Jarod Moore
CSCE 4250
01 May 2025

Shaders - Overview

For this project, I chose to explore shaders. Going into this project I didn't understand what shaders were, or how extensively they actually are used in video games. This project taught me an incredible amount about Graphics programming, the math and theory that goes into it, and the wide array of interesting and cool applications that it has.

My project was completed in the Godot engine, version 4.4, using their rendering pipeline, and shader programming language. This language was similar in structure and syntax to GLSL, while being a little simplified, in accordance with the way the engine works. This worked to my advantage, by allowing me to explore shaders and graphics programming through an easy avenue, before easing myself into exploring methods and implementations done in GLSL, and using the theory and processes for my shaders in Godot. This project gave me a love for graphics programming, and while I'm still new at it all, I'm very excited to continue exploring it after the semester ends.

That aside, my project consisted of me producing various screen-space and spatial shaders that I was interested in, which will be overviewed below:

Spatial, mesh based shaders:

Simple Water Shader:

This was my first introductory shader, and I followed the guide in the godot documentation to learn it. It introduced me to the basic functionality of vertex and fragment shaders, by creating a shader for a quad mesh that resembles water.

"Magic Orb" Shaders:

This was my next exploration. Wanting to understand more, I decided to make a weird "Magic fantasy orb" thing, using a sphere mesh as a basis. This sphere would be procedurally warped in the vertex shader using a "generic 1,2,3 noise" function, which also served as part of a function that creates a visual "breathing" color effect.

The fragment component of the shader also featured basic color emissions, blending multiple colors for different areas of the mesh, adding highlights dependent on distance from the mesh origin, and basic mesh rotation. Most of these were also modified using the TIME, and trig functions to create procedural effects.

"Fireball":

This shader utilizes a noise texture to warp the fragment color in both the vertical and horizontal direction. Using color banding, four colors were applied to the flame, with the fourth color being applied as the “core” of the flame. By utilizing the noise texture, the effect of a spinning, volatile fireball is created.

This was additionally re-used to create “wisps” that could emit light, and would bob up and down in place, through a vertex shader, that would offset the vertices using a sine wave and the game’s current time.

Parallax mapped ice with refraction:

The shader that gave me the most trouble overall, but with a result that I’m very happy with. The ice uses parallax mapping, adapted from Joey de Vries’ LearnOpenGL. This style of parallax mapping uses a texture, in this case of ice, and a black and white displacement map of the texture to create the illusion of depth. This is done by using the tangent-space view direction of the texture, and offsetting the fragment at the current coordinate by a certain height, determined through using the displacement map. By doing this, we can “displace” the fragment’s texture to the determined height, creating what appears to be depth within the object. This is also affected by the viewing angle, which allows parts of the object to overlap others, which isn’t possible by using just normal mapping.

Additionally, this ice shader incorporates refraction, through the use of the object’s tangent, binormal and normal, by adjusting the screen texture UV that is projected onto the object. This effect is done by utilizing the normal map, the object surface normals, and offsetting the screen UV using “ref_normals”, created using the refracted surface normals of the object. The resulting refracted UVs can be applied to the depth and screen textures, creating the refracted or distorted effect on the objects behind, based upon the viewing angle of the camera.

While working on the ice refraction I discovered that this spatial shader didn’t take into account any screen-space post processing that was applied for the player, which applies outlines and a color palette to the player’s view. This led to objects in the reflections appearing to be a different color than what the player was seeing them as. To fix this, I utilized one of Godot’s subviewports to sample the player’s camera view, culling out the mesh with the ice shader applied, and using it as a texture to replace the “screen texture” in the shader. This allowed the ice to still accurately refract the view of the objects behind, when looking into it, while also applying the post-processing effects of my screen-space shaders.

Screen-Space shaders:

Four screen-space shaders were created, all built into one shader file, almost exclusively as a fragment shader. Each of these functions is housed in an if statement, allowing them to be toggled independently of one another, without any problems.

Sobel Operator Outlines:

The Sobel Operator is an edge detection algorithm. It uses two 3x3 kernels that are "convolved" with the original image, in this case the game's screen texture, to approximate the derivatives of the image, in both the horizontal and vertical directions. This has the effect of finding the edges and producing a black and white version of the original image with the detected outlines in white, and the rest of the image as black.

This operation was used on the depth buffer, which is used to sort objects in rendered 3D space, and the normal buffer, which is used for determining the orientation of the vertices on an object in 3D rendered space, usually for lighting. By using the operation on both the depth and the normal, we can create more "full" outlines, because it takes into account both the changes in depth between two objects, and the changes in brightness within the same object. The resulting image can be mixed with an outline color, and be applied on-top of the original screen texture output to produce procedural edge outlines.

Since creating this component of the fragment shader I've discovered that some of the other methods like the Difference of Gaussians, can be used alongside the sobel operator to produce better quality results.

Color Quantization/Posterization:

Posterization (as used in the context of image editing), is the act of reducing a color gradient to fewer regions of color, creating more abrupt changes in color. This can result in "color banding", which is a result where the regions of color can't adequately sample the gradient of color, creating the effect of "bands" of distinct colors instead.

This shader effect was created intentionally for the purpose of reducing the number of colors in an image, to be a component in creating a "cel-shaded" look. It simply works by determining a number of color "levels", in my case a default of 8 levels. The posterization is then applied to every fragment of the screen texture, multiplying the value of the fragment by the number of levels, and rounding it to an integer, before dividing it by the color levels again.

Color Palette application:

Piggybacking off of the posterization of the screen texture, we can apply a custom limited color palette to the image by comparing the color of the current fragment against each color in the palette, to determine which color is the closest to it. This is done by iterating through the palette's colors, and taking the difference of the color by subtracting the palette color from the fragment color. The distance from the original color then is the result of the dot product of the difference with itself. This is then compared against the current "closest distance" color, replacing it if the new distance is closer than the current closest color. This is repeated until every color has been compared, outputting the closest color to the fragment.

Dithering:

Dithering is a method of applying intentional noise to an image to accommodate for quantization error, to reduce the effects or prevent something like color banding, which was

mentioned in the posterization section above. It creates a black and white “gradient” by changing the density of black dots in the image to create a more natural “gradient” between the color bands. When used on a black and white image, the effect looks similar to the “stippling” shading done in traditional art, where dots are used to shade an object, with greater density creating a darker region. Dithering can be used to offset the color banding added by the posterization, or to add an interesting stylistic effect to shadows and color gradients.

All of these screen-space shaders when used together create an interesting cel-shaded or “comic book” visual style, and the color-palette adds another layer to that effect, by allowing you to curate the exact colors used, giving you control over the exact way a scene feels, or to just re-create the monochrome green style of old monitors, or the original GameBoy.

Added 09/13/2025:

The screen-space shader has been uploaded to godotshaders.com here, including in-code comments.