Bachelor’s thesis

Information and Communications Technology

2023

Iiro Lehtiö

Utilizing shaders in 2D games

TURKU AMK
TURKU UNIVERSITY OF
APPLIED SCIENCES

Bachelor's | Abstract

Turku University of Applied Sciences

Information and Communications Technology

2023 | 40

Iiro Lehtiö

Utilizing shaders in 2D games

This thesis serves as an introduction to the possibilities of using shaders in 2D game development. Shaders are compact programs that alter how game graphics are drawn on screen. The thesis familiarises the reader with the elements of visual design in video games. The work conducted for the thesis was commissioned by Juicy.games game company.

Many small game development teams avoid using shaders as implementing them requires a combination of artistic and technical expertise. This is compounded by the fact that some projects may not require the efficient implementations that shaders provide. Shaders can provide cost-saving implementations and offer infinite solutions to graphical paradigms. The purpose of this thesis was to explore how shaders can be utilized effectively.

The thesis highlights different methods of employing shaders in the commissioner's game project. The result was two systems that incorporate shaders to liven up the game world and a tool for a graphic designer to evaluate different shaders in real time inside a game The implementations helped to reach a better understanding of the apprehension beginning developers have towards shaders. As the commissioner's game's development continues, the work carried out in this thesis serves as a firm base for further graphical implementations.

Keywords:

2D, shader, graphic design, video game

Opinnäytetyö (AMK) | Tiivistelmä

Turun ammattikorkeakoulu

Tietotekniikka

2023 | 40

Iiro Lehtiö

Varjostimien hyödyntäminen 2D-peleissä

Tämä opinnäytetyö selvittää varjostimien käyttömahdollisuuksia 2D-pelien kehityksessä. Varjostimet ovat ohjelmia, jotka muuttavat peligrafiikan piirtämistapaa. Opinnäytetyössä esitellään visuaalisen suunnittelun elementtejä videopelien kehityksessä. Työ tehtiin Juicy.games -peliyhtiön toimeksiantona.

Monet pienet pelikehittäjät välttelevät ja epäröivät varjostimien käyttöä, koska niiden käyttö vaatii taiteellista ja teknistä asiantuntemusta. Kaikissa projekteissa ei vaadita varjostimien mahdollistamia tehokkaita graafisia toteutustapoja, minkä seurauksena niiden käyttöä voidaan helposti vältellä. Varjostimilla voidaan tehdä kustannustehokkaita toteutuksia ja ne mahdollistavat loputtomia ratkaisuja erilaisiin graafisiin malleihin. Opinnäytetyö tutkii tapoja, joilla varjostimet voivat vähentää tiimien työkuormaa hyvin suunnitellussa projektissa.

Opinnäytetyö esittelee erilaisia tapoja käyttää varjostimia toimeksiantajan peliprojektissa. Varjostimien toteuttamisen tavoitteena projektissa oli tehostaa pelaajan immersiota. Tuloksena oli kaksi järjestelmää, jotka käyttävät varjostimia pelimaailman elävöittämiseksi, sekä työkalu, jonka avulla graafinen suunnittelija voi testata erilaisia varjostimia reaaliajassa. Toteutusten kautta saavutettiin parempi ymmärrys epäröinnistä, jota varjostimet aiheuttavat aloitteleville pelinkehittäjille. Toimeksiantajan pelin kehityksen jatkuessa opinnäytetyössä toteutettu työ toimii vankkana pohjana tuleville graafisille toteutuksille.

Asiasanat:

2D, varjostin, graafinen suunnittelu, videopelit

Contents

Figures

List of abbreviations

2D	Two dimensional, something that has no depth.
3D	Three dimensional.
Alpha	Transparency value of a colour.
Buffer	region in memory.
CPU	Processor, optimized for complicated calculations.
GPU	Graphical processing unit, excels in running shaders and other task relating to drawing.
IDE	integrated development environment.
Post processing	process of applying effects to an image before it is drawn on screen.
Shader	Compact program that manipulates the drawing of images on screen in real-time
Sprite	2D image
UI	User interface.

1 Introduction

2D games are a popular style of video games being produced in 2023, with roughly half of all games released on Steam, largest PC gaming marketplace, being 2D games. (SteamDB 2023) The genre is seen as easy to learn for beginning developers as the games in the genre are perceived by many to be less complex and requiring simpler graphical assets than 3D games. 2D games often require much less computing power to provide a good-looking game. The power of modern gaming systems enables developers to ignore most limits of performance and focus more on other areas of game design. Shaders are one of the most powerful components of modern game graphics used in both 2D and 3D games.

Shaders are graphics subprograms that can alter how every pixel gets drawn on the screen. Every pixel on the screen most likely passes through at least one shader in a modern video game. Shaders can output any kind of visual effect or image that can be mathematically defined. These infinite solutions require a level of programming expertise not found in many development teams. This can lead to teams implementing graphical solutions that mimic the results provided by shaders, which can be inefficient or even harmful in the long run. As shaders are very efficient and transferable between projects they should be considered for most graphical implementations.

The thesis first explores various graphical areas where shaders can provide a desired result. Chapter 2 discusses graphic design in video games, while Chapter 3 introduces the core elements of visual design. By understanding the basic principles of graphic and game design a better understanding of possible shader applications can be reached. After this, in Chapter 4, the thesis describes the graphics pipeline and the limiting factors of shaders. The development environment is introduced, and the impact shaders can have in game production discussed in Chapters 5 and 6. The implementation part showcases four different methods of shader implementations in Chapter 7.

It is important to note that most game engines are black boxes, and it is uncertain how they use shaders internally. The GameMaker game engine used in the implementation, for instance, offers many functions and built-in variables that handle how sprites are drawn on screen. Many of these methods most likely use shaders in their operation. This thesis regards shaders as their own programs, that are incorporated into a game. This distinction is made because the operation of these shaders can be affected more easily, and they are more transferable between projects contributing to the topic of development resource management.

2 Graphic design in games

Graphic design is the practice of planning visual products so that they effectively convey their message in an effective way. Well-designed video games are frictionless to play and evoke strong emotions through visual associations. In a product such as a video game, graphic design should also consider the available resources that the team producing the product has access to. Graphics production can take large amounts of work hours and device capability running the game must be factored in.

Video games are an inherently visual medium. Graphics are often the most important method of feedback a player receives during gameplay. Usually audio and haptic feedback are also used to support graphics. Combined, these elements represent certain concepts that the player recognizes and guide the player in how they interact with the game. Colors, shapes and movement all have their own associations in the human brain. Knowing how to properly define and combine these factors is the essential skill to have for a graphic designer. Shaders are an invaluable tool to realize good graphics design as shaders can manipulate any of the graphics drawn on the screen in real time.

2.1 Presenting information in games

Areas requiring graphic design can be divided into three broad categories: interactable elements, passive elements and the UI (user interface). Interactable elements are objects in the game world that the player can interact with in some way. Examples of such objects are player character, collectable resources and important locations. In games that do not have a visually discernible game world the UI may belong to this category. Passive elements could include terrain, backgrounds and non-interactable furniture. A UI consists of elements that the player uses to gain information and control the gameplay, the difference to the interactable category being that the UI is often anchored to the game screen.

Interactive objects form the core of the game as the game revolves around the player interacting with these objects. The objective of the graphical designer is to design the interactable objects in a manner that they are easily recognized from each other and the passive elements of the game world. Shaders excel in altering already existing graphical assets in many ways and, this makes them extremely potent tools for highlighting important aspects of the game. Common examples of this technique are drawing an outline around an object or altering its coloration when it is interactable. The real time nature of shaders also makes them ideal for the task as they can be implemented to react to the current situation inside the game and can be applied to a group of objects requiring highlight.

Accessibility can be a large part of the success of a video game. Reducing barriers of access is a major task of the graphic designer. Players with certain visual impairments may be unable to play a game that does not take them into consideration. Certain colour combinations can be hard or impossible to discern for a person with colour blindness. Texts in game may be difficult to read for some players. Shaders could alleviate these issues either in the design process or in a completed product. The game could have an option to enable a colour blindness mode that applies a shader to the entire screen that alters the colours to be more recognisable. This shader could even have modes to different kinds of colour blindness if required. Such a shader could be used by the designer to simulate blurry vision or different ways of perceiving colours, to detect design areas that may raise concern.

2.2 Game experience

Being a visual medium, graphics matter a great deal in video games (Schell 2019, 54). Marketing vice the graphics attract players to the game, but the graphics quality must match the experience. (Zukowski 2023) Game graphics do not only serve to deliver gameplay information to the player. As most games provide their feedback primarily through visuals, graphics are a major method of immersing the player in the game world. Immersion helps the player become more involved with the game and thus increases enjoyment of the game. Games, and other visual mediums, have certain genres that often have similar visual styles that help the players immerse into and understand the game.

The major area where shaders are used in video games is called post processing. After a game’s camera captures the game world and its objects onto a frame, post processing effects can be applied before the frame is drawn onto a screen. A horror game could use stock game assets with bright colors and crisp outlines. The games post processing shader could alter all the game's colors to be muted and the drawing of lights blurry and overflowing providing a hazy result. This would provide a unified look throughout the game without altering every game asset manually. The post processing shader could be altered based on feedback to provide a look that is more in line with design goals and player expectations. (Figure 1)

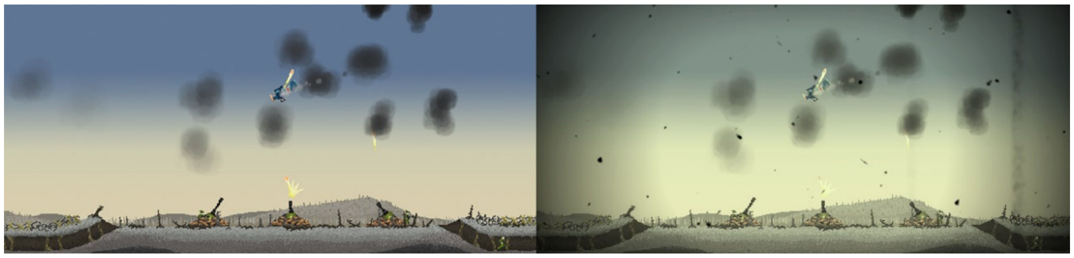


Figure 1. Multiple post-processing filters applied to Triplane Furball scene to create a vintage film feel.

2.3 2D games

Classifying what a 2D game is hard as many gaming related classifications are subjective and tend to vary based on the context. For instance, a game might be classified as a 2D game based either on the gameplay or the graphics. Generally, a 2D game has at least one of these characteristics: the gameplay occurs in two-dimensional space, the game’s graphical assets are 2D images, the perspective is fixed and does not change.

2D games have a set of common graphical design features. As the 2D perspective can be seen as unrealistic, the need for these games to approximate reality is reduced compared to 3D games. With lowered player expectations for realism, physics and lighting can be portrayed in an exaggerated manner that better serves the gameplay. This also empowers more stylized art styles. 2D games utilize many graphical techniques to immerse the player into the game world. Parallax scroll is a technique unique to 2D games, where backgrounds are moved as the player moves on the screen or the screen moves. This effect creates a sense of depth when the backgrounds in the horizon are moved at different speeds based on their distance from the player in the foreground.

Shaders can be utilized to power many of the techniques creating immersion. Color grading, lighting and full screen effects like distortions are all popular means to ground the player into the world. A shader could also make some game objects seem to be more part of the world, for example a shader could draw rippling waves around an island, an effect that is hard to achieve on a complex shape with other means.

3 Graphic design

3.1 Colors

Color usage is a major part of the graphical design process. Color theory is a concept that details how colors interact with each other and how humans associate certain concepts with colors. For example, in most cultures green has positive and red has negative connotations. Colors can have many properties, red and green are complementary or opposing colors, so they have a high contrast which makes the two clear to notice from each other. Intensity, brightness, warmth and transparency are some of the major properties that colors may have. (Meriläinen 2014, 12).

In computer graphics color is often represented as a combination of three or four components. Red, green, blue or red, green, blue and alpha, abbreviated RGB or RGBA. Each component has a value between 0.0 and 1.0, often these values can be represented by a value range between 0 and 255. Each value represents how much red, green and blue the color contains. Alpha is the transparency of the color. A color [1.0, 0.0, 0.0, 1.0] would be fully opaque red, color [0.0, 0.0, 0.0, 0.5] would be half transparent black and [255, 255, 255, 255] would be full white. Color manipulation is a major task that a shader does. Essentially a shader takes a color of a single pixel and outputs another one based on the programming. For example, a shader that simulates day-night cycle in a video game would be run on every pixel on the screen and based on the in-game daytime would alter the balance of white and black in every pixel being drawn. A shader can produce smooth color gradients and transitions which have many stylistic and attention drawing applications. (Figure 2)

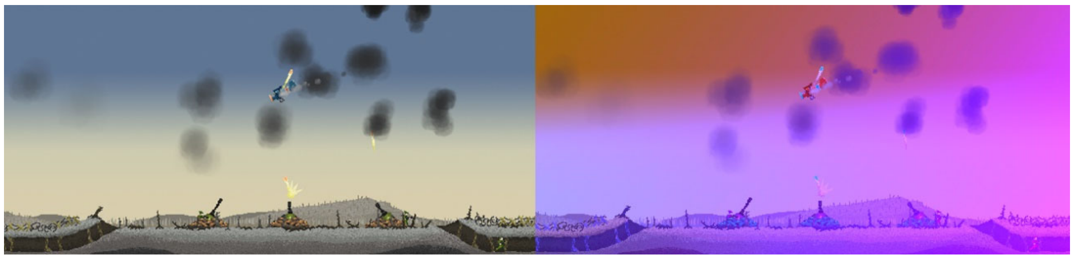


Figure 2. A hue shift and transparent-to-magenta gradient filters applied to a Triplane Furball scene.

3.2 Shapes

Shapes also have their design language like colours. Round shapes relate to soft and safe concepts, while jagged pointy shapes may represent danger. Symbols may have a meaning such as a heart representing health or an arrow progress or direction. Clean sharp lines are easy to read and convey order. Hazy and blurry shapes blend into surrounding graphics and may evoke uncertainty or motion. Shaders can be used to emphasise, alter or produce shapes in major ways.

A shader could be used to draw an outline around a game object such as a button, an important number or a pickable item. This outline helps the player notice the object being highlighted. The outline shader could be programmed to have a high contrast colour compared to the object being highlighted or even have some kind of motion to draw more attention.

One major use case for shaders is blurring of images. In a blur effect neighbouring pixels are blended with each other, producing an unfocused result especially in border areas of the drawn image. Many different blurring methods are used to achieve a desired effect. (Figure 3)

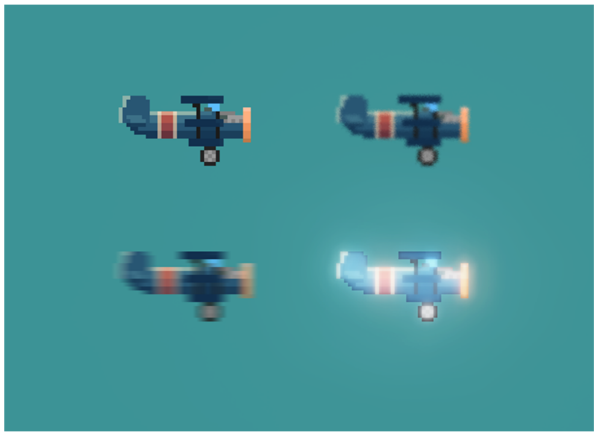


Figure 3. From top-left to bottom-right: no shader, a simple blur shader, vertical blur shader, a glow effect utilizing a type of blur.

On a moving object, blurring could be more intense in the direction the object is moving to provide a heightened sense of motion. The blur shader could consider the pixel's colour properties such as brightness to produce a glowing effect. Blurring all the border areas of the screen can draw focus to the centre. Blurring can be applied as a cost saving measure in a project to obscure low quality image assets for example in backgrounds.

Shaders can be used to produce shapes and full images. Essentially anything that can be mathematically defined can be produced by the shader. Applications of this are unlimited through infinite mathematical patterns and utilising randomness in the shader’s implementation. As shaders draw graphics on screen in real time, the designs can also be manipulated in real time. Real time shape manipulation has many very simple and effective use cases such as flowing patterns on a surface, hypnotic patterns and screen transitions.

3.3 Movement

In graphical design movement is used to draw attention to areas and objects or provide immersion. Movement can also convey actions and emotions. Shaders can be utilised to apply movement to otherwise static graphics. A shader can manipulate the drawn image in two ways. It can alter the dimensions of the sprite, stretching and skewing it or it can alter all the pixels inside the sprite. For example, a shader could draw the pixels of a flag sprite along a sine wave to make it flow in the wind, the properties of the wave could be altered in real time according to the game's wind conditions. Both techniques are showcased in the shader systems implemented for the commissioner.

4 Computer graphics

4.1 Shaders

Shaders are small computer programs that alter how game graphics are drawn on screen. In the context of 2D games there are two types of shaders, vertex and fragment shaders. Simply put, the vertex shader stretches the image, and the fragment shader looks at every pixel in the image and modifies it. These shaders are run on every pixel on screen and the result is a single color with a transparency value.

Shaders are written in shading languages. Popular shading languages are GLSL (OpenGL Shader Language) used in engines such as GameMaker and CryEngine and HLSL (High-level Shader Language) used in Unity and Unreal engines. Both GLSL and HLSL have syntax and semantics like C programming language. As most game engines use programming languages that are easier to learn than C, some beginner developers may avoid using shaders.

4.2 Graphics pipeline

The series of steps required to draw computer graphics on screen is called graphics pipeline or rendering pipeline. Through this process a representation of a 3D scene is drawn onto a 2D screen (De Bock 2013).

Two major parts of this pipeline are vertex and fragment shaders. During the drawing process a sprite is rendered on a quadrilateral having four sides called vertices, a rectangle constructed from two triangles. Colloquially this quadrilateral is called a quad. In the rendering process the vertices are passed through a vertex shader which calculates the vertices’ positions to be passed on. In this stage the quad’s shape could be altered from a rectangle into any shape with four corners. The geometrical data from vertex shader passes a rasterization process which converts the data into fragments which each represent a pixel. Each of the fragments pass through a fragment shader. In this process the fragment shader takes the sprites pixel data based on its assigned coordinate and potentially does modifications to the color data received and passes it on to be drawn on the screen. (Tufro 2018, 40–42).

4.3 Requirements and limitations

All the steps are performed on every pixel in every frame the game draws. In a game running in a resolution of 1920 x 1080 pixels and sixty frames per second the number of these calculations needed per second is 124,416,000. The count increases by a magnitude when increasing resolution and framerate. All this calculation is done by a GPU (graphics processing unit) that can handle these relatively simple but numerous calculations. Unlike CPUs (central processing unit) which are optimized to handle a stream of complex calculations with a handful of computational cores, GPUs are designed to distribute the stream of tasks to be concurrently processed by thousands of cores. The cores in GPUs are also specifically designed to handle certain mathematical calculations extremely fast. The process of distributing the task between multiple cores produces some limitations to shader programming. Cores cannot pass their output to other cores and do not remember what they were doing, the cores are essentially blind to everything else but the data they are given to process. (Gonzalez & Lowe 2015). This means that the implementation of shader effects includes the programming of the data inputs for the shader, which increases games complexity as the shaders are programmed in another language than the game utilizing them.

5 Game development

5.1 GameMaker

GameMaker is a primarily 2D game engine that can also be used to develop 3D games (YoYo Games 2023a). Engine is currently developed by YoYo Games based in Scotland. Engine was introduced in 1999 and has changed names during the years; these name changes coincide with major engine improvements. Core goals for the engine is to be accessible to new game developers, the engine's most recent versions however have expanded the tools more advanced users can utilize. GameMaker uses a proprietary scripting language known as GML. The syntax and features of the language are designed for easy game development. GML resembles JavaScript in syntax and structure and prioritizes clean and easy-to-understand code like Python. Users may also use a drag-and-drop visual style of scripting in GameMaker. GameMaker can export projects to PC, console and mobile platforms. In the current version of the engine shaders are written in GLSL ES shader language (YoYo Games 2023b). Other modern game engines function similarly compared to the GameMaker when thinking about graphical possibilities. The workflow and graphical pipeline may differ between engines, but a shader written in a language supported by two engines operates similarly in both. As such the graphic design principles remain the same even with different engines.

GameMaker games are built from game elements known as assets. Assets include rooms, sprites, sounds, paths, scripts, objects, fonts and shaders. The IDE (integrated development environment) comes with editors for all these assets. Assets files are compiled into a game and are not directly altered during the gameplay. Rooms are an asset that other assets are placed into and could be understood as the game world where everything happens. Objects are core building blocks of GameMaker games, they have several internal variables such as coordinates in the room and a sprite assigned to it. Objects also run code through events. There is plethora of events an object can have such as create event that is run when the object is created, step event that is performed every step of the game, events triggered by pressing specified buttons in a keyboard and draw events that are also run on every step but only handle code relating to drawing things on screen.

The most barebones GameMaker game consists of one room containing one object. This one object could control everything in the game or spawn other assets into the game. It is common to have such controller objects, but games are often built by placing assets manually into the game in the room editor. Assets in a room are placed into a layer. Some assets, such as paths, can only be placed in a layer of specific type. Layers have a draw order, images in the higher layers obscure images in the underlying layers. This order is determined by layer depth ranging from -16000 to 16000, with smaller depth being higher in the hierarchy. Mockup of how layers can be used in the image below. (Figure 4)

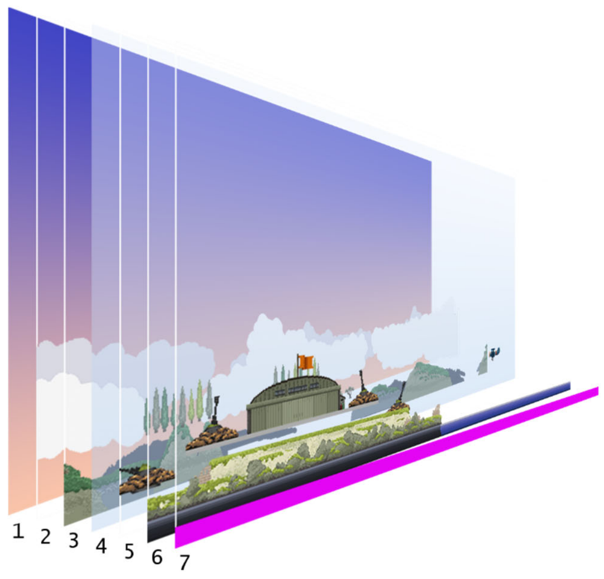
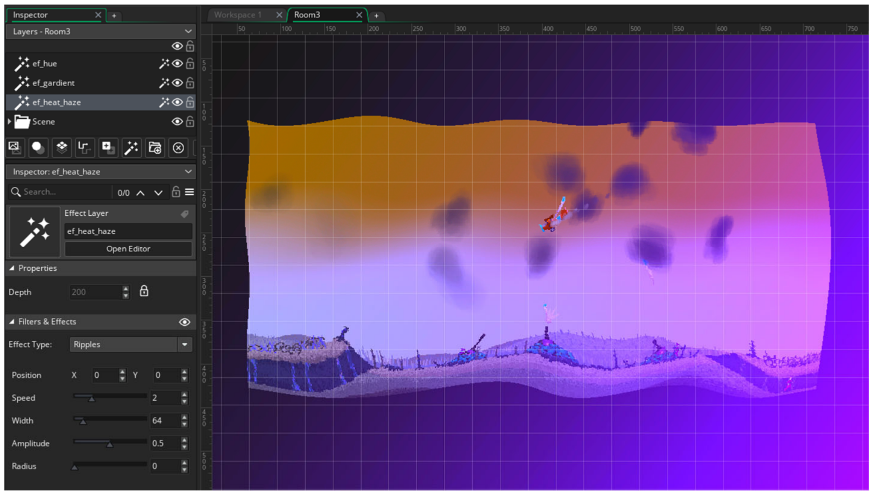


Figure 4. In numerical order: background, clouds, terrain in horizon, shading layer, game objects, foreground terrain, area used by the water effect.

A special shader, called filter, can be applied to a layer that affects every drawn asset in the layer. Filters are ready made shaders in GameMaker engine and as such offer an extremely easy way to incorporate shaders into a video game being worked on. These filters can be applied to an effect layer that applies the filter to all the layers underneath it. (Figure 5)



Effect Layer: ef_heat_haze
Depth 200
Effect Type: Ripples
Position X 0 Y 0
Speed 2
Width 64
Amplitude 0.5
Radius 0

Figure 5. Room editor showing multiple filter layers and adjustment controls.

Filters offer an exceptional way to broaden the developers' thinking of what kind of results shaders can provide. Working with these filters is a quite rigid process as their parameters cannot easily be altered while the game is running. As a part of this thesis a tool was developed to enable applying a filter to existing layers during the game to alleviate this problem.

Dynamic resources can be created during the game. These assets require some extra attention as the game must always check if they exist in-game and they need to be manually cleared from memory when they are not needed anymore. Important resource pertaining to the topic of shaders is surfaces. Surfaces are essentially blank canvases that can be drawn into and then drawn on screen like a sprite. At any point of this process a shader can be applied, when an image or a shape is being drawn on a surface and when the completed surface is being drawn. The water effect implemented in the commissioner’s game revolves mainly around this technique.

5.2 Graphics production resources

GameMaker offers many ways to draw sprites and other visual assets on screen. Two main ways are assigning a sprite to an object and drawing a sprite directly using a sprite drawing function. GameMaker can also be used to draw simple geometric shapes which can be combined to create more intricate shapes. These methods can be altered with simple color transformations, rotations and image size scaling to produce a wide range of visuals. In the images below are some simple examples of these methods and how they are achieved. (Figure 6 and Figure 7) In Figure 6 the object in the middle drawing the plane sprite assigned to it and many variations of the sprite around it. The object also draws a line across the screen and the multicolored background. The white planes are drawn through shader.

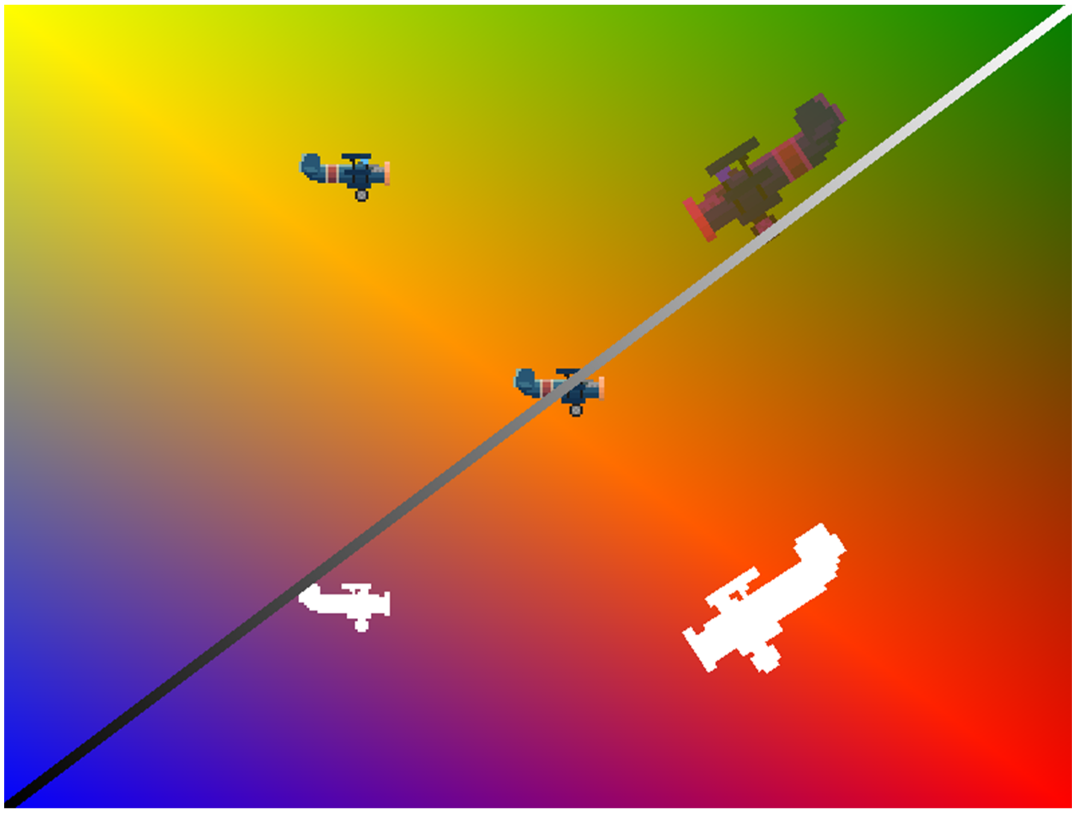
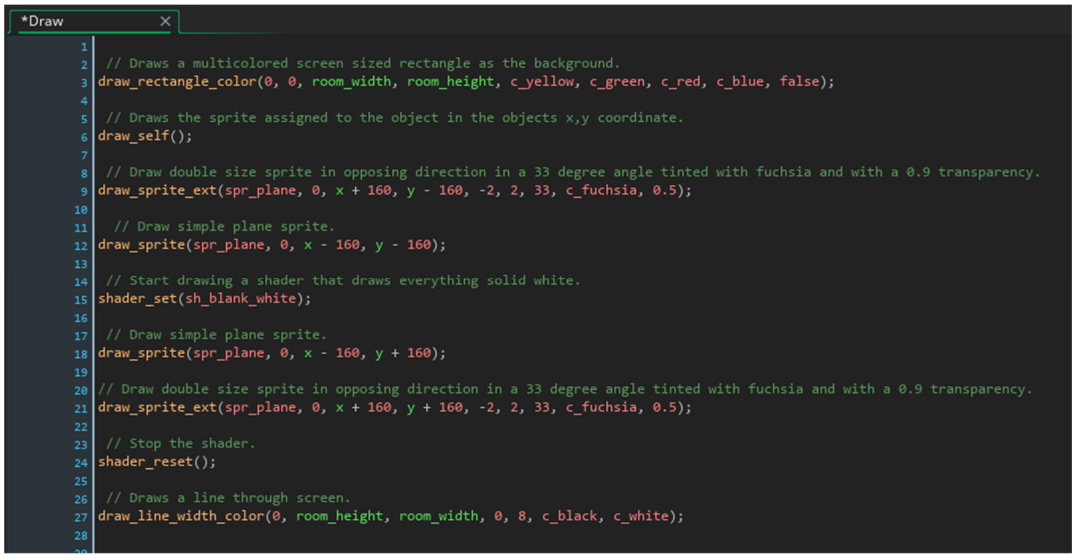


Figure 6. Object drawing multiple sprites on screen in different ways.



```
 // Draws a multicolored screen sized rectangle as the background.
draw_rectangle_color(0, 0, room_width, room_height, c_yellow, c_green, c_red, c_blue, false);

 // Draws the sprite assigned to the object in the objects x,y coordinate.
draw_self();

 // Draw double size sprite in opposing direction in a 33 degree angle tinted with fuchsia and with a 0.9 transparency.
draw_sprite_ext(spr_plane, 0, x + 160, y - 160, -2, 2, 33, c_fuchsia, 0.5);

  // Draw simple plane sprite.
draw_sprite(spr_plane, 0, x - 160, y - 160);

 // Start drawing a shader that draws everything solid white.
shader_set(sh_blank_white);

 // Draw simple plane sprite.
draw_sprite(spr_plane, 0, x - 160, y + 160);

// Draw double size sprite in opposing direction in a 33 degree angle tinted with fuchsia and with a 0.9 transparency.
draw_sprite_ext(spr_plane, 0, x + 160, y + 160, -2, 2, 33, c_fuchsia, 0.5);

 // Stop the shader.
shader_reset();

 // Draws a line through screen.
draw_line_width_color(0, room_height, room_width, 0, 8, c_black, c_white);

```

Figure 7. The code in the “Draw” -event of the object in Figure X.

With some creativity, quite complex graphical implementations can be achieved. It is sometimes preferable to only rely on these methods as there is a low barrier for their implementation. For example, some art styles may not require complex color shadings or a simple outline for a button could be achieved with an alternative button sprite if the game contains only a few buttons. In a larger project some of these implementations may become unwieldy to maintain and scale up, this is especially dangerous in a project that is not well defined. This danger is present in all programming and can only be alleviated with proper planning. From the development resource management standpoint, it becomes vital to understand what techniques offer the most efficient implementation.

Video games are an interactive medium which means that game graphics need to be produced in a flexible and efficient way. As hardware performance has reached the point where a 2D game’s graphic designer does not need to be overtly careful about the required processing power and memory, more concern is instead needed on the available graphics production resources. Game graphics need to adapt to the current situation in the game and be reusable in larger games. This means that game components such as characters, environments and menus are often constructed from multiple parts that can be rearranged and modified on the fly.

Excellent example of a case where a game component is assembled from multiple assets is a character with a body sprite, changeable clothes as separate sprites and a weapon with its own sprite. These sprites could be moved, rotated and mirrored during gameplay in many ways, this would mean that if the sprites would, for example, have shadows drawn on them the shadows could look weird in some situations. This weirdness could lead to reduced immersion and break the player's emotional investment into the game. It would not be reasonable to create separate assets for different lighting conditions for each character, so shaders would offer a way to create a unified look with shadows being drawn on the proper side of the character. In addition to uniform look between assets such a shader would also reduce the time needed to produce the sprites, as the process of drawing shadows could be skipped. If the image assets in the project would share similar color adjustments the benefits would be manyfold. The time and cost savings would scale with the project size and return of investment in shader implementation would become more apparent. Shaders could be used to generate custom textures and even sprites on the fly, further reducing the needed resources for sprite production.

As shaders are subprograms often run on a game engine, they are very transferable between projects. This means that there exists an extensive number of readymade shaders to be incorporated into a project. Game engines, such as GameMaker, often have asset stores from which various game components can be acquired. These available shader assets could range from simple outline shaders into full post-processing systems that can be used to fully transform the look of the game with minimal technical expertise. With these assets common graphical implementations can be achieved with minimal development resources.

6 Triplane Furball

Triplane Furball is a 2D side scrolling dogfighting game. The game is a reimagining of Triplane Turmoil computer game made by a Finnish game company Dodekaedron Software in 1996. In the game, players pilot World War I era fighter planes in single player missions and local multiplayer skirmish battles. The multiplayer map is presented in a unique way, where the map is the length of three monitor widths and is stacked on three layers. (Figure 8) This introduces some unique graphical challenges.

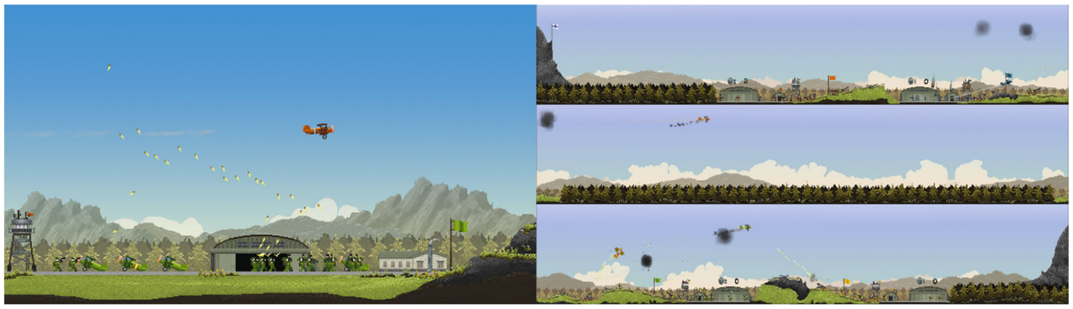


Figure 8. Single player and multiplayer views in Triplane Furball.

The multiplayer view is constructed by having three cameras each capturing a view of a third of the whole map, these views are then drawn on screen in three layers. The drawback of this is that many of the graphical rendering processes must be done thrice, once per each camera capturing the scene. Though the game is not very demanding graphically, this is something that needs to be kept in mind when adding graphical effects as it might impact performance. Also, in the case of the implemented water effect system extra steps are required to draw the effect first onto three surfaces that are drawn on their proper places on the screen. The single player view is recorded by one scrolling camera centered on the player. This means that some of the effects need to adapt to two completely different view types.

Effects that liven up the game world pose a significant design challenge. There are limited resources to produce graphics, so there must be a limited number of graphical assets and the process of putting them into the game needs to be frictionless. This means that single and multiplayer maps need to share assets and systems and they must accommodate the limitations of both game types. The distinctive multiplayer view prevents the usage of some of the more common effects used to increase immersion in a game world. As all players in the multiplayer mode share the same static screen, utilizing a parallax effect that relies on screen movement becomes impossible.

7 Implementation

The original commission was to create a reflecting water shader for the commissioner’s game. As a part of the work four different types of shader implementations were done to explore the possibilities that shaders offer. Simple color transformer, tool that enables the developer to apply filters to game layers in real time, shader system that draws random trees on a map waving in wind and the reflecting water shader.

7.1 Color transform

This was done solely to create an extremely simple shader solution that is still practical and serves a purpose. The shader turns all the pixels that it is assigned to full white, while considering the transparent areas. This kind of an effect could be used in a game when the object takes damage, the shader would then be applied to the object for a few frames. Program 1 and Program 2 show how simply a shader can be done and applied to an object. The Program 2 draws the object always as solid white, so it should be altered in ways that control the activation of the shader. This could be achieved by applying a conditional statement to the starting and stopping of the shader.

Program 1 The solid white color transform GLSL fragment shader program in its entirety.

```
varying vec2 v_vTexcoord;
varying vec4 v_vColour;

void main() {
    // Get the color data of the sprite pixel in the output
    //coordinate.
```

```
        vec4 color = texture2D(gm_BaseTexture, v_vTexcoord);
        // Output a full white color
        // with the alpha value of the sprite.
        gl_FragColor = vec4(1.0, 1.0, 1.0, color.a);
    }
```

Program 2 A simple shader is applied in the GameMaker object Draw-event in GML.

```
    shader_set(shd_solid_white); // Start the solid white shader.
    draw_self(); // Draw the sprite assigned to the object on screen.
    shader_reset(); // Stop the shader.
```

Though the shader can seem insignificant it perfectly sums the power of shaders. The same effect could be achieved without shaders in other ways. The simplest way would be to create a completely white sprite for the object and swap the sprite when the shader would be needed. This would mean that if other objects that have different sprites would need their own white sprites, the number of extra sprites required for this effect alone would quickly become too much to effectively manage. The more scalable solution would be to use surfaces, but the implementation would require more steps and would as a result be a more complex and volatile solution as more care needs to be used when working with surfaces to prevent memory leaks. The shader implementation could also be refined furthermore easily and is more transferable between projects.

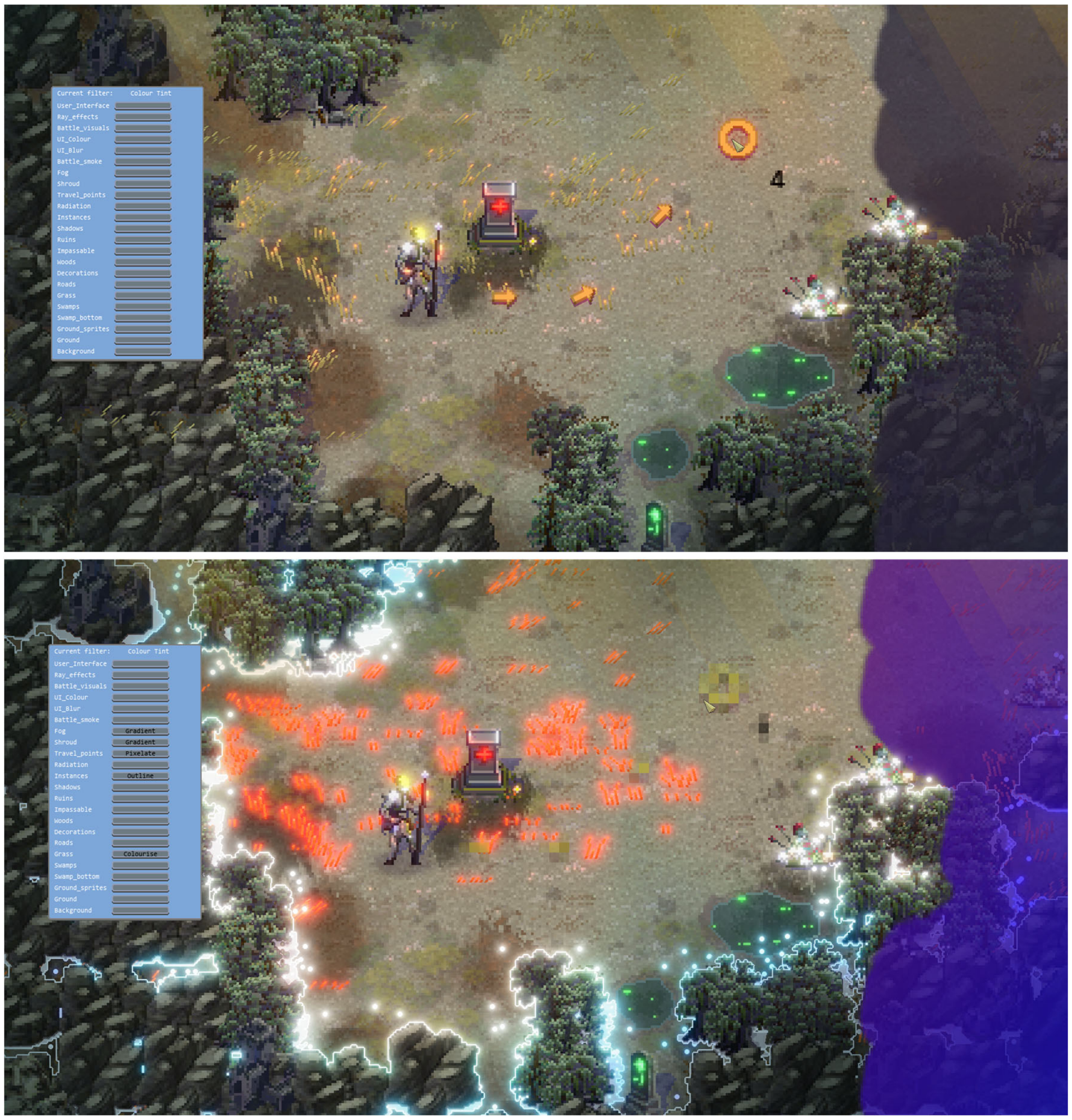
7.2 Filter tester

In GameMaker the room editor provides the means to quickly apply shaders to entire layers, which affect how all the graphics in the layer are drawn. However, the room editor does not accurately represent how the game looks in runtime.

For instance, in Triplane Furball the background sky and rolling clouds are generated at the start of the map, in the room editor the background is a solid static color. This means that applying and adjusting the shader layers becomes a chore as the starting of the game to assess the result takes even in the smallest projects several seconds. These seconds and in large projects minutes add up and hinder the design process. A simple GameMaker tool was developed to apply the shaders to layers in-game.

The tool is a single object that can be dropped into any GameMaker project and can be used with minimal adjustments. How the tool works is that the controlling object detects all the layers in the room and then creates buttons for each layer, the buttons can be used to select and activate a shader for the corresponding layer. The tool can be used to explore the effects that filter effects have in a running game. It can be used to discover new visual effect combinations effortlessly. The tool could also showcase the impact of enabling and disabling full screen effects. Furthermore, the tool could be utilized in the design and testing processes. For example, graphic designers could apply filters that affect game object recognition or simulate color blindness to certain layers to study the readability of the game objects.

The tools technical implementation also shows that the shaders are not the only part contributing to the complexity of shader systems. Although the shaders themselves are built into the GameMaker engine, applying them through code can be quite tedious. Even more tedious task is the implementation of controls to all the adjustments these shaders have. The tool has been employed in the commissioner’s other projects. (Figure 9) Though very useful, the tool could be greatly expanded upon with the adjustment controls.



```
Current filter:    Colour Tint
User_Interface
Ray_effects
Battle_visuals
UI_Colour
UI_Blur
Battle_smoke
Fog
Shroud
Travel_points
Radiation
Instances
Shadows
Ruins
Impassable
Woods
Decorations
Roads
Grass
Swamps
Swamp_bottom
Ground_sprites
Ground
Background
```

```
Current filter:    Colour Tint
User_Interface
Ray_effects
Battle_visuals
UI_Colour
UI_Blur
Battle_smoke
Fog               Gradient
Shroud            Gradient
Travel_points     Pixelate
Radiation
Instances         Outline
Shadows
Ruins
Impassable
Woods
Decorations
Roads
Grass             Colourise
Swamps
Swamp_bottom
Ground_sprites
Ground
Background
```

Figure 9. The filter tester used to apply filters in-game.

7.3 Vertex tree shader

During thesis work the commissioners game went through a sizable refactoring process, as a result many of the graphical assets needed to be adjusted to fit the larger map sizes, meaning that the previously hand drawn maps needed to be drawn again. The old maps were also deemed to be a bit too static and lifeless. This gave a good opportunity to improve and streamline the map production process as two thirds of the maps did not exist at all. The old maps had trees that were placed into the map sprites by hand in an image editor. A shader system designed by Gurpreet Singh Matharoo (Matharoo 2020) was used to draw randomized tree sprites on screen. The system uses vertex and fragment shaders to animate the trees. The system is designed to be used in top-down perspective games and was modified to accommodate the side-on view of Triplane Furball. The original system is an object that fills an area randomly with the trees. In the modified system this fillable area is a combination of a path that travels through a map at ground level and designated zones of vegetation. (Figure 10)

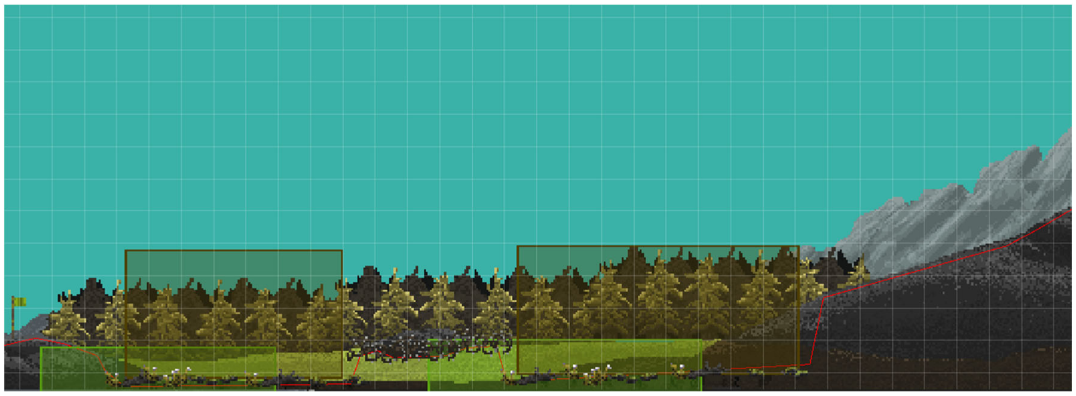
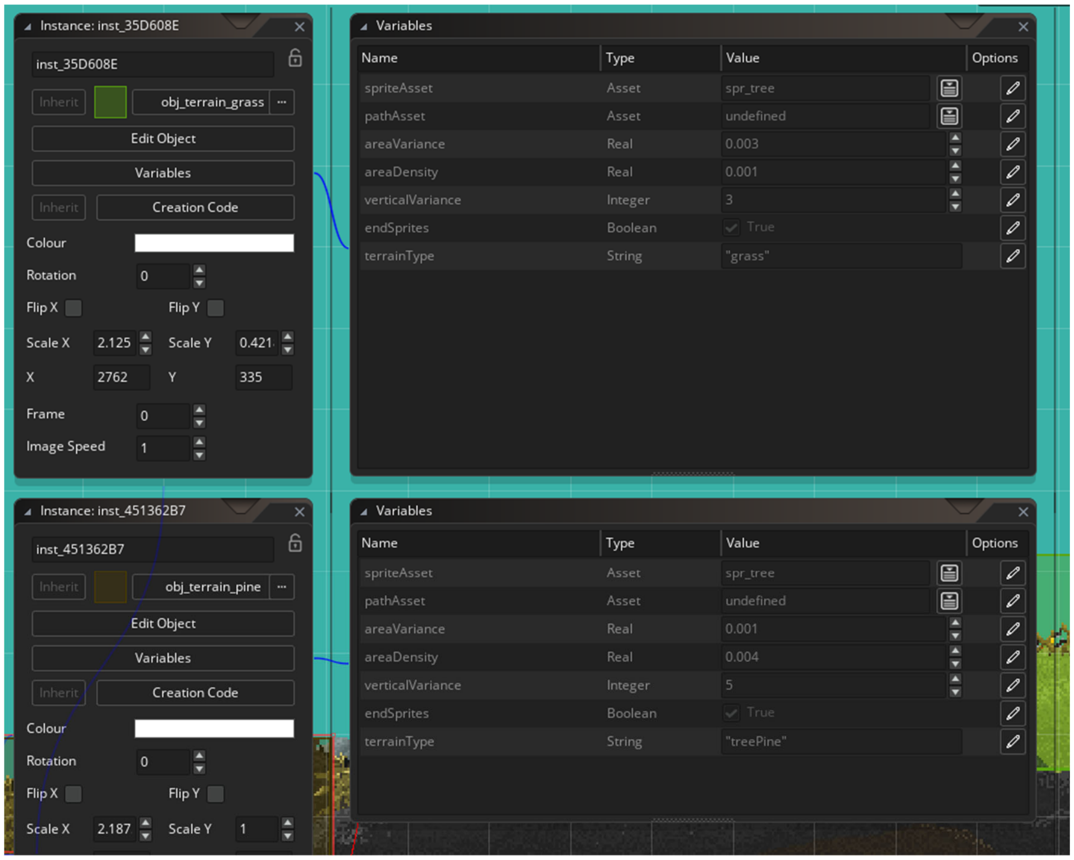


Figure 10. Room editor view showing tree (brown) and undergrowth (green) zones. The height path is the thin red line.

Each instance of the zone object first gets the room’s designated height path when created. The object then generates a list of coordinates along the given path, chosen randomly between its sprite's right-left boundaries. Randomness and other relevant variables can be defined through the Variable Definitions in the room editor for each object. (Figure 11)



Instance: inst_35D608E

inst_35D608E

obj_terrain_grass

Colour

Rotation 0

Scale X 2.125 Scale Y 0.421

X 2762 Y 335

Frame 0

Image Speed 1

Variables

Name	Type	Value
spriteAsset	Asset	spr_tree
pathAsset	Asset	undefined
areaVariance	Real	0.003
areaDensity	Real	0.001
verticalVariance	Integer	3
endSprites	Boolean	✓ True
terrainType	String	"grass"

Instance: inst_451362B7

inst_451362B7

obj_terrain_pine

Colour

Rotation 0

Scale X 2.187 Scale Y 1

Variables

Name	Type	Value
spriteAsset	Asset	spr_tree
pathAsset	Asset	undefined
areaVariance	Real	0.001
areaDensity	Real	0.004
verticalVariance	Integer	5
endSprites	Boolean	✓ True
terrainType	String	"treePine"

Figure 11. Control variables the zone objects have.

The "terrainType" -variable determines what type of sprite is placed at the coordinates the object generates. Depending on the terrain type other actions can also be dictated, such as generation of background elements for the object. Each instance of the zone objects sends the coordinates into a controller object that handles the transformation of data between objects. The controller holds the data arrays for each of the terrain types. If an array is not empty, the controller will then create objects that handle the drawing of each of the terrain types. These objects are given a corresponding array of coordinates and a sprite asset. The object then creates a buffer and populates it with sprites in given coordinates. Each of the objects then draws the buffer into the game, animating what is drawn with a shader. (Figure 12)

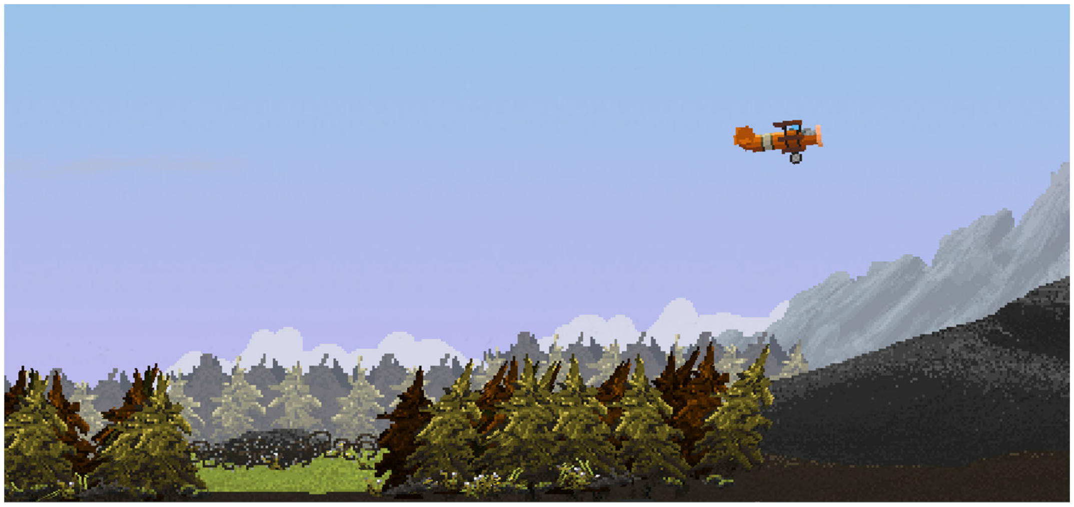


Figure 12. The result of the tree vertex shader.

The system can be expanded in many ways. The shader can be given various values on how the sprites are animated. These values can be altered during game, for example the sway speed and strength could be affected by wind speed and direction. Each of the terrain types could have their own shaders to more accurately simulate their behavior in wind. The system can be used to quickly fill areas with different kinds of animated sprites, cutting down development time. Similar effect could be achieved by placing sprites in the room through code, but they would have to be animated by hand and the animation could not be controlled. Individual tree objects would be the third option, but it would get very inefficient given how many trees there needs to be. The vertex buffer method allows the placement of thousands of trees with minimal performance impact. As a result, this system provides an animate and organic look for the forests with minimal effort.

This system could be made more flexible. The terrain types are hard coded and there can be no variation inside a room for example with tree types. Adding different types of terrain is cumbersome with the requirement of the parent object needing to have the cases for each terrain. The height path needs to be drawn manually in the room editor in each map which takes time, but it has the benefit of having more control compared to the more automated systems which also proved to be too inefficient. However, these are not major issues, given the scope and focus of the project.

7.4 Water shader

The main task of the commission work was the creation of a water effect for the game maps. The effect would need to have two properties: it would have to mirror the player and game world and it needed to be animated in some way. The first assumption of the mirroring effect implementation was that it would be challenging to achieve because of the tri-camera used in the multiplayer game and thus it was placed as a second priority for the water effect. The realistic wave effect would be an effect that would be extremely inefficient to bring to life with anything other than a shader. Though the water effect would be completely achievable only with a shader, the necessity of chopping up the result to three parts with surfaces was the reason for the hybrid approach. This also simplified the shader itself making it easier to maintain and refactor in future.

The system is used to create areas of water. These areas are single objects, in most maps this object will be stretched across the entire width of the map. The object has a simple blue water texture on itself. The effect is created by creating a surface that is the size of the water object with some transparent space added on top. This surface acts as a texture for the shader. As a shader cannot draw outside the texture it has been assigned to, the transparent area is needed for the waves to be drawn. The object also creates another surface above itself. On this capture surface is drawn everything on the screen in the surface’s area. This capture surface is then mirrored vertically and vertically scaled down by a one-to-three ratio using the draw_surface_part_ext() -function. Squashing down the captured area creates an illusion of depth and adds more content for the often-small water area. The surface is also given a grey color tint to more clearly distinct the mirrored object from real ones. The capture surface is then drawn, through a wave shader, on the first surface that was created. The resulting surface can then be drawn on screen as is or in three parts based on the current map.

The water shader works by creating horizontal and vertical waves. These waves are made by combining two different sine waves, which when added together create a more random and natural wave pattern. To make the waves move, game time is passed on to the shader to be a component in the sine wave. The shader takes its coordinate and alters its position based on the waves. This new coordinate is used to pick a color in the texture to be drawn in the shader's original coordinate. Program 3 depicts all these steps taken for the required result.

Program 3 Main function of the GLSL shader that creates a wavy motion on a texture.

```
void main(){
    // Set up pixel coordinates
    vec2 p = v_vTexcoord;
    float pY = 1.0 - p.y;
    float pX = 1.0 - p.x;

    // Get a normalized value for each pixel.
    float pixelW = (p.x / waterW);
    float pixelH = (p.y / waterH);

    // Combine two different, opposing, sine waves to create a more
    // random wave pattern. Adjust x,y coordinate based on the waves.
    p.y += pY * (
        sin((pixelW * 0.1) + time) * (1.3 * waterH) +
        sin((pixelW * 0.15) - time * 1.2) * (1.5 * waterH)
    );
    p.x += pX * (
        sin((pixelH * 0.1) + time) * (1.1 * waterW) +
        sin((pixelH * 0.15) - time * 1.1) * (1.4 * waterW)
    );

    // Get the final color by looking at the color in texture
    //at the modified coordinate and
    // multiplying it with the base color.
    gl_FragColor = v_vColour * texture2D( gm_BaseTexture, p);

    // Make sure the pixel is drawn with full alpha
    // if not completely transparent.
    gl_FragColor.a = ceil(gl_FragColor.a);
}
```

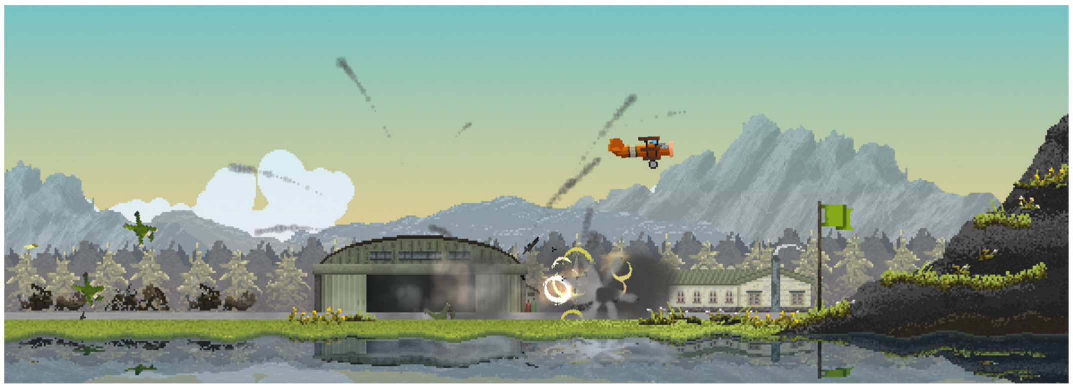


Figure 13. Final water effect in action.

The result is an effect that fits well with the art style of the game. (Figure 12) The system showcases the power of shaders well as the effect creates a very different visual feeling with the added benefit of being easily modified and expanded upon. Reflections can be considered as one of the most powerful methods of immersing the player in the game world. The system reduces map production workload by a lot as there is no need to draw the water areas in the maps anymore. Performance vise the system is very efficient but could be improved upon by further optimizing the code. The biggest issues are how surfaces are handled and there exist many ways to enhance the process.

These improvements become a requirement if multiple water objects are needed down the line, but they are easily implemented and already planned for.

8 Conclusion

The goal for the thesis was to research when and where it is appropriate to use shaders in a 2D game. The topic of when and where shaders should be used is hard to tackle. While there is no simple method to quantifying the benefits of some of the results shaders provide, at the same time it is definitive that many graphical implementations where shaders are employed are only possible with shaders.

During the implementation process it became apparent that although shaders can be seen as difficult, with extensive use of linear algebra, trigonometry and stricter programming languages, the equally complex part in integrating shaders into a project can be the surrounding system that utilizes the shaders. One of the main advantages the shaders possess are that these systems serve as a vector for both the programmers and the artists to affect the looks of the game. For an artist, the ability to adjust a few variables in a shader system to make significant changes to a game is a potent tool. Similarly, for a programmer, the capability to integrate another system with a shader for visualizing a new feature is equally powerful. As much as gorgeous graphics matter in video games, more than that what matters is to ensure the complementation of game’s development. Shaders and other programmable graphics solutions leverage the code to do the heavy lifting in an ambitious game project.

The topic discussed in this thesis is very broad but necessary, as the subject requires a more general understanding of visual design and production in games. Programmable graphics is a field that has endless possibilities and applications. Most of the subtopics warrant deeper handling in other works. Game development can be seen as an all-encompassing trade where technical engineering and artistry meet with marketing, business management and psychology.

References

DeBock, S. 2013. Introduction to the Graphics Pipeline. Graphics Pipeline. Accessed: 15.11.2023. https://www.gamedev.net/tutorials/programming/graphics/introduction-to-the-graphics-pipeline-r3344/.

Gonzalez, P. & Lowe, J. 2015. What is a shader? Why are Shaders famously painful? Accessed 15.11.2023. https://thebookofshaders.com/01/.

Meriläinen, M. 2014. Applying Colour Theory to Game Development. Thesis. Business Information Technology. Kajaani: Kajaani University of Applied Sciences. Accessed 15.11.2023. https://urn.fi/URN:NBN:fi:amk-2014120919075.

Fast GRASS Tutorial -- GameMaker Studio 2 (Vertex Buffers for Beginners). 20.3.2020. GameMakerStation - Matharoo. https://www.youtube.com/watch?v=MTdAdVt4EIM.

Schell, J. 2019. The Art of Game Design: A Book of Lenses, 3rd edition, A K Peters/CRC Press, E-book available on Amazon. Accessed: 15.11.2023.

SteamDB 2023. Steam Game Release Summary. Steam "2D" Game Releases by Month. Accessed: 15.11.2023. https://steamdb.info/stats/releases/?tagid=3871.

Tufro, F. 2018. 2D Shader Development. 40–42. Hidden People Club. Accessed 15.11.2023. E-book available at https://www.2dshaders.com/.

YoYo Games. 2023a. Introduction To GameMaker. Accessed 15.11.2023. https://manual.yoyogames.com/Introduction/Introduction_To_GameMaker_Studio_2.htm.

YoYo Games. 2023b. Shaders. Accessed 15.11.2023. https://manual.yoyogames.com/GameMaker_Language/GML_Reference/Asset_Management/Shaders/Shaders.htm.

Zukowski, C. 2023 My advice to AAA-developers trying to make indie games. Lesson #4: Art attracts but gameplay sells. Accessed: 15.11.2023.

https://howtomarketagame.com/2023/11/06/my-advice-to-aaa-developers-trying-to-make-indie-games/.