the rendering pipeline, responsible for configuring resources and issuing draw commands. It checks if the pipeline state is dirty and ensures that all necessary resources are set up for rendering. It also provides flexibility for instance-based rendering and optional debugging features, like normal visualization.

```
void PennyRender::renderScreen(Texture renderTexture) {

    SetRenderTargets_(OutputTexture,OutputRTV);

    //RenderCommand::ClearRenderTargets(OutputRTVs_0.Get().get(), 1, float4(0, 0, 0, 0));

    //rendering scene
    Begin();//

    uint32_t Indexdata[]
    {
        0,1,2
    };

    std::vector<pen::Vertex> movedvertices = vertices_screen;

    pen::InstanceData testdata;

    testdata.TextureIndex = 10;

    for (auto& a : movedvertices) { a.Position += float3(2.5, 0, 0); }
    DrawVertices(movedvertices.data(), 3, Indexdata, 3, &testdata);



    pen::Vertex Quad[4] =
    {
        {float3(-1,1,0),float3(0,0,0),float2(0,0),    float3(0,0,-1)    },
        {float3(-1,-1,0),float3(0,0,0),float2(0,1),    float3(0,0,-1)    },
        {float3(1,-1,0),float3(0,0,0),float2(1,1),    float3(0,0,-1)    },
```

```
        {float3(1,1,0),float3(0,0,0),float2(1,0),    float3(0,0,-1)    }
    };

    uint32_t quadindices[6] =
    {
        0,1,2,2,3,0
    };

    testdata.TextureIndex = renderTexture->GetViewIndex();

    matrix World = glm::translate(float3(-5, 0, 0));
    testdata.World = World * glm::translate(float3(5, 0, -5)) * (90.f, 1.f, 1.f) * glm::scale(float3(9,
5, 1)); //find out proper window size later (maybe ask naman if he knows?)
    testdata.ColourScale = float4(1, 1, 1, 0.9);
    testdata.SpecularColor = SpecularC;
    testdata.SpecularPower = SpecularPower;
    testdata.SpecularWeight = SpecularWeight;

    DrawVertices(Quad, 4, quadindices, 6, &testdata);



    End();

    //end of render

    GraphicsContext::SetRenderTargetToBackBuffer();

}
```

```
PennyRender::renderScreen()
-------------------------------------------------------------------------------------------------------
--------------
Purpose
-------------------------------------------------------------------------------------------------------
--------------

The renderScreen() function is responsible for rendering the scene onto a texture and then drawing a quad
to the screen, using the specified Texture as the source for the quad. It sets up the necessary render
targets, handles the rendering of vertices and index data, and applies instance-specific transformation
and texture data before completing the rendering process.

-------------------------------------------------------------------------------------------------------
--------------
Function Overview
```

Setting Render Targets

```
SetRenderTargets_(OutputTexture, OutputRTV);
```

This function call sets the render targets for the current rendering pass. OutputTexture is the texture where the scene will be rendered, and OutputRTV is the render target view used for that texture.

Beginning the Render

```
Begin();
```

This function call starts the rendering process by preparing necessary buffers and setting up the pipeline. It’s a prerequisite for rendering any geometry.

Defining Quad Vertices for Full-Screen Rendering

```
pen::Vertex Quad[4] = {
    {float3(-1,1,0),float3(0,0,0),float2(0,0), float3(0,0,-1)},
    {float3(-1,-1,0),float3(0,0,0),float2(0,1), float3(0,0,-1)},
    {float3(1,-1,0),float3(0,0,0),float2(1,1), float3(0,0,-1)},
    {float3(1,1,0),float3(0,0,0),float2(1,0), float3(0,0,-1)}
};
```

Defines the vertices of a quad, which will be rendered as a full-screen object. The quad has four corners, with each vertex having a position (float3), a texture coordinate (float2), and a normal vector (float3).

Defining Quad Indices

```
uint32_t quadindices[6] = { 0, 1, 2, 2, 3, 0 };
```

Defines the indices for the quad. The indices refer to the vertices in Quad[] and determine how the vertices are connected to form two triangles (one for each half of the quad).

Setting Instance Data for Quad

```
testdata.TextureIndex = renderTexture->GetViewIndex();
matrix World = glm::translate(float3(-5, 0, 0));
testdata.World = World * glm::translate(float3(5, 0, -5)) * (90.f, 1.f, 1.f) * glm::scale(float3(9, 5,
1));
testdata.ColourScale = float4(1, 1, 1, 0.9);
testdata.SpecularColor = SpecularC;
testdata.SpecularPower = SpecularPower;
testdata.SpecularWeight = SpecularWeight;
```

Sets the instance data (testdata) for the quad:

TextureIndex: Uses the texture's view index (renderTexture->GetViewIndex()).
World: Applies a series of transformations, including translation, rotation, and scaling to the quad.
ColourScale, SpecularColor, SpecularPower, and SpecularWeight: Additional data for shading, lighting, and material properties (such as color scaling and specular effects).

Drawing the Quad

```
DrawVertices(Quad, 4, quadindices, 6, &testdata);
```

This call renders the quad using the vertices (Quad), indices (quadindices), and the instance data (testdata). The quad is drawn to the screen, using the texture specified in testdata.

Ending the Render

```
End();
```

This function call ends the rendering process. It handles any final updates to buffers, synchronizes resources, or finalizes the frame for output.

Setting the Render Target to the Back Buffer

```
GraphicsContext::SetRenderTargetToBackBuffer();
```

After the custom rendering is completed, this function call restores the render target back to the back buffer (i.e., the final screen output), preparing for the next frame or draw call.

Output

This function performs rendering to a texture and draws the resulting texture as a full-screen quad. It allows for advanced rendering effects, such as post-processing or offscreen rendering, followed by displaying the rendered texture.
The output is rendered on the back buffer (screen), completing the frame's visual output.

Conceptual Summary

The renderScreen() function is designed to facilitate offscreen rendering onto a texture and then display that texture on the screen as a quad. This is useful for various advanced graphics techniques like post-processing, shadow mapping, or screen-space effects. It efficiently manages the render targets, applies transformations, and draws both simple geometry and textured surfaces.

```
void PennyRender::MultiRenderPass(Texture renderTexture) { // not working


/*    renderScreenTest(renderTexture, OutputTextures_[0], OutputRTVs_[0], OutputSRVs_[0]);
    Texture tex = OutputSRVs_[0].Get();
    renderToScreen_end(OutputRTVs_[0], OutputSRVs_[0]);


    renderScreenTest(renderTexture, OutputTextures_1, OutputRTVs_1, OutputSRVs_1);*/
    renderScreenPass(renderTexture, OutputTextures_0, OutputRTVs_0, OutputSRVs_0); //
    Texture tex = OutputSRVs_0.Get();
    renderToScreen_end(OutputRTVs_0, OutputSRVs_0);
    renderScreen(tex);//always has to be the last one (as its "texture" -SRV- is what gets pasted through
to the preive screen)
    //renderToScreen_end(OutputRTVs_0, OutputSRVs_0);


}
```

PennyRender::MultiRenderPass()

Purpose

The MultiRenderPass() function is intended to execute multiple rendering passes in sequence. It handles rendering to offscreen textures, followed by final rendering to the screen. Although currently commented out and non-functional, the general idea is to process different effects or steps in multiple passes, updating the output after each pass, and finally displaying the result.

Function Overview

Rendering the First Pass to an Offscreen Texture

```
renderScreenPass(renderTexture, OutputTextures_0, OutputRTVs_0, OutputSRVs_0);
```

This line calls renderScreenPass() to perform the first render pass. The pass renders the scene to the texture specified by OutputTextures_0, using the corresponding render target view (OutputRTVs_0) and shader resource view (OutputSRVs_0). This is part of an offscreen rendering technique, for post-processing effects or intermediate results.

Getting the Texture for Further Processing

```
Texture tex = OutputSRVs_0.Get();
```

After completing the first render pass, this line retrieves the texture from OutputSRVs_0 (the shader resource view) that contains the output of the first render pass. This texture will be used in subsequent passes or for display.

Finalizing the Render to Screen

```
renderToScreen_end(OutputRTVs_0, OutputSRVs_0);

This function call, renderToScreen_end(), finalizes the rendering for the current pass by updating the
render target views and shader resource views. It effectively marks the end of the render pass and
prepares the scene for displaying or processing further.

--------------------------------------------------------------------------------------
Rendering the Final Result to the Screen

renderScreen(tex);

This function call uses the texture (tex) from the previous render pass and renders it to the screen as a
full-screen quad. This is usually the final step in the rendering pipeline when applying post-processing
effects or displaying the final result.

--------------------------------------------------------------------------------------
Commented Out / Additional Render Passes
/*
renderScreenTest(renderTexture, OutputTextures_[0], OutputRTVs_[0], OutputSRVs_[0]);
Texture tex = OutputSRVs_[0].Get();
renderToScreen_end(OutputRTVs_[0], OutputSRVs_[0]);


renderScreenTest(renderTexture, OutputTextures_1, OutputRTVs_1, OutputSRVs_1);
*/

Purpose:
The commented-out code represents additional render passes that were planned but are currently not
functional. These passes would render to different textures and views (OutputTextures_[0],
OutputTextures_1, etc.), and the result would then be passed through the screen render process.
These lines suggest the possibility of handling multiple intermediate render passes (e.g., different
visual effects or steps).

--------------------------------------------------------------------------------------
Output

Effect:
The function ensures that multiple render passes are performed and that the final rendered image is
displayed on the screen. The function first renders to an offscreen texture, then processes the result,
and finally renders the processed texture to the screen.
The code hints that the function was designed for handling multiple render passes, but it is currently
incomplete or not working.

--------------------------------------------------------------------------------------
Conceptual Summary

The MultiRenderPass() function is designed to facilitate a multi-pass rendering pipeline where
intermediate results are rendered offscreen, processed, and then displayed. Although not fully functional
in its current form, this pattern is commonly used in rendering techniques like post-processing effects,
```

shadow mapping, or deferred rendering, where each pass serves a specific purpose, such as applying a filter or calculating lighting.
The function concludes with rendering the final processed texture to the screen.

```
void PennyRender::renderToScreen_end(MultiResource<RenderTargetViewRef> OutputRTV, MultiResource<TextureSRVRef>
OutputSRV) {
////next frame
++OutputSRV;
++OutputRTV;
}
```

PennyRender:: renderToScreen_end(MultiResource<RenderTargetViewRef> OutputRTV, MultiResource<TextureSRVRef> OutputSRV)

Purpose

The renderToScreen_end() function is used to finalize the process of rendering to the screen by updating the render target and shader resource view (SRV) for the next frame.

Function Overview

Incrementing the Render Target View (RTV)

```
++OutputRTV;
```

Purpose: This operation increments the Render Target View (RTV) to point to the next render target in the sequence.
Effect: This essentially prepares the system to render the next frame to a new target, moving forward in the sequence of render targets.

Incrementing the Shader Resource View (SRV)

```
++OutputSRV;
```

Purpose: Similar to the RTV, this operation increments the Shader Resource View (SRV) to the next texture in the sequence.
Effect: It ensures that the next texture resource (for example, a texture containing the result of the current frame's rendering) is used in subsequent shader operations, typically for post-processing or any other shader-based effects.

Conceptual Summary

The renderToScreen_end() function serves as a mechanism to increment the render target and shader resource view pointers, effectively preparing them for the next frame. This allows the system to move from one frame's output to the next frame's input seamlessly, ensuring continuous rendering.

Drawing to the render target

```
void PennyRender::renderToScreen_init() {

    //--textDesc init
    TextureDesc textDesc{};

    textDesc.Format = DataFormat_DEFAULT_BACKBUFFER;
    textDesc.AccessFlags = ResourceAccessFlag_GpuRead | ResourceAccessFlag_GpuWrite;
    textDesc.Type = TextureType_RenderTarget;
    textDesc.Dimension = TextureDimension_Texture2D;
    textDesc.Width = GraphicsContext::GetClientWidth();
    textDesc.Height = GraphicsContext::GetClientHeight();
    textDesc.Depth = 1;
    textDesc.Mips = 1;
    textDesc.Clear = ClearValue(float4(0,0,0,0));

    OutputTexture.ForEach([&](auto& Texture) { Texture = ResourceFactory::CreateTextureBuffer(textDesc);
});

    //create output view desc
    TextureViewDesc srv{};
    srv.Dimension = TextureDimension_Texture2D;
    srv.ArraySize = 1;
    srv.Format = DataFormat_DEFAULT_BACKBUFFER;
    srv.MipLevels = 1;
    srv.MipSlice = 0;

    //RenderTargetDesc rtv_desc{.MipSlice = 0, .PlaneSlice = 0};
    //OutputRTV.ForEach([&, i = 0](auto& view) mutable -> void {view =
ResourceFactory::CreateRenderTargetView(&rtv_desc, OutputTexture[i]); i++; });
    OutputRTV.ForEach([&, i = 0](auto& view) mutable -> void {view =
ResourceFactory::CreateRenderTargetView(0, OutputTexture[i]); i++; }); //the 0 here will cout some
warning messages, but einar said to ignore it since it works and doesnt negatively effect anything
    OutputSRV.ForEach([&, i = 0](auto& view) mutable -> void {view =
ResourceFactory::CreateTextureShaderResourceView(srv, OutputTexture[i]); i++; });



}
```

PennyRender::renderToScreen_init

Purpose

Initializes the output render targets and shader resource views used for rendering the final scene image to the screen. This setup supports post-processing, multi-pass rendering, and general output-to-texture workflows.

Function Overview

Texture Descriptor Setup

```
TextureDesc textDesc{};
textDesc.Format = DataFormat_DEFAULT_BACKBUFFER;
textDesc.AccessFlags = ResourceAccessFlag_GpuRead | ResourceAccessFlag_GpuWrite;
textDesc.Type = TextureType_RenderTarget;
...
textDesc.Width = GraphicsContext::GetClientWidth();
textDesc.Height = GraphicsContext::GetClientHeight();
```

Describes a 2D texture used as a render target with GPU read/write access.

Clear colour:

Set to transparent black (0,0,0,0) for clean slate rendering.

Create Render Target Textures

```
OutputTexture.ForEach([&](auto& Texture) {
    Texture = ResourceFactory::CreateTextureBuffer(textDesc);
});
```

Allocates the actual GPU texture resources based on textDesc — typically one for each render pass or buffer frame.

Shader Resource View Descriptor Setup

```
TextureViewDesc srv{};
srv.Dimension = TextureDimension_Texture2D;
srv.ArraySize = 1;
srv.Format = DataFormat_DEFAULT_BACKBUFFER;
srv.MipLevels = 1;
srv.MipSlice = 0;
```

Defines how the texture will be accessed in shaders — as a 2D resource with one mip level.

Create Render Target Views (RTV)

```
OutputRTV.ForEach([&, i = 0](auto& view) mutable {
    view = ResourceFactory::CreateRenderTargetView(0, OutputTexture[i]);
    i++;
});
```

For each output texture, a corresponding render target view is created so it can be written to by the GPU.

Note:

The hardcoded 0 might emit a warning (as noted in the comment), but is safe per developer comments.

Create Shader Resource Views (SRV)

```
OutputSRV.ForEach([&, i = 0](auto& view) mutable {
    view = ResourceFactory::CreateTextureShaderResourceView(srv, OutputTexture[i]);
    i++;
});
```

Creates a view that allows shaders to sample from the rendered texture (e.g., for post-processing).

Conceptual Summary

This function:

Allocates GPU render targets at screen resolution.

Sets up render target views to render into.

Sets up shader resource views to read from.

It’s foundational to enabling post-processing and effects that render from one texture and display the result on screen or in a subsequent pass.

```
void PennyRender::renderToScreen_init_pass(MultiResource<TextureBufferRef> OutputTexture,
MultiResource<RenderTargetViewRef> OutputRTV, MultiResource<TextureSRVRef>  OutputSRV)
{
    //--textDesc init
    TextureDesc textDesc{};
    //textDesc.Format = renderTexture.get()->GetFormat();
    textDesc.Format = DataFormat_DEFAULT_BACKBUFFER;
    textDesc.AccessFlags = ResourceAccessFlag_GpuRead | ResourceAccessFlag_GpuWrite;
    textDesc.Type = TextureType_RenderTarget;
    textDesc.Dimension = TextureDimension_Texture2D;
    //textDesc.Width = renderTexture.get()->GetWidth(); //textDesc.Height = renderTexture.get()-
>GetHeight();
    textDesc.Width = GraphicsContext::GetClientWidth();
```

```
    textDesc.Height = GraphicsContext::GetClientHeight();
    textDesc.Depth = 1;
    textDesc.Mips = 1;
    textDesc.Clear = ClearValue(float4(0, 0, 0, 0));

    OutputTexture.ForEach([&](auto& Texture) { Texture = ResourceFactory::CreateTextureBuffer(textDesc);
});

    //create srv disc
    TextureViewDesc srv{};
    srv.Dimension = TextureDimension_Texture2D;
    srv.ArraySize = 1;
    srv.Format = DataFormat_DEFAULT_BACKBUFFER;
    srv.MipLevels = 1;
    srv.MipSlice = 0;

    RenderTargetDesc rtv_desc{ .MipSlice = 0, .PlaneSlice = 0 };

    //OutputRTV.ForEach([&, i = 0](auto& view) mutable -> void {view =
ResourceFactory::CreateRenderTargetView(&rtv_desc, OutputTexture[i]); i++; });
    OutputRTV.ForEach([&, i = 0](auto& view) mutable -> void {view =
ResourceFactory::CreateRenderTargetView(0, OutputTexture[i]); i++; });
    OutputSRV.ForEach([&, i = 0](auto& view) mutable -> void {view =
ResourceFactory::CreateTextureShaderResourceView(srv, OutputTexture[i]); i++; });

}
```

```
PennyRender::renderToScreen_init
-------------------------------------------------------------------------------------------
Purpose

Initializes the output render targets and shader resource views used for rendering the final scene image
to the screen. This setup supports post-processing, multi-pass rendering, and general output-to-texture
workflows.

-------------------------------------------------------------------------------------------
Function Overview
-------------------------------------------------------------------------------------------
Texture Descriptor Setup

TextureDesc textDesc{};
textDesc.Format = DataFormat_DEFAULT_BACKBUFFER;
textDesc.AccessFlags = ResourceAccessFlag_GpuRead | ResourceAccessFlag_GpuWrite;
textDesc.Type = TextureType_RenderTarget;
...
```

```
textDesc.Width = GraphicsContext::GetClientWidth();
textDesc.Height = GraphicsContext::GetClientHeight();



 Describes a 2D texture used as a render target with GPU read/write access.
Clear colour:
 Set to transparent black (0,0,0,0) for clean slate rendering.

-------------------------------------------------------------------------------------
Create Render Target Textures

OutputTexture.ForEach([&](auto& Texture) {
    Texture = ResourceFactory::CreateTextureBuffer(textDesc);
});

 Allocates the actual GPU texture resources based on textDesc — typically one for each render pass or
buffer frame.

-------------------------------------------------------------------------------------
Shader Resource View Descriptor Setup

TextureViewDesc srv{};
srv.Dimension = TextureDimension_Texture2D;
srv.ArraySize = 1;
srv.Format = DataFormat_DEFAULT_BACKBUFFER;
srv.MipLevels = 1;
srv.MipSlice = 0;


Defines how the texture will be accessed in shaders — as a 2D resource with one mip level.

-------------------------------------------------------------------------------------
Create Render Target Views (RTV)

OutputRTV.ForEach([&, i = 0](auto& view) mutable {
    view = ResourceFactory::CreateRenderTargetView(0, OutputTexture[i]);
    i++;
});



 For each output texture, a corresponding render target view is created so it can be written to by the
GPU.
Note:
 The hardcoded 0 might emit a warning (as noted in the comment), but is safe per developer comments.

-------------------------------------------------------------------------------------
Create Shader Resource Views (SRV)
```

```
OutputSRV.ForEach([&, i = 0](auto& view) mutable {
    view = ResourceFactory::CreateTextureShaderResourceView(srv, OutputTexture[i]);
    i++;
});


 Creates a view that allows shaders to sample from the rendered texture (e.g., for post-processing).

-------------------------------------------------------------------------------------------
Conceptual Summary

This function:
Allocates GPU render targets at screen resolution.
Sets up render target views to render into.
Sets up shader resource views to read from.
It’s foundational to enabling post-processing and effects that render from one texture and display the
result on screen or in a subsequent pass.
```

```
void PennyRender::renderScreen(Texture renderTexture) {


    SetRenderTargets_(OutputTexture,OutputRTV);


    //RenderCommand::ClearRenderTargets(OutputRTVs_0.Get().get(), 1, float4(0, 0, 0, 0));


    //rendering scene
    Begin();//


    uint32_t Indexdata[]
    {
        0,1,2
    };


    std::vector<pen::Vertex> movedvertices = vertices_screen;


    pen::InstanceData testdata;


    testdata.TextureIndex = 10;


    for (auto& a : movedvertices) { a.Position += float3(2.5, 0, 0); }
    DrawVertices(movedvertices.data(), 3, Indexdata, 3, &testdata);



    pen::Vertex Quad[4] =
    {
```

```
        {float3(-1,1,0),float3(0,0,0),float2(0,0),    float3(0,0,-1)    },
        {float3(-1,-1,0),float3(0,0,0),float2(0,1),    float3(0,0,-1)    },
        {float3(1,-1,0),float3(0,0,0),float2(1,1),    float3(0,0,-1)    },
        {float3(1,1,0),float3(0,0,0),float2(1,0),    float3(0,0,-1)    }
    };

    uint32_t quadindices[6] =
    {
        0,1,2,2,3,0
    };

    testdata.TextureIndex = renderTexture->GetViewIndex();

    matrix World = glm::translate(float3(-5, 0, 0));
    testdata.World = World * glm::translate(float3(5, 0, -5)) * (90.f, 1.f, 1.f) * glm::scale(float3(9,
5, 1)); //find out proper window size later (maybe ask naman if he knows?)
    testdata.ColourScale = float4(1, 1, 1, 0.9);
    testdata.SpecularColor = SpecularC;
    testdata.SpecularPower = SpecularPower;
    testdata.SpecularWeight = SpecularWeight;

    DrawVertices(Quad, 4, quadindices, 6, &testdata);

    //current_instancedata = testdata;///this is where we get data for the multiple passes

    End();

    //end of render

    GraphicsContext::SetRenderTargetToBackBuffer();

}
```

```
PennyRender::renderScreen()
-------------------------------------------------------------------------------------------------------
--------------
Purpose
-------------------------------------------------------------------------------------------------------
--------------

The renderScreen() function is responsible for rendering the scene onto a texture and then drawing a quad
to the screen, using the specified Texture as the source for the quad. It sets up the necessary render
targets, handles the rendering of vertices and index data, and applies instance-specific transformation
and texture data before completing the rendering process.
```

Function Overview

Setting Render Targets

```
SetRenderTargets_(OutputTexture, OutputRTV);
```

This function call sets the render targets for the current rendering pass. OutputTexture is the texture where the scene will be rendered, and OutputRTV is the render target view used for that texture.

Beginning the Render

```
Begin();
```

This function call starts the rendering process by preparing necessary buffers and setting up the pipeline. It’s a prerequisite for rendering any geometry.

----------Defining Quad Vertices for Full-Screen Rendering

```
pen::Vertex Quad[4] = {
    {float3(-1,1,0),float3(0,0,0),float2(0,0), float3(0,0,-1)},
    {float3(-1,-1,0),float3(0,0,0),float2(0,1), float3(0,0,-1)},
    {float3(1,-1,0),float3(0,0,0),float2(1,1), float3(0,0,-1)},
    {float3(1,1,0),float3(0,0,0),float2(1,0), float3(0,0,-1)}
};
```

Defines the vertices of a quad, which will be rendered as a full-screen object. The quad has four corners, with each vertex having a position (float3), a texture coordinate (float2), and a normal vector (float3).

Defining Quad Indices

```
uint32_t quadindices[6] = { 0, 1, 2, 2, 3, 0 };
```

Defines the indices for the quad. The indices refer to the vertices in Quad[] and determine how the vertices are connected to form two triangles (one for each half of the quad).

Setting Instance Data for Quad

```
testdata.TextureIndex = renderTexture->GetViewIndex();
matrix World = glm::translate(float3(-5, 0, 0));
testdata.World = World * glm::translate(float3(5, 0, -5)) * (90.f, 1.f, 1.f) * glm::scale(float3(9, 5,
1));
testdata.ColourScale = float4(1, 1, 1, 0.9);
```

```
testdata.SpecularColor = SpecularC;
testdata.SpecularPower = SpecularPower;
testdata.SpecularWeight = SpecularWeight;
```

Sets the instance data (testdata) for the quad:
TextureIndex: Uses the texture's view index (renderTexture->GetViewIndex()).
World: Applies a series of transformations, including translation, rotation, and scaling to the quad.
ColourScale, SpecularColor, SpecularPower, and SpecularWeight: Additional data for shading, lighting, and material properties (such as color scaling and specular effects).

Drawing the Quad

```
DrawVertices(Quad, 4, quadindices, 6, &testdata);
```

This call renders the quad using the vertices (Quad), indices (quadindices), and the instance data (testdata). The quad is drawn to the screen, using the texture specified in testdata.

Ending the Render

```
End();
```

This function call ends the rendering process. It handles any final updates to buffers, synchronizes resources, or finalizes the frame for output.

Setting the Render Target to the Back Buffer

```
GraphicsContext::SetRenderTargetToBackBuffer();
```

After the custom rendering is completed, this function call restores the render target back to the back buffer (i.e., the final screen output), preparing for the next frame or draw call.

Output

This function performs rendering to a texture and draws the resulting texture as a full-screen quad. It allows for advanced rendering effects, such as post-processing or offscreen rendering, followed by displaying the rendered texture.
The output is rendered on the back buffer (screen), completing the frame's visual output.

Conceptual Summary

The renderScreen() function is designed to facilitate offscreen rendering onto a texture and then display that texture on the screen as a quad. This is useful for various advanced graphics techniques like post-

processing, shadow mapping, or screen-space effects. It efficiently manages the render targets, applies transformations, and draws both simple geometry and textured surfaces.

```
  void SetRenderTargets_(MultiResource<TextureBufferRef> OutputTexture,
MultiResource<RenderTargetViewRef> OutputRTV) {

        TextureBarrier renderTargetBarrier{};
        renderTargetBarrier.SyncBefore = SKTBD_SYNC_PIXEL_SHADING;
        renderTargetBarrier.SyncAfter = SKTBD_SYNC_RENDER_TARGET;
        renderTargetBarrier.AccessBefore = SKTBD_ACCESS_COMMON;
        renderTargetBarrier.AccessAfter = SKTBD_ACCESS_RENDER_TARGET;
        renderTargetBarrier.LayoutBefore = SKTBD_LAYOUT_COMMON;
        renderTargetBarrier.LayoutAfter = SKTBD_LAYOUT_RENDER_TARGET;
        renderTargetBarrier.Resource = OutputTexture.Get().get();//
        renderTargetBarrier.SubresourceRange = TextureSubresourceRange();

        BarrierGroup grouptoRTV(&renderTargetBarrier);
        RenderCommand::Barrier(&grouptoRTV, 1);

        RenderCommand::SetRenderTargets(OutputRTV.Get().get(), 1,
GraphicsContext::GetDefaultDepthBuffer());



    }
```

```
void SetRenderTargets_(MultiResource<TextureBufferRef> OutputTexture, MultiResource<RenderTargetViewRef>
OutputRTV)
-------------------------------------------------------------------------------------------------------
--
Purpose

Prepares and sets the render target for rendering by:
Transitioning the texture resource to a render-target-compatible state using a barrier.
Binding the appropriate render target view (RTV) and depth buffer to the graphics pipeline.
This function ensures that the GPU is synchronized and ready to render into the target texture.
-------------------------------------------------------------------------------------------------------
--
Function Overview
-------------------------------------------------------------------------------------------------------
--
Define Texture Barrier for Layout Transition
```

```
TextureBarrier renderTargetBarrier{};
renderTargetBarrier.SyncBefore = SKTBD_SYNC_PIXEL_SHADING;
renderTargetBarrier.SyncAfter = SKTBD_SYNC_RENDER_TARGET;
renderTargetBarrier.AccessBefore = SKTBD_ACCESS_COMMON;
renderTargetBarrier.AccessAfter = SKTBD_ACCESS_RENDER_TARGET;
renderTargetBarrier.LayoutBefore = SKTBD_LAYOUT_COMMON;
renderTargetBarrier.LayoutAfter = SKTBD_LAYOUT_RENDER_TARGET;
renderTargetBarrier.Resource = OutputTexture.Get().get();


 Ensures the output texture is transitioned from a general/common state to a state suitable for render
target usage.
Why It’s Important:
 DirectX 12-style explicit graphics APIs require manual state transitions to ensure correct GPU access
behavior.

--------------------------------------------------------------------------------------------------------
--
Apply the Barrier

BarrierGroup grouptoRTV(&renderTargetBarrier);
RenderCommand::Barrier(&grouptoRTV, 1);


 Wraps the barrier into a group and submits it, forcing the transition on the GPU before the next draw
calls.

--------------------------------------------------------------------------------------------------------
--
Set the Render Target and Depth Buffer

RenderCommand::SetRenderTargets(OutputRTV.Get().get(), 1, GraphicsContext::GetDefaultDepthBuffer());

Binds the specified render target (RTV) and the default depth buffer as active targets for rendering.

--------------------------------------------------------------------------------------------------------
--
Conceptual Summary

This function:
Ensures the render target texture is in the right GPU state.
Binds it for rendering.
Enables subsequent draw calls to properly output to the desired texture.
It’s a critical utility for frame rendering, post-processing passes, and any render-to-texture workflows.
```

```
void PennyRender::MultiRenderPass(Texture renderTexture) { // not working


/*    renderScreenTest(renderTexture, OutputTextures_[0], OutputRTVs_[0], OutputSRVs_[0]);
    Texture tex = OutputSRVs_[0].Get();
    renderToScreen_end(OutputRTVs_[0], OutputSRVs_[0]);
```

```
    renderScreenTest(renderTexture, OutputTextures_1, OutputRTVs_1, OutputSRVs_1);*/
    renderScreenPass(renderTexture, OutputTextures_0, OutputRTVs_0, OutputSRVs_0); //
    Texture tex = OutputSRVs_0.Get();
    renderToScreen_end(OutputRTVs_0, OutputSRVs_0);
    renderScreen(tex);//always has to be the last one (as its "texture" -SRV- is what gets pasted through
to the preive screen)
    //renderToScreen_end(OutputRTVs_0, OutputSRVs_0);

}
```

```
PennyRender::MultiRenderPass()
-----------------------------------------------------------------------------------------
Purpose

The MultiRenderPass() function is intended to execute multiple rendering passes in sequence. It handles
rendering to offscreen textures, followed by final rendering to the screen. Although currently commented
out and non-functional, the general idea is to process different effects or steps in multiple passes,
updating the output after each pass, and finally displaying the result.

-----------------------------------------------------------------------------------------
Function Overview
-----------------------------------------------------------------------------------------
Rendering the First Pass to an Offscreen Texture

renderScreenPass(renderTexture, OutputTextures_0, OutputRTVs_0, OutputSRVs_0);


This line calls renderScreenPass() to perform the first render pass. The pass renders the scene to the
texture specified by OutputTextures_0, using the corresponding render target view (OutputRTVs_0) and
shader resource view (OutputSRVs_0). This is part of an offscreen rendering technique, for post-
processing effects or intermediate results.
-----------------------------------------------------------------------------------------
Getting the Texture for Further Processing

Texture tex = OutputSRVs_0.Get();

After completing the first render pass, this line retrieves the texture from OutputSRVs_0 (the shader
resource view) that contains the output of the first render pass. This texture will be used in subsequent
passes or for display.
-----------------------------------------------------------------------------------------
Finalizing the Render to Screen

renderToScreen_end(OutputRTVs_0, OutputSRVs_0);
```

This function call, renderToScreen_end(), finalizes the rendering for the current pass by updating the render target views and shader resource views. It effectively marks the end of the render pass and prepares the scene for displaying or processing further.

Rendering the Final Result to the Screen

```
renderScreen(tex);
```

This function call uses the texture (tex) from the previous render pass and renders it to the screen as a full-screen quad. This is usually the final step in the rendering pipeline when applying post-processing effects or displaying the final result.

Commented Out / Additional Render Passes

```
/*
renderScreenTest(renderTexture, OutputTextures_[0], OutputRTVs_[0], OutputSRVs_[0]);
Texture tex = OutputSRVs_[0].Get();
renderToScreen_end(OutputRTVs_[0], OutputSRVs_[0]);


renderScreenTest(renderTexture, OutputTextures_1, OutputRTVs_1, OutputSRVs_1);
*/
```

Purpose:
The commented-out code represents additional render passes that were planned but are currently not functional. These passes would render to different textures and views (OutputTextures_[0], OutputTextures_1, etc.), and the result would then be passed through the screen render process.
These lines suggest the possibility of handling multiple intermediate render passes (e.g., different visual effects or steps).

Output

Effect:
The function ensures that multiple render passes are performed and that the final rendered image is displayed on the screen. The function first renders to an offscreen texture, then processes the result, and finally renders the processed texture to the screen.
The code hints that the function was designed for handling multiple render passes, but it is currently incomplete or not working.

Conceptual Summary

The MultiRenderPass() function is designed to facilitate a multi-pass rendering pipeline where intermediate results are rendered offscreen, processed, and then displayed. Although not fully functional in its current form, this pattern is commonly used in rendering techniques like post-processing effects, shadow mapping, or deferred rendering, where each pass serves a specific purpose, such as applying a filter or calculating lighting.
The function concludes with rendering the final processed texture to the screen.

void PennyRender::renderToScreen_end(MultiResource<RenderTargetViewRef> OutputRTV, MultiResource<TextureSRVRef> OutputSRV) {
////next frame
++OutputSRV;
++OutputRTV;
}

PennyRender:: renderToScreen_end(MultiResource<RenderTargetViewRef> OutputRTV, MultiResource<TextureSRVRef> OutputSRV)

Purpose

The renderToScreen_end() function is used to finalize the process of rendering to the screen by updating the render target and shader resource view (SRV) for the next frame.

Function Overview

Incrementing the Render Target View (RTV)
++OutputRTV;

Purpose: This operation increments the Render Target View (RTV) to point to the next render target in the sequence.
Effect: This essentially prepares the system to render the next frame to a new target, moving forward in the sequence of render targets.

Incrementing the Shader Resource View (SRV)
++OutputSRV;

Purpose: Similar to the RTV, this operation increments the Shader Resource View (SRV) to the next texture in the sequence.
Effect: It ensures that the next texture resource (for example, a texture containing the result of the current frame's rendering) is used in subsequent shader operations, typically for post-processing or any other shader-based effects.

Conceptual Summary

The renderToScreen_end() function serves as a mechanism to increment the render target and shader resource view pointers, effectively preparing them for the next frame. This allows the system to move from one frame's output to the next frame's input seamlessly, ensuring continuous rendering.

```
void PennyRender::PennyBoardIMGUI_PreveiwScreen() {
    //-- barriers to tye/sync render target
    TextureBarrier renderTargetBarrier{};
    renderTargetBarrier.SyncBefore = SKTBD_SYNC_RENDER_TARGET;
```

```
    renderTargetBarrier.SyncAfter = SKTBD_SYNC_PIXEL_SHADING;
    renderTargetBarrier.AccessBefore = SKTBD_ACCESS_RENDER_TARGET;
    renderTargetBarrier.AccessAfter = SKTBD_ACCESS_COMMON;
    renderTargetBarrier.LayoutBefore = SKTBD_LAYOUT_RENDER_TARGET;
    renderTargetBarrier.LayoutAfter = SKTBD_LAYOUT_COMMON;
    renderTargetBarrier.Resource = OutputTexture.Get().get();
    renderTargetBarrier.SubresourceRange = TextureSubresourceRange();//


    BarrierGroup grouptoRTV(&renderTargetBarrier);
    RenderCommand::Barrier(&grouptoRTV, 1);
    //--


    ImGui::SeparatorText("Preveiw screen");


    ImGui::SliderInt("Screen size", &previewscreenSize, 1, 4, "Screen size / %d%");
    ImGui::SetItemTooltip("Preveiw screen size, Fullsize/ n");


    //preveiw screen
    ImGuiIO& io = ImGui::GetIO();
    ImTextureID my_tex_id = OutputSRV.Get()->GetImTextureID();//turning srv to a usable ImTextureID - for
preview screen
    //move to next frame
    renderToScreen_end(OutputRTV, OutputSRV);


    float my_tex_w = (float)1366 / previewscreenSize;
    float my_tex_h = (float)768 / previewscreenSize;
    {
        static bool use_text_color_for_tint = false;


        ImVec2 uv_min = ImVec2(0.0f, 0.0f);                 // Top-left
        ImVec2 uv_max = ImVec2(1.0f, 1.0f);                 // Lower-right
        ImVec4 tint_col = use_text_color_for_tint ? ImGui::GetStyleColorVec4(ImGuiCol_Text) :
ImVec4(1.0f, 1.0f, 1.0f, 1.0f);   // No tint
        ImVec4 border_col = ImGui::GetStyleColorVec4(ImGuiCol_Border);
        ImGui::Image(my_tex_id, ImVec2(my_tex_w, my_tex_h), uv_min, uv_max, tint_col, border_col);
        ImGui::SetItemTooltip("Preveiw screen;\nshow the effected in development before adding to the
proper screen/view");
    }
```

```
PennyRender::PennyBoardIMGUI_PreveiwScreen()
-------------------------------------------------------------------------------------------------------
-------
Purpose
```

This function prepares and displays a preview of the final rendered output texture inside an ImGui window. It ensures proper GPU synchronization (using a barrier) before showing the texture, and provides basic UI controls to scale the preview.

Barrier: Transition the Render Target to Common State

```
TextureBarrier renderTargetBarrier{};
renderTargetBarrier.SyncBefore = SKTBD_SYNC_RENDER_TARGET;
renderTargetBarrier.SyncAfter = SKTBD_SYNC_PIXEL_SHADING;
renderTargetBarrier.AccessBefore = SKTBD_ACCESS_RENDER_TARGET;
renderTargetBarrier.AccessAfter = SKTBD_ACCESS_COMMON;
renderTargetBarrier.LayoutBefore = SKTBD_LAYOUT_RENDER_TARGET;
renderTargetBarrier.LayoutAfter = SKTBD_LAYOUT_COMMON;
renderTargetBarrier.Resource = OutputTexture.Get().get();
renderTargetBarrier.SubresourceRange = TextureSubresourceRange();
BarrierGroup grouptoRTV(&renderTargetBarrier);
RenderCommand::Barrier(&grouptoRTV, 1);
```

Ensures the output texture is no longer being written to as a render target and is now in a state where it can be read as a shader resource for preview.

ImGui UI Controls and Preview Header

```
ImGui::SeparatorText("Preveiw screen");
ImGui::SliderInt("Screen size", &previewscreenSize, 1, 4, "Screen size / %d%");
ImGui::SetItemTooltip("Preveiw screen size, Fullsize/ n");
```

Provides a user interface to scale the preview size using a slider.
previewscreenSize controls how large the preview appears relative to full size (1 = full size, 4 = quarter size).

Get Texture ID and Advance Frame

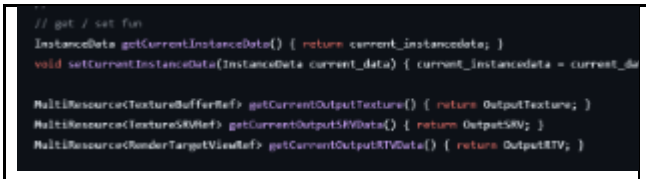
```
ImGuiIO& io = ImGui::GetIO();
ImTextureID my_tex_id = OutputSRV.Get()->GetImTextureID(); // Get a usable texture ID for ImGui
renderToScreen_end(OutputRTV, OutputSRV); // Advance resource pointer to next frame
```

Converts the Shader Resource View into a format (ImTextureID) usable by ImGui for image display.
Also advances to the next resource in the MultiResource pool to handle double/triple buffering.

Display the Texture as an ImGui Image

```
float my_tex_w = (float)1366 / previewscreenSize;
float my_tex_h = (float)768 / previewscreenSize;


ImVec2 uv_min = ImVec2(0.0f, 0.0f); // Top-left
ImVec2 uv_max = ImVec2(1.0f, 1.0f); // Bottom-right
ImVec4 tint_col = ImVec4(1.0f, 1.0f, 1.0f, 1.0f); // No tint
ImVec4 border_col = ImGui::GetStyleColorVec4(ImGuiCol_Border);


ImGui::Image(my_tex_id, ImVec2(my_tex_w, my_tex_h), uv_min, uv_max, tint_col, border_col);
ImGui::SetItemTooltip("Preveiw screen;\nshow the effected in development before adding to the proper
screen/view");

Displays the rendered texture inside the ImGui UI window as a resizable image.
Uses ImGui::Image() to show the texture and adds a helpful tooltip explaining the purpose of the preview
screen.

-------------------------------------------------------------------------------------------------------
-------
Conceptual Summary

Ensures GPU synchronization before accessing the texture as an SRV.
Provides a simple and flexible UI element in ImGui to preview the rendering output.
Helps developers test and debug visual effects in isolation before applying them to the final screen
output.
```

Misc Functions

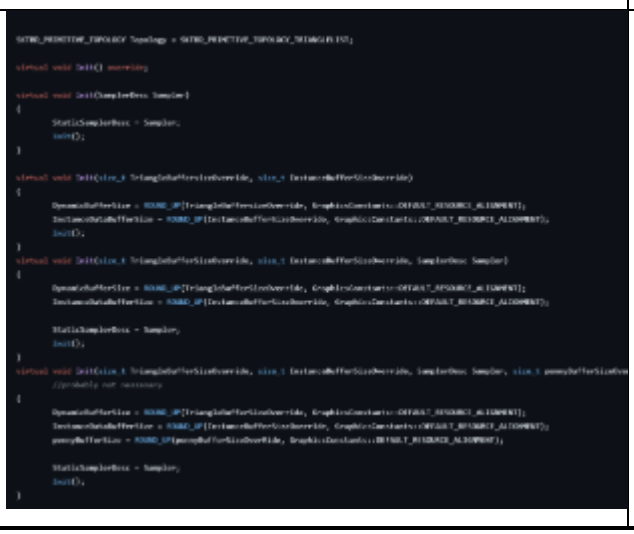


```
void save_SavedEffect(SavedEffect effect) {addEffectToList(savedEffect_List, eff
SavedEffect ConvertToSavedEffect(const char* SavedName , ShaderToggles EffectIn
{
        SavedEffect effect;

        effect.SavedName = SavedName;
        effect.EffectInUse = EffectInUse;

        effect.screen_buff = screen_buff;
        effect.scanline_buff = scanline_buff;
        effect.vignet_buff = vignet_buff;
        effect.posterize_buff = posterize_buff;
        effect.bloom_buff = bloom_buff;
        effect.bloom2_buff = bloom2_buff;
        effect.blurGauBuffer_buff = blurGauBuffer_buff;
        effect.chromaTrue_buff = chromaTrue_buff;
        effect.chromaStylized_buff = chromaStylized_buff;
        effect.colourGrad_buff = colourGrad_buff;
        effect.emboss_buff = emboss_buff;
        effect.glass_buff = glass_buff;
        effect.pixel_buff = pixel_buff;

        //addEffect(savedEffect_List, effect);
        return effect;

}
```

```
void ApplySavedEffect(SavedEffect effect) {

        SetCurrentState(effect.EffectInUse);

        m_ScreenBuffer = effect.screen_buff;
        m_ScanlineBuffer = effect.scanline_buff;
        m_VignettingBuffer = effect.vignet_buff;
        m_PosterizeBuffer = effect.posterize_buff;

        m_BloomBuffer = effect.bloom_buff;
        m_Bloom2Buffer = effect.bloom2_buff;
        m_BlurGauBuffer = effect.blurGauBuffer_buff;
        m_ChromaTrueBuffer = effect.chromaTrue_buff;

        m_ChromaStylizedBuffer = effect.chromaStylized_buf
        m_ColourGradBuffer = effect.colourGrad_buff;
        m_EmbossBuffer = effect.emboss_buff;
        m_GlassBuffer = effect.glass_buff;

        m_PixelBuffer = effect.pixel_buff;

}
```

```
void ResetAll() {

        SetCurrentState(DEFAULT_STATE_);

        m_ScreenBuffer = ScreenBuffer();
        m_ScanlineBuffer = ScanlineBuffer();
        m_VignettingBuffer = VignettingBuffer();
        m_PosterizeBuffer = PosterizeBuffer();

        m_BloomBuffer = BloomBuffer();
        m_Bloom2Buffer = Bloom2Buffer();
        m_BlurGauBuffer = BlurGauBuffer();
        m_ChromaTrueBuffer = ChromaTrueBuffer();

        m_ChromaStylizedBuffer = ChromaStylizedBuffer();
        m_ColourGradBuffer = ColourGradBuffer();
        m_EmbossBuffer = EmbossBuffer();
        m_GlassBuffer = GlassBuffer();

        m_PixelBuffer = PixelBuffer();

}
```

```
// - current pipeline
void SetCurrentState(ShaderToggles effect = DEFAULT_STAT
{
        CURRENT_STATE_ = effect;
        m_Pipeline_dirty = true;
}
```



```
void SetLights(std::vector<Light> Lights) { m_Lights = Lights; };

void AddLight(Light light) { m_Lights.push_back(light); }
Light& GetLight(uint32_t idx) { return m_Lights[idx]; }

//RendererConfiguration
ShaderInputLayoutRef GetRootSig() { return RootSig; };
ShaderToggles GetShaderToggles() const { return CURRENT_STATE_; };

//Debug normals
void SetDrawDebugNormals(bool DrawNormals) { m_VisualiseNormals = DrawNormals;

void SetAmbientLight(float3 light_colour) {
	m_Frame.AmbientLight = light_colour;
}
```

The Shader library

Structs

Quick sentence about the structs.

```
#ifndef STRUCTS_Penny2
#define STRUCTS_Penny2

//Para == parameters
```

```
struct ScreenPara //not working
{
    float2 screenSize_;
    int PushData_instanceNo;
    int padding;
};
```

```
struct scanlinePara
{
    float2 screenSize_;
    float LineThickness_;
    float DimmedFactor_;

    float transferPower_;
    float vertical_;
    int PushData_instanceNo;
    float padding;
};
```

```
struct vignettingPara //
{

    float2 screenSize_;
    float2 padding;

    float innerRadius;
    float outerRadius;
    float opacity;
    int PushData_instanceNo;
};
```

```
  struct posterizePara //
  {
      float3 step_areas;
      uint PushData_instanceNo;
  };
```

```
struct bloomPara //
{
    int PushData_instanceNo;
    int width;
    float angleSteps;
    float radiusSteps;

    float ampFactor;
    float3 padding; // align to 16 bytes
};
```

```
struct bloom2Para  //
{
    int PushData_instanceNo;
    float ampFactor;
    float threshold;
    float padding0;
};
```

```
struct blurGauPara  //
{
    float Luminance;
    float2 screenSize_;
    float padding0;

    float weight[8]; //last values ar padding

    float texscrMultiplier[8];//last values ar padding

    //float weight[5];
    //float padding1[3]; // padding to align next array

    //float texscrMultiplier[5];
    //float padding2[3]; // padding for alignment
};
```

```
struct ChromaTruePara  //
{
    int PushData_instanceNo;
    float2 screenSize_;
    float aberrationFactor;

    float aberrationFactor_x;
    float aberrationFactor_y;
    float2 padding; // padding for alignment
};
```

```
struct ChromaStylizedPara //might nc
                          //not wc
{
    int PushData_instanceNo;
    float3 padding0;

    float4 chromaticWeights1;
    float4 chromaticWeights2;
    float4 chromaticOffset1;
    float4 chromaticOffset2;
};
```

```
struct colourGradPara
{
    int PushData_instanceNo;
    float3 padding0;

    float3 startColor;
    float padding1;

    float3 endColor;
    float padding2;
};
```

```
struct EmbossPara  //
{
    int PushData_instanceNo;
    float2 screenSize_;
    float padding0;

    int greyScale;
    float width_;
    float height_;
    float padding1;
};
```

```
struct glassPara  //
{
    int PushData_instanceNo;
    float2 screenSize_;
    float2 panelSize;

    float2 padding; // for alignment
};
```

```
struct pixelPara  //
{
    int PushData_instanceNo;
    float2 screenSize_;
    float2 pixelSize;

    float2 padding; // for alignment
};

#endif
```

Shaders

Quick sentence about the library.

Scanline Pixel Shader

```
#include "../../CMP203/CMP203_Shaders/Structs.hlsli"
#include "PennyStructs_2.hlsli"

//sampler registers
SamplerState g_sampler : register(s0);
//buffers
ConstantBuffer<Constants> PushData : register(b0, space0);
StructuredBuffer<InstanceData> Instances : register(t0, space0);
ConstantBuffer<scanlinePara> Parameters : register(b2, space0);


float4 main(VertexOutput input) : SV_Target0
{
    //get values from buffer
    float LineThickness = Parameters.LineThickness_;
    float DimmedFactor = Parameters.DimmedFactor_;
    float transferPower = Parameters.transferPower_;
    float vertical = Parameters.vertical_;
    float2 screenSize = Parameters.screenSize_;

    float4 resultColor;

     // Access the texture with the instance data
    Texture2D texture0 = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureIDX];
    //get pixel colour from sampling texture
    float4 textureColor = texture0.Sample(g_sampler, input.uv);

    // Convert UVs to actual pixel positions
    int pixelX = input.uv.x * screenSize.x;
    int pixelY = input.uv.y * screenSize.y;


    // Determine if we're in a "band" area or a "gap" based on pixel location
    int xType = (pixelX / LineThickness) % 2;
    int yType = (pixelY / LineThickness) % 2;

    if (vertical != 0.5)//lines
    {
        if ((vertical && xType == 0) || (!vertical && yType == 0))
        {
            // Brighten the pixel (emphasize the line)
            resultColor = textureColor + (textureColor * (transferPower * (1 - DimmedFactor)));
        }
        else
        {
            // Dim the pixel
            resultColor = textureColor * DimmedFactor;
        }
    }
    else if (vertical == 0.5) // crosshatch / matrix
    {
        if (xType == 0 && yType == 0)
        {   // Brighten intersections more
            resultColor = textureColor + (textureColor * 3 * (transferPower * (1 - DimmedFactor)));
        }
        else
        {
            // Dims the left pixels
            resultColor = textureColor * DimmedFactor;
        }
    }

    return resultColor;
}
```

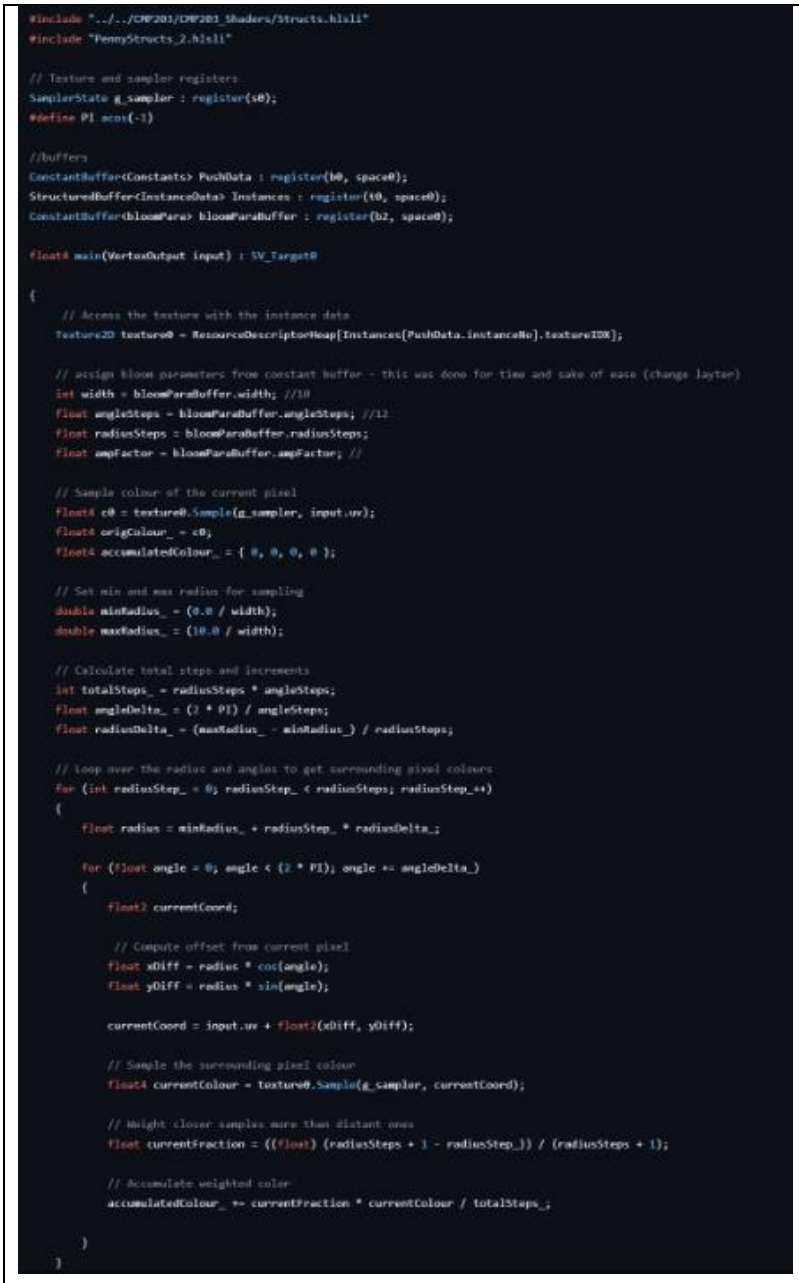
Vignetting Pixel Shader

```hlsl
#include "../../CMP203/CMP203_Shaders/Structs.hlsli"
#include "PennyStructs_2.hlsli"


//sampler registers
SamplerState g_sampler : register(s0);
#define PI acos(-1)

//buffers
ConstantBuffer<Constants> PushData : register(b0, space0);
StructuredBuffer<InstanceData> Instances : register(t0, space0);

ConstantBuffer<vignettingPara> Parameters : register(b2, space0); //also not working

float4 main(VertexOutput input) : SV_Target0
{
     // assign parameters from constant buffer
    float innerRadius = Parameters.innerRadius;
    float outerRadius = Parameters.outerRadius;
    float opacity = Parameters.opacity;

     // Access the texture with the instance data
    Texture2D texture0 = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureIDX];
    //gets pixel colour from sampling texture
    float4 textureColor = texture0.Sample(g_sampler, input.uv);

    // Adds vertical dimming effect
    float verticalDim = 0.5 + sin(input.uv.y * PI) * 0.9;

    // Convert UVs from [0,1] to [-1,1] coordinate space for radius calculation
    float xTrans = (input.uv.x * 2) - 1;
    float yTrans = 1 - (input.uv.y * 2);

    // calculates the radial distance from the center of the screen
    float radius = sqrt(pow(xTrans, 2) + pow(yTrans, 2));

    // Calculate how far past the innerRadius the pixel is
    float subtraction = max(0, radius - innerRadius) / (outerRadius - innerRadius);
    float factor = 1 - subtraction;

    // Apply the vignette effect based on the calculated factor
    float4 vignetColor = textureColor * factor;

    // Apply additional vertical fading
    vignetColor *= verticalDim;


    // Blend with original coloure based on opacity
    vignetColor *= opacity;
    textureColor *= 1 - opacity;


    // Combine both to get the final colour
    return textureColor + vignetColor;
}
```

posterize Pixel Shader

```hlsl
#include "../../CMP203/CMP203_Shaders/Structs.hlsli"
//#include "PennyStructs.hlsli"
#include "PennyStructs_2.hlsli"


// sampler registers
SamplerState g_sampler : register(s0);
#define PI acos(-1)

ConstantBuffer<Constants> PushData : register(b0, space0);

StructuredBuffer<InstanceData> Instances : register(t0, space0);
ConstantBuffer<posterizePara> Parameters : register(b2, space0); //

float4 main(VertexOutput input) : SV_Target0
{
     // Access the texture with the instance data
    Texture2D texture0 = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureTEX];

    //sample to get pixle colour
    float4 textureColour = texture0.Sample(g_sampler, input.uv);

    //Parameters controlling how many steps each channel is broken into -
    // e.g., (4, 4, 4) reduces each channel to 4 values/colours
    float3 step_areas = Parameters.step_areas;

    // Scale the colours by number of steps
    textureColour.r = textureColour.r * step_areas.r;
    textureColour.g = textureColour.g * step_areas.g;
    textureColour.b = textureColour.b * step_areas.b;

     // round down the values to lock them into steps, make sure the colour is opaque by having the last value be 1
    float4 result = float4((floor(textureColour.r) + 0.5) / step_areas.r, (floor(textureColour.g) + 0.5) / step_areas.g, (floor(textureColour.b) + 0.5) / step_areas.b, 1);

    return result;

}
```

bloom Pixel Shader

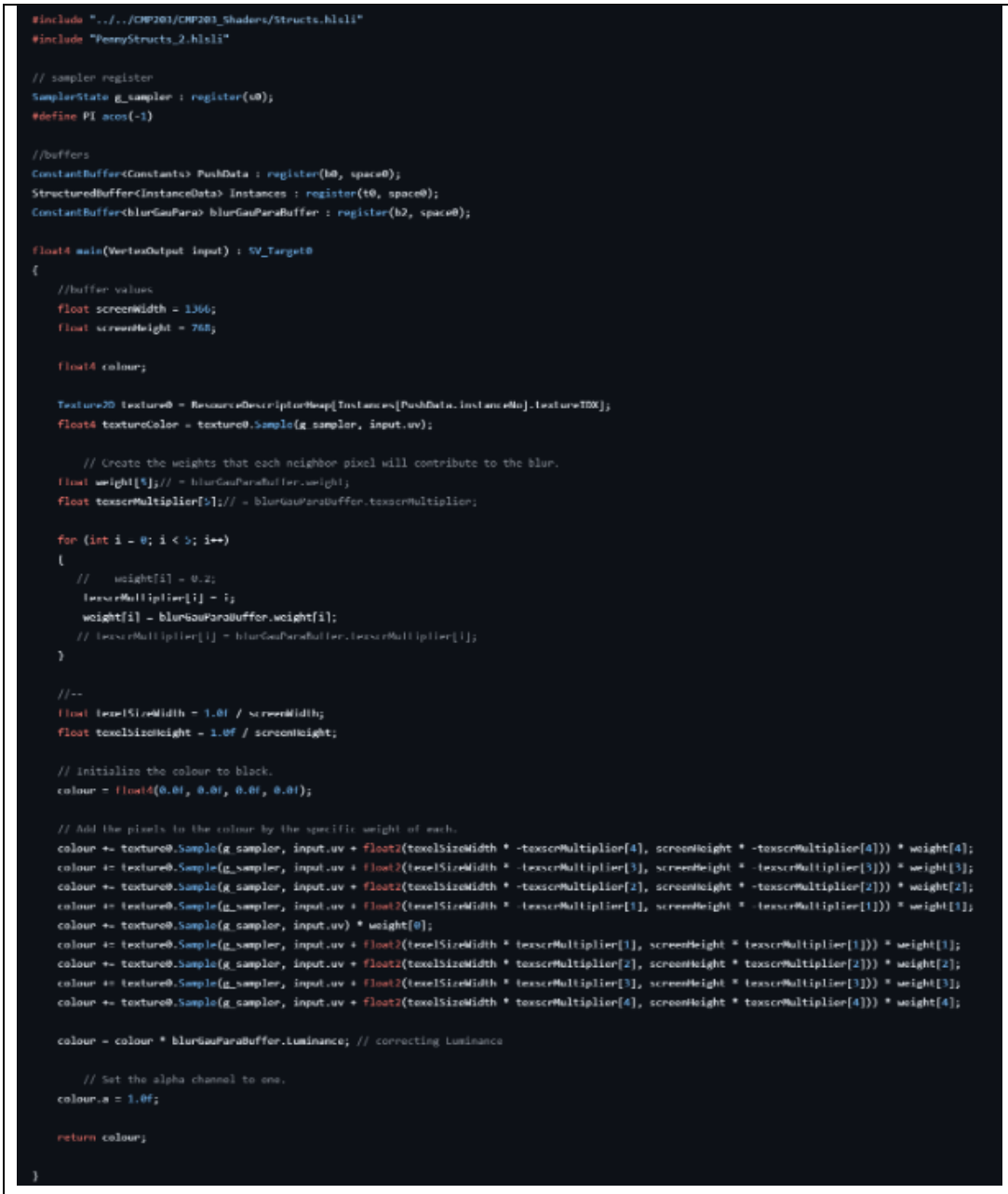


```hlsl
#include "../../CMP201/CMP201_Shaders/Structs.hlsli"
#include "PennyStructs_2.hlsli"

// Texture and sampler registers
SamplerState g_sampler : register(s0);
#define PI acos(-1)

//buffers
ConstantBuffer<Constants> PushData : register(b0, space0);
StructuredBuffer<InstanceData> Instances : register(t0, space0);
ConstantBuffer<bloomPara> bloomParaBuffer : register(b2, space0);

float4 main(VertexOutput input) : SV_Target0

{
    // Access the texture with the instance data
    Texture2D texture0 = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureIDX];

    // assign bloom parameters from constant buffer - this was done for time and sake of ease (change layter)
    int width = bloomParaBuffer.width; //10
    float angleSteps = bloomParaBuffer.angleSteps; //12
    float radiusSteps = bloomParaBuffer.radiusSteps;
    float ampFactor = bloomParaBuffer.ampFactor; //

    // Sample colour of the current pixel
    float4 c0 = texture0.Sample(g_sampler, input.uv);
    float4 origColour_ = c0;
    float4 accumulatedColour_ = { 0, 0, 0, 0 };

    // Set min and max radius for sampling
    double minRadius_ = (0.0 / width);
    double maxRadius_ = (10.0 / width);

    // Calculate total steps and increments
    int totalSteps_ = radiusSteps * angleSteps;
    float angleDelta_ = (2 * PI) / angleSteps;
    float radiusDelta_ = (maxRadius_ - minRadius_) / radiusSteps;

    // Loop over the radius and angles to get surrounding pixel colours
    for (int radiusStep_ = 0; radiusStep_ < radiusSteps; radiusStep_++)
    {
        float radius = minRadius_ + radiusStep_ * radiusDelta_;

        for (float angle = 0; angle < (2 * PI); angle += angleDelta_)
        {
            float2 currentCoord;

             // Compute offset from current pixel
            float xDiff = radius * cos(angle);
            float yDiff = radius * sin(angle);

            currentCoord = input.uv + float2(xDiff, yDiff);

            // Sample the surrounding pixel colour
            float4 currentColour = texture0.Sample(g_sampler, currentCoord);

            // Weight closer samples more than distant ones
            float currentFraction = ((float) (radiusSteps + 1 - radiusStep_)) / (radiusSteps + 1);

            // Accumulate weighted color
            accumulatedColour_ += currentFraction * currentColour / totalSteps_;

        }
    }
```

Bloom 2 Pixel Shader

```hlsl
#include "../../CMP203/CMP203_Shaders/Structs.hlsli"
#include "PennyStructs_2.hlsli"

// Texture and sampler registers
SamplerState g_sampler : register(s0);
#define PI acos(-1)

//buffers
ConstantBuffer<Constants> PushData : register(b0, space0);
StructuredBuffer<InstanceData> Instances : register(t0, space0);
ConstantBuffer<bloom2Para> bloom2ParaBuffer : register(b2, space0);

float4 main(VertexOutput input) : SV_Target0
{
    // Access the texture with the instance data
    Texture2D texture0 = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureIDX];


    // assign bloom parameters from constant buffer
    float ampFactor = bloom2ParaBuffer.ampFactor; //
    float threshold = bloom2ParaBuffer.threshold; //

    float4 bloomColour = texture0.Sample(g_sampler, input.uv); //bloom colour

    // if pixel colour is lower than or == threshhold
    if ((bloomColour.x <= threshold) || (bloomColour.y <= threshold) || (bloomColour.z <= threshold))
    {
        bloomColour = float4(0.0f, 0.0f, 0.0f, 0.0f);//set current pixle colour to 0
    }

    // Sample orignal pixle colour
    float4 outputPixel = texture0.Sample(g_sampler, input.uv);

    //add bloom (* by amp factor to ajust brightness) to the orignal texture/pixel colour
    outputPixel += (bloomColour * ampFactor);

    //return the new colour
    return outputPixel;

}
```

Blur Gau Para



```hlsl
#include "../../CMP203/CMP203_Shaders/Structs.hlsli"
#include "PennyStructs_2.hlsli"

// sampler register
SamplerState g_sampler : register(s0);
#define PI acos(-1)

//buffers
ConstantBuffer<Constants> PushData : register(b0, space0);
StructuredBuffer<InstanceData> Instances : register(t0, space0);
ConstantBuffer<blurGauPara> blurGauParaBuffer : register(b2, space0);

float4 main(VertexOutput input) : SV_Target0
{
    //buffer values
    float screenWidth = 1366;
    float screenHeight = 768;

    float4 colour;

    Texture2D texture0 = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureTEX];
    float4 textureColor = texture0.Sample(g_sampler, input.uv);

        // Create the weights that each neighbor pixel will contribute to the blur.
    float weight[5];// = blurGauParaBuffer.weight;
    float texscrMultiplier[5];// = blurGauParaBuffer.texscrMultiplier;

    for (int i = 0; i < 5; i++)
    {
       //    weight[i] = 0.2;
        texscrMultiplier[i] = i;
        weight[i] = blurGauParaBuffer.weight[i];
       // texscrMultiplier[i] = blurGauParaBuffer.texscrMultiplier[i];
    }

    //--
    float texelSizeWidth = 1.0f / screenWidth;
    float texelSizeHeight = 1.0f / screenHeight;

    // Initialize the colour to black.
    colour = float4(0.0f, 0.0f, 0.0f, 0.0f);

    // Add the pixels to the colour by the specific weight of each.
    colour += texture0.Sample(g_sampler, input.uv + float2(texelSizeWidth * -texscrMultiplier[4], screenHeight * -texscrMultiplier[4])) * weight[4];
    colour += texture0.Sample(g_sampler, input.uv + float2(texelSizeWidth * -texscrMultiplier[3], screenHeight * -texscrMultiplier[3])) * weight[3];
    colour += texture0.Sample(g_sampler, input.uv + float2(texelSizeWidth * -texscrMultiplier[2], screenHeight * -texscrMultiplier[2])) * weight[2];
    colour += texture0.Sample(g_sampler, input.uv + float2(texelSizeWidth * -texscrMultiplier[1], screenHeight * -texscrMultiplier[1])) * weight[1];
    colour += texture0.Sample(g_sampler, input.uv) * weight[0];
    colour += texture0.Sample(g_sampler, input.uv + float2(texelSizeWidth * texscrMultiplier[1], screenHeight * texscrMultiplier[1])) * weight[1];
    colour += texture0.Sample(g_sampler, input.uv + float2(texelSizeWidth * texscrMultiplier[2], screenHeight * texscrMultiplier[2])) * weight[2];
    colour += texture0.Sample(g_sampler, input.uv + float2(texelSizeWidth * texscrMultiplier[3], screenHeight * texscrMultiplier[3])) * weight[3];
    colour += texture0.Sample(g_sampler, input.uv + float2(texelSizeWidth * texscrMultiplier[4], screenHeight * texscrMultiplier[4])) * weight[4];

    colour = colour * blurGauParaBuffer.Luminance; // correcting Luminance

        // Set the alpha channel to one.
    colour.a = 1.0f;

    return colour;

}
```

Chroma True Pixel Shader

```hlsl
#include "../../CMP203/CMP203_Shaders/Structs.hlsli"
#include "PennyStructs_2.hlsli"

//sampler registers
SamplerState g_sampler : register(s0);
#define PI acos(-1)


//buffers
ConstantBuffer<Constants> PushData : register(b0, space0);
StructuredBuffer<InstanceData> Instances : register(t0, space0);
ConstantBuffer<ChromaTruePara> ChromaTrueParaBuffer : register(b2, space0);


float4 main(VertexOutput input) : SV_Target0
{
  // Access the texture with the instance data
    Texture2D texture0 = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureIDX];

    //buffer values assigned
    float aberrationFactor = ChromaTrueParaBuffer.aberrationFactor;
    float aberrationFactor_x = ChromaTrueParaBuffer.aberrationFactor_x;
    float aberrationFactor_y = ChromaTrueParaBuffer.aberrationFactor_y;

    // calculate distances from the reference point
    float distX = input.uv.x - aberrationFactor_x;
    float distY = -input.uv.y + aberrationFactor_y;

    //init all rgb colour channles first
    float2 rColour, gColour, bColour;

    // Red channel gets pushed outward
    rColour.x = input.uv.x + aberrationFactor * distX;
    rColour.y = input.uv.y + aberrationFactor * distY;

    // Green channel stays centered (acts like an anchor)
    gColour.x = input.uv.x;
    gColour.y = input.uv.y;

    // Blue channel gets pulled inward (opposite to red)
    bColour.x = input.uv.x - aberrationFactor * distX;
    bColour.y = input.uv.y - aberrationFactor * distY;

    // Sample color from each adjusted channel
    float4 rTex = texture0.Sample(g_sampler, rColour);
    float4 gTex = texture0.Sample(g_sampler, gColour);
    float4 bTex = texture0.Sample(g_sampler, bColour);

    // add up final pixel color with shifted RGB values and make sure opacity is 1(full)
    float4 result = { rTex[0], gTex[1], bTex[2], 1 };

    return result;

}
```

Chroma Stylized Pixel Shader

```hlsl
#include "../../CMP203/CMP203_Shaders/Structs.hlsli"
#include "PennyStructs_2.hlsli"


// sampler registers
SamplerState g_sampler : register(s0);
#define PI acos(-1)

//buffers
ConstantBuffer<Constants> PushData : register(b0, space0);
StructuredBuffer<InstanceData> Instances : register(t0, space0);
ConstantBuffer<ChromaStylizedPara> ChromaStylizedParaBuffer : register(b2, space0);


float4 main(VertexOutput input) : SV_Target0
{

     // Access texture with instance data
    Texture2D texture0 = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureIDX];

    //assign the buffer values from constBuffer
    float4 chromaticWeights1 = ChromaStylizedParaBuffer.chromaticWeights1;
    float4 chromaticWeights2 = ChromaStylizedParaBuffer.chromaticWeights2;
    float4 chromaticOffset1 = ChromaStylizedParaBuffer.chromaticOffset1;
    float4 chromaticOffset2 = ChromaStylizedParaBuffer.chromaticOffset2;

    //set all colours initially to 0
    float4 vColor_chroma1 = float4(0, 0, 0, 0);
    float4 vColor_chroma2 = float4(0, 0, 0, 0);
    float4 vColor_chroma = float4(0, 0, 0, 0);


    //1
    for (int i = 0; i < chromaticOffset1.z; i++)
    {
        vColor_chroma1 += chromaticWeights1 * texture0.Sample(g_sampler, input.uv + chromaticOffset1.xy);
    }
    vColor_chroma1 = vColor_chroma1 / chromaticOffset1.z;

    //2
    for (int i = 0; i < chromaticOffset2.z; i++)
    {
        vColor_chroma2 += chromaticWeights2 * texture0.Sample(g_sampler, input.uv + chromaticOffset2.xy);
    }
    vColor_chroma2 = vColor_chroma2 / chromaticOffset2.z;

    //combine all the pixel colours
    vColor_chroma = vColor_chroma2 + vColor_chroma1;

    return vColor_chroma; //NUM_SAMPLES

}
```

Colour Grad Pixel Shader

```hlsl
#include "../../CMP203/CMP203_Shaders/Structs.hlsli"
#include "PennyStructs_2.hlsli"

//sampler registers
SamplerState g_sampler : register(s0);
#define PI acos(-1)

//buffers
ConstantBuffer<Constants> PushData : register(b0, space0);
StructuredBuffer<InstanceData> Instances : register(t0, space0);

ConstantBuffer<colourGradPara> colourGradParaBuffer : register(b2, space0);

float4 main(VertexOutput input) : SV_Target0
{
    //bufferValues
    float3 startColor = colourGradParaBuffer.startColor;
    float3 endColor = colourGradParaBuffer.endColor;

     // Access the texture with the instance data
    Texture2D texture0 = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureIDX];
    //get pixel colour using sampler
    float4 textureColour = texture0.Sample(g_sampler, input.uv);

    //lumanince is calcluate
    float lum = (textureColour.r + textureColour.g + textureColour.b) / 3.0;
    float lumComp = 1.0 - lum;

        // Create colour based on luminance and the colour range
    float4 modColour;
    modColour.r = lum * startColor.r + lumComp * endColor.r;
    modColour.g = lum * startColor.g + lumComp * endColor.g;
    modColour.b = lum * startColor.b + lumComp * endColor.b;
    modColour.a = 1;

        // Multiply the original colour by the modified color
    float4 result = textureColour * modColour;

        //-- luminance restoration
    float resultLum = (result.r + result.g + result.b) / 3.0;
    result *= lum / resultLum;

    return result;


}
```

Emboss Pixel Shader

```hlsl
#include "../../CMP203/CMP203_Shaders/Structs.hlsli"
#include "PennyStructs_2.hlsli"


//sampler registers
SamplerState g_sampler : register(s0);
#define PI acos(-1)

//buffers
ConstantBuffer<Constants> PushData : register(b0, space0);
StructuredBuffer<InstanceData> Instances : register(t0, space0);
ConstantBuffer<EmbossPara> EmbossParaBuffer : register(b2, space0);

float4 main(VertexOutput input) : SV_Target0
{

    //float2 screenSize = (1366, 768);
    float2 screenSize = EmbossParaBuffer.screenSize_;

    bool greyScale = EmbossParaBuffer.greyScale;//some times this could potentially produce a really coo

     // Access the texture with the instance data
    Texture2D texture0 = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureIDX];

    // assign values from constant buffer
    float width_ = EmbossParaBuffer.width_;
    float height_ = EmbossParaBuffer.height_;

    // Calculate one-pixel step sizes in UV space
    float dx = 1 / width_;
    float dy = 1 / height_;

    // Sample neighboring pixels in a cross-diagonal pattern
    float4 textureColour_0 = texture0.Sample(g_sampler, input.uv + float2(-dx, -dy));
    float4 textureColour_1 = texture0.Sample(g_sampler, input.uv + float2(0, -dy));
    float4 textureColour_2 = texture0.Sample(g_sampler, input.uv + float2(-dx, 0));
    float4 textureColour_3 = texture0.Sample(g_sampler, input.uv + float2(dx, 0));
    float4 textureColour_4 = texture0.Sample(g_sampler, input.uv + float2(0, dy));
    float4 textureColour_5 = texture0.Sample(g_sampler, input.uv + float2(dx, dy));

    // Combine sampled values to emphasize edges and depth (emboss calculation)
    float4 textureColor = (-textureColour_0 - textureColour_1 - textureColour_2 + textureColour_3 + text

    //If grayscale is enabled, convert the result to greyscale
    if (greyScale == true)
    { // Average the RGB channels and add 0.5 to adjust the brightness
       textureColor = (textureColor.r + textureColor.g + textureColor.b) / 3 + 0.5; // grey scale
    }

    return textureColor;

}
```

glass Pixel Shader

```hlsl
#include "../../CMP203/CMP203_Shaders/Structs.hlsli"
#include "PennyStructs_2.hlsli"


//sampler register
SamplerState g_sampler : register(s0);
#define PI acos(-1)


//buffers
ConstantBuffer<Constants> PushData : register(b0, space0);
StructuredBuffer<InstanceData> Instances : register(t0, space0);

ConstantBuffer<glassPara> glassParaBuffer : register(b2, space0);

float4 main(VertexOutput input) : SV_Target0
{
    //bufferValues
    glassParaBuffer.screenSize_;

    float width = 1366;//screen size
    float height = 768;//screen size

    // buffer values assigned
    // -- Panel size controls frequency of the distortion pattern
    float xSize = glassParaBuffer.panelSize.x;
    float ySize = glassParaBuffer.panelSize.y;

    // Access the texture with the instance data
    Texture2D texture0 = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureIDX];

    float2 tex2;

    // Determines position within the panel grid
    int xSubpixel = (input.uv.x * width) % xSize;
    int ySubpixel = (input.uv.y * height) % ySize;

    //calculates a small offset based on position within the panel grid
    float2 offsets = { 0.8 * xSubpixel / width, 0.8 * ySubpixel / height };

     // Apply the offset to the input UVs to simulate refraction through glass
    input.uv += offsets;

    // additional offset for sampling
    int offset = 2;
    tex2.x = input.uv.x + (offset / width);
    tex2.y = input.uv.y + (offset / height);

    //sample colur with offset
    float4 textureColour_1 = texture0.Sample(g_sampler, tex2);

    return textureColour_1;


}
```

Pixelate Pixel Shader

```hlsl
#include "../../CMP203/CMP203_Shaders/Structs.hlsli"
#include "PennyStructs_2.hlsli"

// Texture and sampler registers
SamplerState g_sampler : register(s0);
#define PI acos(-1)

//buffers
ConstantBuffer<Constants> PushData : register(b0, space0);
StructuredBuffer<InstanceData> Instances : register(t0, space0);

ConstantBuffer<pixelPara> pixelParaBuffer : register(b2, space0); //


float4 main(VertexOutput input) : SV_Target0
{
   // Size of each virtual "pixel"
    float2 pixelSize = pixelParaBuffer.pixelSize;

    float2 tex1;

    // assigning pixel size with values from buffers
    float width_ = pixelSize.x;
    float height_ = pixelSize.y;

    // Convert UV coordinates to screen-space "pixel" positions
    int pixelX = input.uv.x * width_;
    int pixelY = input.uv.y * height_;

    //Align UVs to the top-left corner of the current pixel block
    tex1.x = ((pixelX / pixelSize) * pixelSize) / width_;
    tex1.y = ((pixelY / pixelSize) * pixelSize) / height_;

     // Access the texture with the instance data
    Texture2D texture0 = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureIDX];

    // Use the UVs to get a "blocky" pixelated color
    float4 textureColor = texture0.Sample(g_sampler, tex1);

    return textureColor;

}
```

Default shader

```
#include "../../CMP203/CMP203_Shaders/Structs.hlsli"
#include "PennyStructs_2.hlsli"

// Texture and sampler registers
SamplerState g_sampler : register(s0);
#define PI acos(-1)

ConstantBuffer<Constants> PushData : register(b0, space0);
StructuredBuffer<InstanceData> Instances : register(t0, space0);

cbuffer FRAMEDATA : register(b1, space0)
{
    Frame FrameData;
}

ConstantBuffer<ScreenPara> para : register(b2, space0);


float4 main(VertexOutput input) : SV_Target0
{
    //this just passes through the texture colour with no changes - basically regular texture pixle shader

    Texture2D texture = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureIDX];

    float4 c0 = texture.Sample(g_sampler, input.uv);

    return c0;

}
```

Vertex shader

This was originally from the 203 extenders