Innovative Shader Design: An Interactive Editor with Real-Time Preview

Marilyne Bassoulou

Master's Project Thesis
MSc Computer Animation and Visual Effects

Bournemouth University
August 2024

Abstract

This paper documents the design and implementation of an interactive shader editor tool which consists of a node-based interface and a code-based interface. The main goal of the tool is to provide developers with a platform to create, modify, and compile shaders with a real time preview of the changes. Moreover, it includes a node-based interface which is tailored towards artists where they can develop 2D shaders without prior programming knowledge. The editor includes real-time shader previewing, offering instant visual feedback on any modifications. This integrated approach aims to improve accessibility and usability for artists and developers, creating a more inclusive environment for shader development. The tool addresses the needs of both visual and code-based workflows to simplify shader creation and expand its appeal.

Table of Contents

List of Figures

Introduction

Shaders are fundamental components in modern computer graphics, responsible for determining how surfaces and effects are rendered on-screen. They are small programs that run on the GPU, controlling the final appearance of pixels, vertices, and textures in a scene. Shaders play a crucial role in today's world through visual effects such as material properties, lighting and even shadows. They are important in achieving realistic graphics in various applications, from simulations to animations to games. Shaders come in different types, including vertex shaders, fragment shaders, and compute shaders. Vertex shaders process each vertex's data and determine its position in 3D space, while fragment shaders compute the color and other attributes of each pixel on the screen (Jeremias and Quilez 2014). Compute shaders, on the other hand, are used for general-purpose computing tasks that can be parallelized across the GPU (Mannub and Hinkenjann 2005).

Shader development has traditionally been a complex and technical task, often requiring extensive knowledge of graphics programming. Recent advancements have started to simplify this process by introducing more intuitive tools and interfaces. Interactive shader development has emerged as a key approach, facilitating real-time experimentation and iteration (Jensen et al. 2007). This method allows users to interact with shaders dynamically, providing immediate feedback that is essential for refining visual effects while optimizing performance. Interactive shader development approaches facilitate a more intuitive interaction with shaders by allowing users to see the effects of their changes immediately. This has led to the creation of innovative tools that combine visual and code-based editing methods. Node-based systems, for example, simplify shader creation by allowing users to construct shaders through visual nodes that represent various operations and parameters. This approach not only enhances accessibility for artists but also provides a robust platform for developers to fine-tune and extend shader functionality (Mannub and Hinkenjann 2005). In addition to the node-based interface, the editor features an integrated code editor. This component provides developers with the ability to write and modify shader code directly, leveraging advanced code editing features such as syntax highlighting and error checking. The combination of visual and textual editing tools allows for a flexible development environment where both artists and developers can work concurrently and efficiently (Jeremias and Quilez 2014).

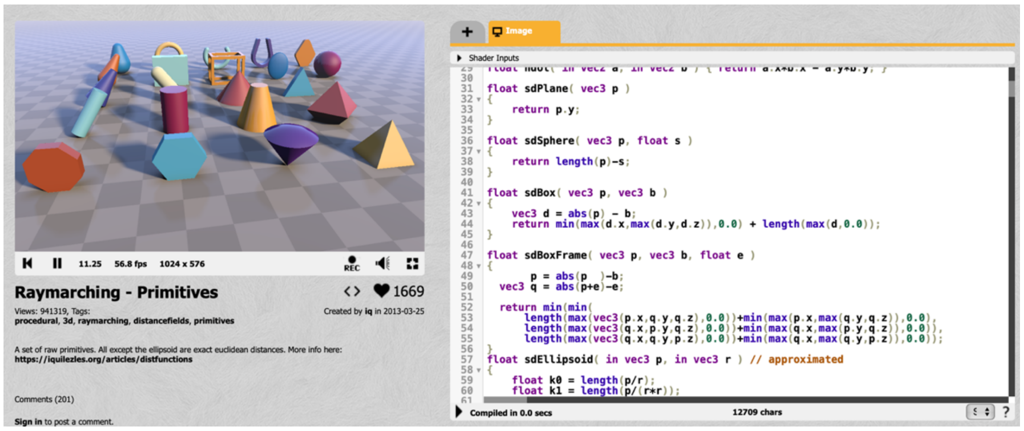
This project focuses on designing and implementing an interactive shader editor that can be used by developers and artists. The goal is to present a tool that supports users in creating, compiling, and manipulating shaders, with the added benefit of real-time visual feedback. The following sections will detail the design and implementation of this editor, illustrating how it enhances the shader development process and supports a collaborative workflow.

Related Work

Shader development tools have advanced significantly over time. This section introduces a couple of work previously done related to this field, with a focus on interactive shader tools and their impact on shader development

Shadertoy

Shadertoy (2013) is an innovative platform that has had a significant impact on interactive shader development. Shadertoy provides an online environment where users can experiment with fragment shaders in real-time. The platform supports GLSL (OpenGL Shading Language) and allows for immediate visual feedback, which is essential for iterative shader design and refinement (Jeremias and Quilez 2014). Shadertoy's primary advantage is its real-time rendering capability. Users can observe changes to shader code instantly, facilitating rapid experimentation and optimization of visual effects. It also promotes an active community by allowing users to share their shaders and discover those developed by others. This collaborative feature increases learning and drives innovation within the shader development community. (Jeremias and Quilez 2014).



Raymarching - Primitives 1669

Views: 941319, Tags:
procedural, 3d, raymarching, distancefields, primitives

Created by **iq** in 2013-03-25

A set of raw primitives. All except the ellipsoid are exact euclidean distances. More info here:
https://iquilezles.org/articles/distfunctions

Comments (201)

Sign in to post a comment.

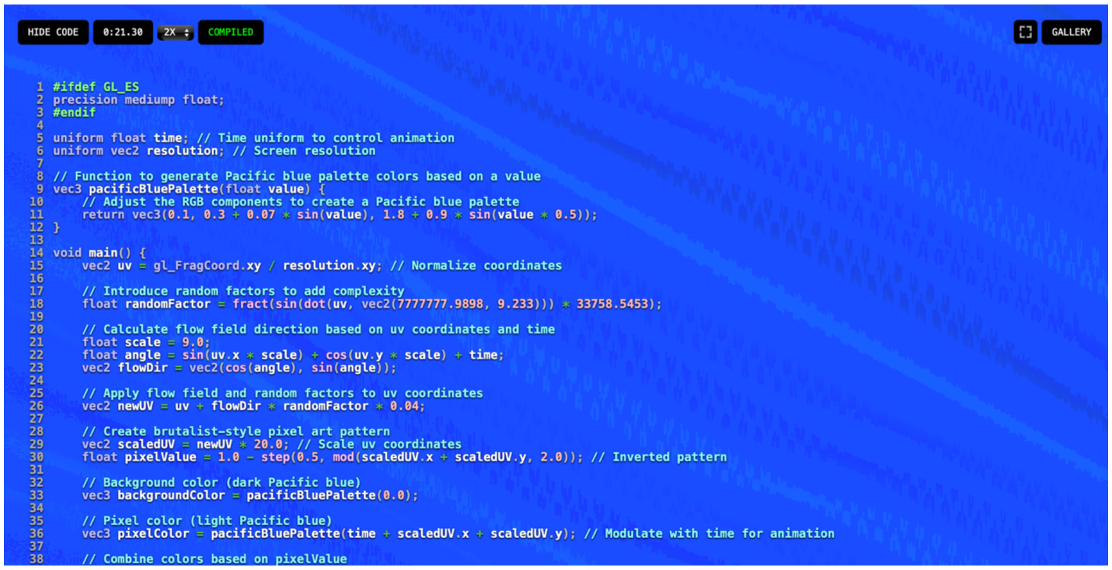
```
float sdPlane( vec3 p )
{
    return p.y;
}

float sdSphere( vec3 p, float s )
{
    return length(p)-s;
}

float sdBox( vec3 p, vec3 b )
{
    vec3 d = abs(p) - b;
    return min(max(d.x,max(d.y,d.z)),0.0) + length(max(d,0.0));
}

float sdBoxFrame( vec3 p, vec3 b, float e )
{
       p = abs(p  )-b;
  vec3 q = abs(p+e)-e;

  return min(min(
      length(max(vec3(p.x,q.y,q.z),0.0))+min(max(p.x,max(q.y,q.z)),0.0),
      length(max(vec3(q.x,p.y,q.z),0.0))+min(max(q.x,max(p.y,q.z)),0.0)),
      length(max(vec3(q.x,q.y,p.z),0.0))+min(max(q.x,max(q.y,p.z)),0.0));
}
float sdEllipsoid( in vec3 p, in vec3 r ) // approximated
{
    float k0 = length(p/r);
    float k1 = length(p/(r*r));
```

Compiled in 0.0 secs   12709 chars

Figure 1: "Raymarching - Primitives" (Shadertoy 2013)

GLSL Sandbox

GLSL Sandbox's basic online platform for GLSL shaders provides a complementary way to interactive shader creation. This tool prioritizes usability and accessibility, making it a valuable resource for both new and experienced shader developers (Jensen et al. 2007). GLSL Sandbox enables users to preview shader results in real-time and adjust shader parameters dynamically. The simple interface facilitates instant feedback, which is essential for fine-tuning shaders and trying out various visual effects. The platform’s community features, including shader sharing and modification, also contribute to a collaborative environment that encourages knowledge exchange and creative exploration (Mannub and Hinkenjann 2014).



```glsl
#ifdef GL_ES
precision mediump float;
#endif

uniform float time; // Time uniform to control animation
uniform vec2 resolution; // Screen resolution

// Function to generate Pacific blue palette colors based on a value
vec3 pacificBluePalette(float value) {
    // Adjust the RGB components to create a Pacific blue palette
    return vec3(0.1, 0.3 + 0.07 * sin(value), 1.8 + 0.9 * sin(value * 0.5));
}

void main() {
    vec2 uv = gl_FragCoord.xy / resolution.xy; // Normalize coordinates

    // Introduce random factors to add complexity
    float randomFactor = fract(sin(dot(uv, vec2(7777777.9898, 9.233))) * 33758.5453);

    // Calculate flow field direction based on uv coordinates and time
    float scale = 9.0;
    float angle = sin(uv.x * scale) + cos(uv.y * scale) + time;
    vec2 flowDir = vec2(cos(angle), sin(angle));

    // Apply flow field and random factors to uv coordinates
    vec2 newUV = uv + flowDir * randomFactor * 0.04;

    // Create brutalist-style pixel art pattern
    vec2 scaledUV = newUV * 20.0; // Scale uv coordinates
    float pixelValue = 1.0 - step(0.5, mod(scaledUV.x + scaledUV.y, 2.0)); // Inverted pattern

    // Background color (dark Pacific blue)
    vec3 backgroundColor = pacificBluePalette(0.0);

    // Pixel color (light Pacific blue)
    vec3 pixelColor = pacificBluePalette(time + scaledUV.x + scaledUV.y); // Modulate with time for animation

    // Combine colors based on pixelValue
```

–

Figure 2: Example of shader created using GLSL Sanbox (GLSL Sandbox).

Python Scripting

The integration of python scripting into shader development represents a significant advancement in the field. As discussed by Mannub and Hinkenjann (2005), python scripts offer a high-level approach to shader management, providing a more flexible and programmable environment for developers.

Using Python for shader development simplifies the creation and manipulation of shaders by automating tasks and customizing tool functionalities. This approach is particularly beneficial for developers familiar with Python, as it allows them to leverage their existing programming skills to enhance shader development workflows. The ability to integrate Python scripting also facilitates the development of custom tools and extensions, further extending the capabilities of shader development environments.

Node-based Editors

Node-based shader editors have greatly impacted shader creation by offering visual interfaces that simply the complexity of traditional shader programming. Tools such as Unreal Engine's Material Editor and Blender's Shader Editor, are great examples as users construct shaders through graphical nodes representing different functions and operations (Jensen et al. 2007). The node-based approach simplifies shader development by providing a visual representation of the shader's data flow, making it more accessible to users with limited programming experience. This method not only enhances usability but also supports collaborative workflows by providing a common platform for artists and developers to work together on shader design.

Technical Background

This section delves into the technical concepts that are fundamental to the design and implementation of the interactive shader editor

Fragment Shader Pipeline

The fragment shader pipeline is a key component of modern graphics programming, influencing how pixels are processed and rendered on the screen. In a typical graphics pipeline, vertices are first transformed from 3D space to screen space through a series of stages including vertex transformation, primitive assembly, and rasterization. The rasterization stage generates fragments, which are potential pixels on the screen, from the primitives (points, lines, or triangles) (Jensen et al. 2007).

The fragment shader operates on these fragments, calculating their final color and attributes based on various inputs like texture maps, lighting conditions, and material properties. This process involves evaluating the shader code, which manipulates the fragment's attributes to produce the desired visual effects. The results are then written to the framebuffer, which represents the final image displayed on the screen.

In the context of the interactive shader editor, leveraging the fragment shader pipeline allows for real-time visual feedback. Users can instantly see the effects of shader modifications, facilitating an iterative development process. This real-time feedback is crucial for refining visual effects and optimizing performance, as it provides immediate insight into how changes impact the final rendered output (Jeremias and Quilez 2014).

Ray Marching and Signed Distance Functions (SDFs)

Ray marching is an advanced technique for rendering complex 3D scenes and effects within shaders, particularly effective when using Signed Distance Functions (SDFs). Ray marching involves casting a ray from the camera into the scene and stepping along the ray’s path to determine intersections with surfaces (Mannub and Hinkenjann 2005). This method is particularly suited for rendering implicit surfaces described by SDFs. Traditional rasterization processes geometry in a scene to determine pixel colors. On the other hand. Ray marching computes the distance from a ray’s origin to the nearest surface in the scene. This method is computationally expensive but enables for highly detailed visual effects that would be difficult to achieve with traditional rendering techniques (Jeremias and Quilez 2014).

An SDF provides the shortest distance from any point in space to the nearest surface. By evaluating the SDF at each step along the ray, the algorithm can determine whether the ray has intersected a surface. If the distance falls below a predefined threshold, the algorithm considers the ray to have hit the surface, allowing for shading calculations. If the ray does not intersect any surface within the maximum number of steps, it is deemed to have missed all surfaces (Jeremias & Quilez 2014). The use of SDFs offers several advantages: they allow for the representation of complex geometries with minimal memory usage and enable smooth blending between shapes. The interactive shader editor integrates ray marching and SDFs, providing users with the ability to create and manipulate sophisticated 3D shaders. This integration supports procedural generation of surfaces and volumes, granting users extensive creative control over their visual outputs.

Lighting

Lighting is one of the most important components of shader programming, as it directly influences both the realism and visual attractiveness of the generated scene. Lighting calculations in real-time shader creation must be accurate and efficient in order to preserve performance while producing the intended visual effects. Lighting in shader development can be grouped into different categories with varying degrees of complexity and realism. Below are examples of lighting generally used to contribute to creating realistic environments in real-time applications.

- Ambient is the fundamental type of light used in shader development. It represents the non-directional light found in an environment. It provides a foundation level of illumination to make sure that the scene is not black. This light is frequently used along with other lighting models to produce more realistic scene renderings.
- Diffuse lighting is essential for simulating how light scatters across rough surfaces. It is based on Lambert's cosine law, which states that the brightness of a surface is directly proportional to the cosine of the angle between the surface normal and the incoming light direction (Pharr et al. 2023). The intensity of diffuse light I_d on a surface is calculated as

  $$I_d = L_d \cdot \max(0, \mathbf{N} \cdot \mathbf{L})$$

  L_d is the light intensity, $\boldsymbol{N}$ is the normal vector at the surface point, and $\boldsymbol{L}$ is the direction vector from the light source to the surface point (Akenine-Möller et al. 2018). The dot product $\boldsymbol{N} \cdot \boldsymbol{L}$ determines the angle at which the light hits the surface, influencing the brightness perceived by the viewer.

- Specular lighting occurs when the light hits an object’s surface and reflects back. This sort of lighting depends on the viewer's position relative to the light source and the surface. It is used in shaders such as metal or glass. The specular reflection is often calculated using models such as the Phong or Blinn-Phong models, which consider both the angle of incidence and the shininess of the surface (Glassner, 2014).

Implementation Overview

This project focuses on implementing a real-time shader editor, integrating both a code editor and a node editor. The node editor is designed for compiling simple 2D shaders using node-based interface, while the code editor supports more complex shader techniques such as ray marching. The goal of this project is to provide flexible environment for real-time shader creation and testing, with an emphasis on ease of use and efficiency. The aim of this project is not on achieving photorealistic rendering but rather on providing a practical, user-friendly platform for developing and testing shaders in real-time. This was implanted using OpenGL, Python, and Qt API.

Functionality

The node editor is a visual interface that allows users to create shaders by connecting various nodes, each representing a specific operation. This editor is ideal for users who may not be comfortable writing GLSL code directly or for quickly prototyping shader effects. The nodes are implemented using the NodeGraphQt library, which provides a flexible framework for creating and managing node-based UIs in Python.

Node Editor

Core Components:

- Material Node: Represents the core material properties, including base color, specular highlights, and shininess. Users can select shading models (Lambert or Phong) and adjust material properties using sliders and color pickers.

- Color Node: Provides a constant color output that can be used as input to other nodes. It includes a color picker widget, allowing users to choose a specific color.
- Blend Node: Allows users to blend two colors using different blending modes such as Multiply, Screen, and Overlay. This node is useful for creating complex color effects.
- Texture Node: Facilitates the inclusion of texture maps into the shader. Users can load textures from files and connect them to other nodes, such as UV coordinates.
- UV Node: Generates UV coordinates, which are essential for texture mapping. The UV Node ensures that textures are properly mapped to surfaces in the shader.
- Gradient Node: Generates a gradient color based on UV coordinates, useful for creating smooth color transitions across a surface.

In the node editor, users right click and choose the desired node into the workspace and connecting them in a correct manner.

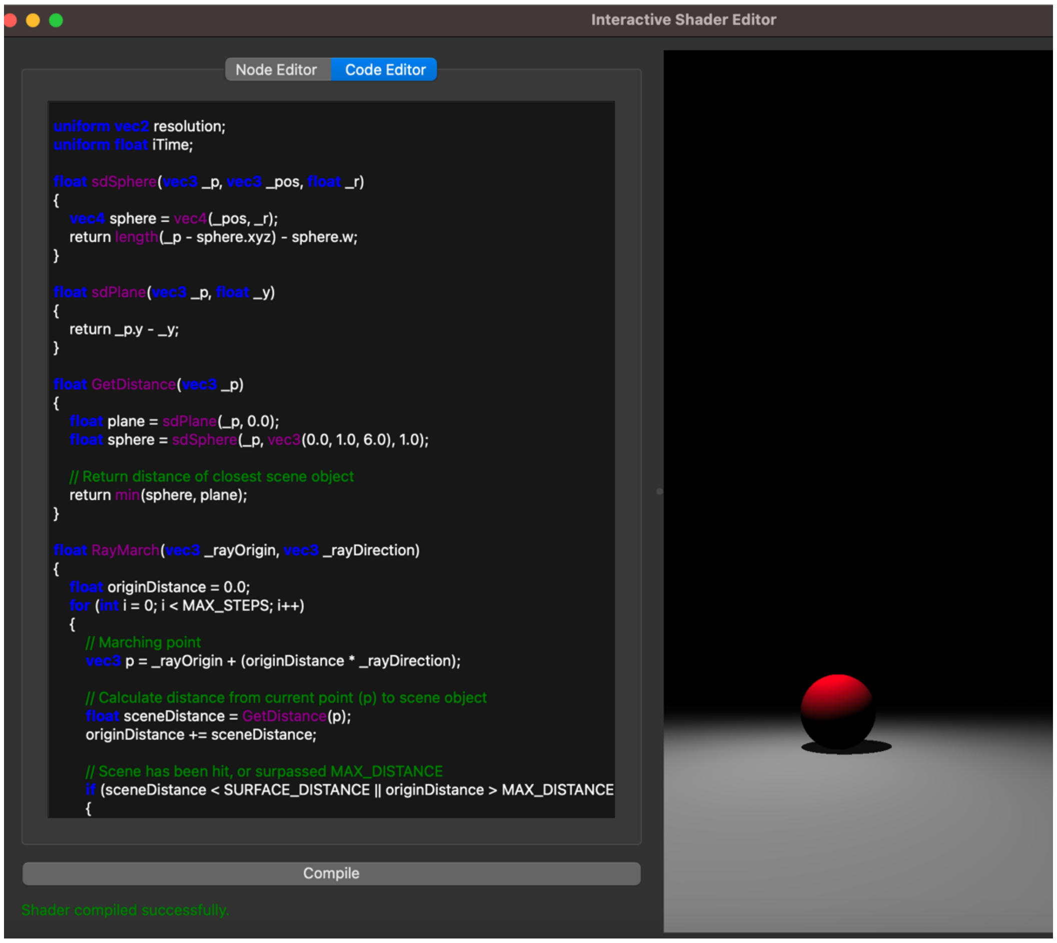
Code Editor

The code editor is designed to allow users with programming skills develop complex 2D and 3D shaders.

Core Features:

- Syntax Highlighter: This component provides syntax highlighting for GLSL code, improving readability and helping users spot errors quickly.
- Error Highlighting: The editor parses error messages from the GLSL compiler and highlights the corresponding lines in the code, allowing users to debug their shaders efficiently.
- Real-Time Compilation: The editor compiles GLSL code in real-time, with any errors or successful compilations immediately reflected in the shader preview.

The code editor is designed for users who are comfortable with GLSL and wish to implement more complex shaders, such as those using ray marching. The editor supports syntax highlighting, error checking, and real-time compilation of GLSL code, enabling users to write and test shaders efficiently. Figure 3 shows the code editor layout along with an example compiled shader.



```
uniform vec2 resolution;
uniform float iTime;

float sdSphere(vec3 _p, vec3 _pos, float _r)
{
   vec4 sphere = vec4(_pos, _r);
   return length(_p - sphere.xyz) - sphere.w;
}

float sdPlane(vec3 _p, float _y)
{
   return _p.y - _y;
}

float GetDistance(vec3 _p)
{
   float plane = sdPlane(_p, 0.0);
   float sphere = sdSphere(_p, vec3(0.0, 1.0, 6.0), 1.0);

   // Return distance of closest scene object
   return min(sphere, plane);
}

float RayMarch(vec3 _rayOrigin, vec3 _rayDirection)
{
   float originDistance = 0.0;
   for (int i = 0; i < MAX_STEPS; i++)
   {
      // Marching point
      vec3 p = _rayOrigin + (originDistance * _rayDirection);

      // Calculate distance from current point (p) to scene object
      float sceneDistance = GetDistance(p);
      originDistance += sceneDistance;

      // Scene has been hit, or surpassed MAX_DISTANCE
      if (sceneDistance < SURFACE_DISTANCE || originDistance > MAX_DISTANCE
      {
```

```
Shader compiled successfully.
```

Figure 3: Code Editor UI with example compiled shader

Ray Marching Integration

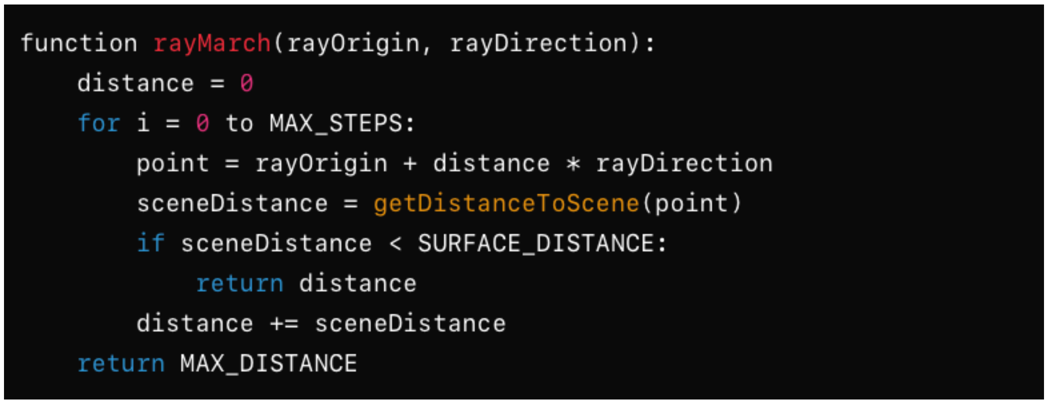
Ray marching is a technique used in the code editor to render complex 3D scenes. It involves simulating the path of light as it interacts with objects in a scene, calculating distances to surfaces using signed distance functions (SDFs).

Key Components of the Ray Marching Implementation:

- Signed Distance Functions (SDFs): These functions calculate the distance from a point in space to the nearest surface, which is essential for determining the point of intersection along a ray's path.

- Ray Marching Loop: The core algorithm iteratively advances a ray through the scene, checking distances at each step until it intersects a surface or exceeds a maximum distance.

Pseudocode for Ray Marching:



```
function rayMarch(rayOrigin, rayDirection):
    distance = 0
    for i = 0 to MAX_STEPS:
        point = rayOrigin + distance * rayDirection
        sceneDistance = getDistanceToScene(point)
        if sceneDistance < SURFACE_DISTANCE:
            return distance
        distance += sceneDistance
    return MAX_DISTANCE
```

Figure 4: Ray Marching Pseudocode

Challenges Encountered

One of the most significant challenges encountered during the development of this project was the inherent limitations of the node editor, which is designed to compile only 2D shaders. The node editor was intentionally simplified to focus solely on 2D shaders. This decision was made as it was a difficult process to incorporate nodes to compile a 3D render. Users are limited to working with 2D surfaces, which can be a significant drawback when attempting to create more complex or three-dimensional visual effects. The lack of support for 3D shaders in the node editor means that users who wish to develop shaders with depth, such as those used in ray marching or 3D texture mapping, must switch to the code editor, which can disrupt workflow continuity. Adding support for 3D shaders, for example, would involve not only extending the types of nodes available but also reworking the underlying GLSL code generation and real-time preview systems to handle the increased complexity. This was challenge in the initial design choices, where the decision to focus on 2D shaders has long-term implications for the system's scalability.

Another challenge was ensuring seamless integration between the node editor and the code editor. Since the node editor is restricted to 2D shaders, users who need to transition to more complex 3D shaders must use the code editor. However, this transition is not always smooth. The difference in capabilities between the two editors can lead to inconsistencies in the shader development process. For instance, a shader developed in the node editor might need to be entirely rewritten when switching to the code editor for 3D effects, resulting in a duplication of effort.

Efficiency and Effectiveness

The node editor allows for experimentation and compiling of 2D shader. The code editor, on the other hand, provides direct control over shader logic, making it suitable for advanced users who require fine-grained control over both 2D or 3D shaders.

Strengths

Real-Time Feedback is one of the main strengths this tool. The ability to see changes reflected immediately in the preview window greatly enhances the development process, allowing for quick iteration and refinement of shaders. Moreover, the use of a flexible node-based framework means that new features, such as additional shading models or texture handling, can be integrated with minimal disruption to existing functionality. Specific to the code editor, it supports both 2D and 3D shaders. This versatility makes it a powerful tool for users who need to develop shaders for a wide range of applications, from simple 2D effects to complex 3D scenes. The ability to handle 3D shaders is particularly important for developers working in game development, simulations, or any field requiring realistic lighting and shading.

The node editor is very simple and straightforward to use. It does not have many buttons or is not crowded. The real time preview is also present allowing the user to see the preview node by node as they make their changes.

Limitations

The node editor was simplified to focus solely on 2D shaders. This inherently restricts the types of shaders that can be created. Users are limited to working with 2D surfaces, which can be a significant drawback when attempting to create more complex or three-dimensional visual effects. The lack of support for 3D shaders in the node editor means that users who wish to develop shaders with depth, such as those used in ray marching or 3D texture mapping, must switch to the code editor, which can disrupt workflow continuity.

Given that shader development often involves working with 3D effects and complex lighting models, users may find the node editor's 2D limitation to be a significant constraint. Managing user expectations and providing clear guidance on when to use the node editor versus the code editor was a necessary part of the project's documentation and user interface design.

Another challenge was ensuring seamless integration between the node editor and the code editor. Since the node editor is restricted to 2D shaders, users who need to transition to more complex 3D shaders must use the code editor. However, this transition is not always smooth. The difference in capabilities between the two editors can lead to inconsistencies in the shader development process. For instance, a shader developed in the node editor might need to be entirely rewritten when switching to the code editor for 3D effects, resulting in a duplication of effort.

GLSL is a strict language with a high sensitivity to syntax and type errors. The code editor's real-time feedback can help catch these errors quickly, but resolving them can still be challenging, especially in complex shaders. Unlike traditional programming environments that offer advanced debugging tools, the code editor is limited in its ability to provide detailed, step-by-step debugging support. This can make it difficult to identify and fix issues, particularly when dealing with intricate shader logic or when errors are buried deep within the code.

Conclusion

The primary objective of this project was to create an interactive shader editor that by creating a tool that is accessible to both artists and developers. This goal was achieved through the implementation of a hybrid interface that combines node-based graph along with text-based code editor. The real-time preview functionality successfully enables immediate visual feedback, a critical feature that supports and helps users to visualize the changes made to the shaders. There are several areas where the project could be improved. Firstly, the current node editor is restricted to a 2D interface, which does not effectively represent the complexities and spatial relationships of shaders that operate in a 3D space. This can limit users with no programing experience to compile 3D shaders using this tool. Secondly, the editor handles basic shaders well, but performance can degrade with very complex shaders or when running on lower-end hardware. Optimizing the rendering pipeline could help improve the overall usage.

Currently, the editor supports a range of shader functions but is still very limited in terms of integrating with external APIs and advanced rendering techniques such as ray tracing or volumetric rendering. Improving the tool is essential to be able to have full functionality. Further developments to the node editor need to be made. The node editor needs to be developed to allow the creation of 3D shaders. Moreover, more nodes need to be implemented so users can compile more complex 2D and 3D shaders.

Some additional suggestions for future development include:

- Advanced Shader Libraries and Templates: Providing a library of pre-made shaders and templates can help users quickly start projects and learn from existing examples, which is especially beneficial for those new to shader programming.
- Integration with More Rendering Engines: Currently optimized for OpenGL, expanding support to include other rendering engines like Vulkan, DirectX, or even game engines such as Unreal Engine and Unity would make the tool more versatile and appealing to a broader audience.

References

Akenine-Moller, T., Haines, E. and Hoffman, N., 2018. *Real-time rendering, fourth edition*. 4th ed. London: CRC Press.

Anon., 2024. *GLSL sandbox* [online]. Glslsandbox.com. Available from: https://glslsandbox.com/e [Accessed 15 June 2024].

Beautypi, 2024. *Shadertoy BETA* [online]. Shadertoy.com. Available from: https://www.shadertoy.com [Accessed 15 June 2024].

Glassner, A. S., 2014. *Principles of digital image synthesis*. Morgan Kaufmann.

Jensen, P. D. E., Francis, N., Larsen, B. D. and Christensen, N. J., 2007. Interactive shader development. *In*: *Proceedings of the 2007 ACM SIGGRAPH symposium on Video games*. New York, NY, USA: ACM.

Jeremias, P. and Quilez, I., 2014. Shadertoy: Learn to create everything in a fragment shader. *In*: *SIGGRAPH Asia 2014 Courses*. New York, NY, USA: ACM.

Kajiya, J. T., 1986. The rendering equation. *Computer graphics* [online], 20 (4), 143–150. Available from: http://dx.doi.org/10.1145/15886.15902.

Mannuß, F., 2005. *Interactive Shader Development Using Python Scripts* [online]. Psu.edu. Available from: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=816db3b70345dec2e93003ac052d36b8130d6a0a [Accessed 10 July 2024].

Pharr, M. and Jakob, W., 2023. *Physically Based Rendering, fourth edition: From Theory to Implementation*. London: MIT Press.