ProcSDF : An Extensible Node-Based Raymarching Playground

Abstract

Raymarching is a powerful and versatile rendering algorithm that uses Signed Distance Fields (SDFs) to create highly detailed and visually stunning 3D scenes. As raymarching gains popularity in artistic and research endeavors, we present ProcSDF, an intuitive and extensible node-based raymarching playground that provides an interactive workflow to experiment with this technique.

This report provides a comprehensive overview of ProcSDF, discussing its features, design, and implementation details. Our software offers a node-based workflow that enables users to create complex and unique 3D scenes by modifying and combining various operations and objects. Additionally, ProcSDF is highly extensible, allowing users to create and integrate custom nodes and materials for increased creative possibilities.

To showcase the power and efficiency of ProcSDF, we present example renders and their corresponding render times. These examples demonstrate the ability of ProcSDF to generate a wide range of organic shapes and complex structures with minimal effort and maximum flexibility.

Looking forward, we anticipate that ProcSDF will continue to evolve and expand with new features and capabilities, further establishing its place as an essential tool for artists and researchers in the field of raymarching. Our software will enable researchers to develop and test new algorithms for SDFs and inspire artists to create innovative and visually striking 3D scenes.

Contents

Bibliography

Chapter 1

Introduction

Computer graphics is a vast field that involves creating, manipulating, and representing visual content using computers. It encompasses various sub-fields, including 3D modeling, animation, simulation, visualization, and rendering. Computer graphics' primary goal is to produce realistic (or stylistic), accurate, and aesthetically pleasing images. [1] [2]

Rendering is one of the fundamental areas of computer graphics that focuses on generating images or animations from 3D models using various techniques. The rendering process involves simulating the behavior of light and materials in a scene to produce a photo-realistic or stylized image. Several rendering techniques are used in computer graphics, including rasterization, ray tracing, and raymarching. Rasterization is the most common rendering technique in real-time applications, such as video games and virtual reality. It involves projecting the 3D geometry of a scene onto a 2D plane and then filling in the pixels of the resulting image using a shading algorithm. While rasterization is fast and efficient, it has several limitations, including the inability to accurately handle complex geometry, shadows, and reflections. Rasterization thus relies on approximations of realistic phenomena to generate shadows and similar entities. Ray tracing is a more advanced rendering technique that simulates the path of light rays as they interact with objects in a scene. It traces the path of rays from the camera into the scene and then calculates the color and intensity of each pixel by tracing the rays back to the light source. Ray tracing can produce highly realistic images, including accurate shadows, reflections, and refractions. [3] However, it is computationally expensive and requires specialized hardware or software optimizations for real-time performance. [4]

Raymarching is a specialized rendering algorithm that uses Signed Distance Fields (functions that represent an object by returning the distance to its closest surface from any point in the 3D coordinate space). We can build upon simple primitives (whose distance functions are easily available) to combine them and create much more complicated shape. [5]

1.1 Objective

The objective of this project was to build and introduce ProcSDF, a Graphical User Interface (GUI) solution designed to enable users to experiment with raymarching without the need for coding. The software uses a node-based approach to make the modeling process more intuitive, allowing artists, enthusiasts, and researchers to experiment with raymarching and SDFs more easily. ProcSDF is an accessible tool that allows users to create complex geometries and high-quality renders with ease.

1.2 Motivation

The current landscape of raymarching tools includes WOMP [6] and SDFPerf [7]. Of these, WOMP [6], which was released in 2022, appears to be the most relevant to our objectives. It leverages SDF-based raymarching to enable users to manipulate 3D scenes via a web-based stack-like user interface. On the other hand, SDFPerf [7] is similar to our proposed approach, but it has not been updated since 2020 and only renders normal maps.

However, the process of setting up raymarching typically requires a considerable amount of code, and graphical user interface (GUI) solutions are scarce. WOMP [6] has addressed this limitation by adopting a stack-based approach. Nevertheless, this reliance on coding can be a significant obstacle for artists and enthusiasts who lack the requisite technical skills. To overcome this challenge, we aim to create a user-friendly platform that provides accessibility to individuals interested in creating models using raymarching. Additionally, we seek to offer researchers a seamless and intuitive environment to explore signed distance functions (SDFs).

1.3 Problem Statement

ProcSDF aims to address the limitations of existing tools for creating Signed Distance Function (SDF) content. The current tools lack an intuitive GUI for interactively designing and visualizing SDFs. Existing raymarchers, such as ShaderToy, do provide a real-time preview of the SDF as it is being created, but they still require the user to write the SDF code by hand, which can be a significant hurdle for beginners. ProcSDF aims to solve these problems by providing an intuitive, user-friendly GUI for creating SDFs. The software will allow users to create SDFs by manipulating a node graph, where each node represents a mathematical operation that can be performed on the input parameters. The user can then visualize the SDF in real-time using a built-in raymarcher. The software automatically generates the fragment shader code corresponding to the node graph, so that users don't have to write the code manually. In addition,

ProcSDF is designed with extensibility in mind, allowing users to easily add custom nodes or modify existing ones. The software is also optimized for performance, with a focus on minimizing the number of operations required to compute the SDF.
ProcSDF differs from existing softwares because:

1. This software's graphical user interface (GUI) is based on a node system, in which the setup of 3D primitives and execution of various operations is carried out through a node graph. Each node within the graph represents a primitive, such as spheres or cylinders, or an operation like a union or intersection. Ultimately, the object node consolidates the individual nodes into a cohesive 3D object. This procedural approach to content generation empowers users to create complex geometries without requiring any programming expertise.

2. ProcSDF's comprehensive feature set and intuitive interface make it an accessible tool for artists, designers, and researchers interested in experimenting with raymarching and Signed Distance Functions (SDFs). It allows users to export high-quality renders that include materials and lighting settings, with the freedom to customize the aspect ratio and other render parameters. This end-to-end solution empowers users to model intricate 3D scenes and effortlessly generate and export them into highly-detailed renders. ProcSDF's export functionality is particularly useful for artists and designers who need to create professional-looking renders for presentations or portfolio pieces.

3. ProcSDF's flexibility extends beyond its basic feature set, as users can augment its functionality by writing their own nodes and materials. This feature empowers users to tailor the software to their specific needs and achieve results that may not have been possible with the default functionalities alone. By enabling users to create custom nodes and materials, ProcSDF opens up a world of possibilities for those seeking to push the boundaries of 3D rendering. This feature is particularly useful for researchers seeking to test and experiment with novel SDF-based rendering techniques.

The challenges involved in creating ProcSDF include designing an intuitive GUI that makes interacting with the raymarcher fun and easy, developing algorithms for generating fragment shader code from the node graph, ensuring the software is extensible, and optimizing performance to minimize the computational overhead. To conclude, the contributions of ProcSDF and this project could be listed as follows :

1. ProcSDF offers a no-code, intuitive interface as a raymarching playground for artists and enthusiasts to explore and experiment with. This tool provides an excellent platform for

users to iterate and refine their ideas related to raymarching, which is becoming increasingly popular in generative art. Introducing this no-code tool is poised to expand the boundaries of raymarching-related art and further facilitate experimentation and creative expression in this field.

2. ProcSDF offers an intuitive environment for researchers to test and experiment with their Signed Distance Functions (SDFs), thanks to its extensible features, Custom Nodes, and Custom Materials. These features empower users to write their own nodes and materials, enabling the import of SDFs created using other processes into ProcSDF. The imported SDFs can then be incorporated into 3D scenes within ProcSDF, allowing researchers to inspect their visual representation and evaluate their effectiveness. Further details on the Custom Nodes and Custom Materials features can be found in sections 3.1.3 and 3.1.3.

3. ProcSDF's implementation details provide a valuable example workflow for converting node graphs into shader code, which can benefit those looking to build similar tools in the future. The implementation process tackles and overcomes challenges such as proper application of transforms, evident in the code and discussed in detail in the following sections. The insights gained from ProcSDF's implementation can serve as a useful reference for developers looking to create comparable tools and overcome similar challenges.

The code for ProcSDF is open-sourced with the MIT license and can be obtained on our GitHub repository hosted at https://github.com/angad-k/ProcSDF.

Chapter 2

Literature Survey

ProcSDF can be better appreciated with some preliminary understanding of raymarching and SDFs - the primary building blocks around which it is built. In this section, we describe both of them and then move on to the related work done in this field.

2.1 Signed Distance Fields (SDFs)

To accurately render objects in a scene using the raymarching algorithm, we first need to describe these objects using mathematical functions. One such function that is commonly used in raymarching is the Signed Distance Function (SDF) [8]. An SDF is a simple mathematical function that represents the surface of a 3D object in the scene. An SDF takes in three parameters *(x, y, z)* and gives out a float value which is the shortest distance between the point specified by these parameters and the surface of the 3D object. The SDF can also indicate whether the point is inside or outside of the object's surface by returning a negative or positive distance value, respectively. SDFs are highly versatile and can be derived for a variety of primitives such as spheres, cones, boxes, cylinders, torus, and more. **Inigo Quilez's resources** on SDFs give a number of examples for the same [9]. By combining SDFs for different primitives, complex shapes and structures can be created. Moreover, SDFs can be easily modified to add deformations, perturbations, or other visual effects to the object's surface. Understanding SDFs is crucial for implementing the raymarching algorithm effectively. In the next section, we will describe how the raymarching algorithm uses SDFs to render objects in a scene.

Additionally, the importance of SDFs extends beyond raymarching algorithms. They have also found applications in fields such as computer graphics, computer-aided design, and robotics, due to their ability to accurately represent and manipulate 3D shapes in a simple and intuitive manner. Therefore, understanding SDFs is not only valuable in the context of our specific ray-

marching software, but also has broader applications in computer science and engineering. **Jasm Cole's blog titled "Signed Distance Fields"** explores them in detail [10].

2.2 Raymarching

The concept of raymarching builds upon Signed Distance Functions (SDFs) to create an algorithm that enables a range of interesting 3D rendering capabilities with only a modest increase in computation. In a 3D world where SDFs represent all surfaces, it becomes possible to calculate the closest surface to any given point in 3D space and the corresponding distance. This approach provides a powerful framework for exploring and creating complex 3D scenes, enabling users to model and render detailed and intricate shapes and forms. To implement raymarching, we first send out rays from the camera or eye in various directions, depending on the desired view. As these rays move through the 3D scene, we calculate their distance from every object in the scene using the SDFs representing each object. Among these distances, we select the smallest, which indicates the distance the ray can travel in any direction without hitting any object. We then move the ray forward in its direction for that distance and repeat the process until we find an intersection. Detecting intersections is easy with SDFs, as we can define an intersection as occurring when the distance to the closest surface is less than a threshold. This simple yet powerful approach allows us to render complex 3D scenes remarkably efficiently, making it an increasingly popular tool in computer graphics and game development. In summary, raymarching involves sending rays through a 3D scene, calculating their distance from each object in the scene using SDFs, and moving them forward until an intersection is detected, all while relying on the mathematical simplicity and flexibility of SDFs. **Michael Walczyk's blog titled "Raymarching"** explains it in more detail. [11].

The algorithm is visually represented in figure 2.1, the circular spots on the camera's ray being the points to which it was marched in each iteration. One can clearly see where the marching in Raymarching comes from.

Now, coming to the benefits of using raymarching over traditional rasterized or even raytraced workflows. Since all objects are defined by an SDF that returns the distance to their closest surface, we can combine multiple SDFs using various mathematical operations to achieve several results(e.g., Union, Subtraction, Soft Union, etc.) - this is easily achieved using raymarching. At the same time, the same becomes non-trivial in the normal workflows where objects are represented by vertices, edges, and their resulting faces. Since SDFs mathematically abstract away all the topology details, we can use such mathematical operations to achieve various useful outcomes in raymarching.

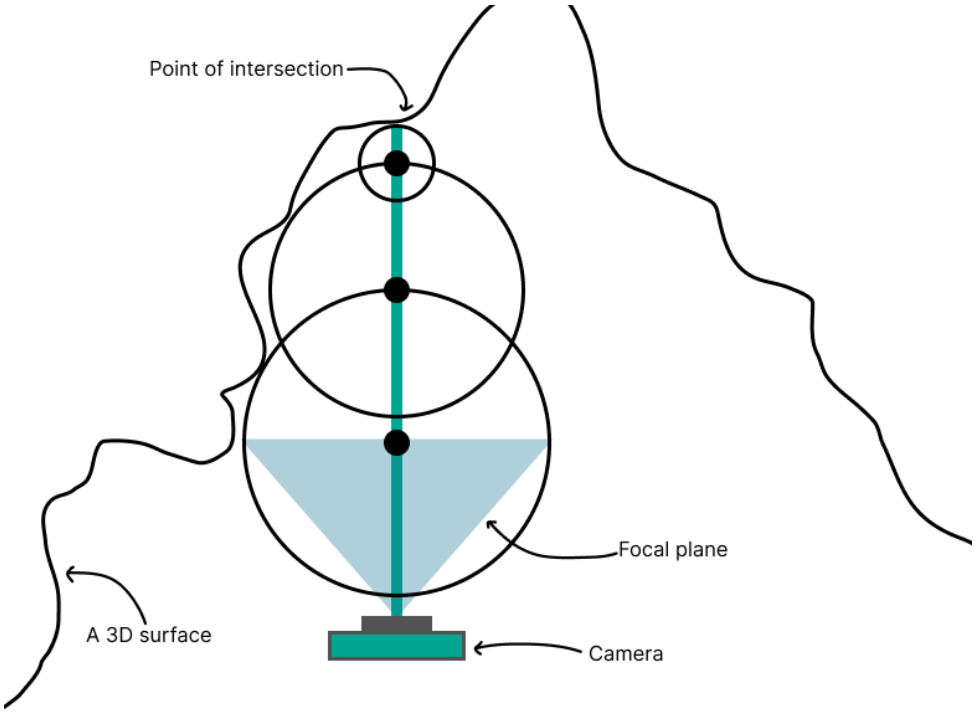


Figure 2.1: A diagram explaining the raymarching algorithm iterations

2.3 Related work

The research paper **Interactive modeling of implicit surfaces using a direct visualization approach with signed distance functions** [12] presents a novel approach for modeling 3D surfaces using signed distance functions (SDFs). Implicit surfaces are mathematical representations of surfaces in 3D space, defined by a function that returns a signed distance value for any given point in space. SDFs are a type of implicit surface representation that contains information about the shape and the sign of the distance value. The paper describes an approach to create and manipulate implicit surfaces in real-time using a graphical user interface. The basis for 3D scene creation using SDFs can be found in this research paper.

The applications of raymarching can be seen in papers like **Interactive Indirect Illumination Using Mipmap-based Ray Marching and Local Means Replaced Denoising** [13] and **Many-Lights Real Time Global Illumination Using Sparse Voxel Octree** [14] where raymarching is used in the process for generating global illumination in a 3D scene. **ClonAR: Rapid redesign of real-world objects** [15] uses raymarching to create a Virtual Reality (VR) based workflow for recreating and editing real-world objects in 3D.

While we got the theoretical foundations of SDFs and raymarching[16] from the above papers, the node-based approach that we use in ProcSDF can be found in the research paper **Designing Gratin: A GPU-Tailored Node-Based System**[17] which proposes a new node-based sys-

tem for graphical processing units (GPUs). Node-based systems are used for data processing and computational workflows, allowing users to create complex data processing pipelines by connecting individual nodes that perform specific operations. The problem that Gratin tries to solve is the inefficient utilization of parallelism and the high memory bandwidth of GPUs. This research paper also shows that our nodal approach to rendering 3D scenes in ProcSDF is not entirely new and is a common practice. However, it does so for the problem of prototyping graphic pipelines rather than raymarching. Similarly, **Node-Based Shape Grammar Representation and Editing** [18] uses a node-based architecture for procedural content creation. Thus, it can be seen that a similar application in raymarching would prove to be useful too.

Chapter 3

Solution Methodology

3.1 ProcSDF: Features and usage

This section presents an overview of ProcSDF's layout and components and instructions on how to use them. ProcSDF displays its main window when launched, as shown in Figure 3.1. The default scene features a red sphere, which the user can edit or replace. Additionally, users have the option to load previously saved project files.

The inspector, located in the bottom-left part of the window, provides a set of controls that allow users to modify various settings for the 3D scenes, such as camera position, aspect ratio, and render settings. Users can also access the custom nodes and materials created through the inspector. The node workspace is in the right side of the window, and it displays the node graph editor. This is where users can construct the 3D scene by dragging and dropping nodes, connecting them with wires, and configuring their properties. The node workspace allows users to work with procedural modeling and manipulate the geometry of the objects. Finally, the rendered 3D scene is displayed on the top-left side of the window, providing users with a real-time view of their work. The node workspace, inspector, and rendered 3D scene are the three main parts of the ProcSDF window, and further sections delve into each part in more detail.

Node Workspace

The Node Workspace in ProcSDF is where users can build their 3D scene by adding and connecting nodes. The workspace, visible in figure 3.2, allows the user to drag nodes from the inspector and connect them by dragging pins from one node into the inputs of the other. The node graph created in this workspace describes all the objects in the scene. The *Recompile* button is located in the Node Workspace and allows users to trigger shader regeneration and recompilation. Node and material parameters are defined as uniforms, so their changes are reflected in real time.

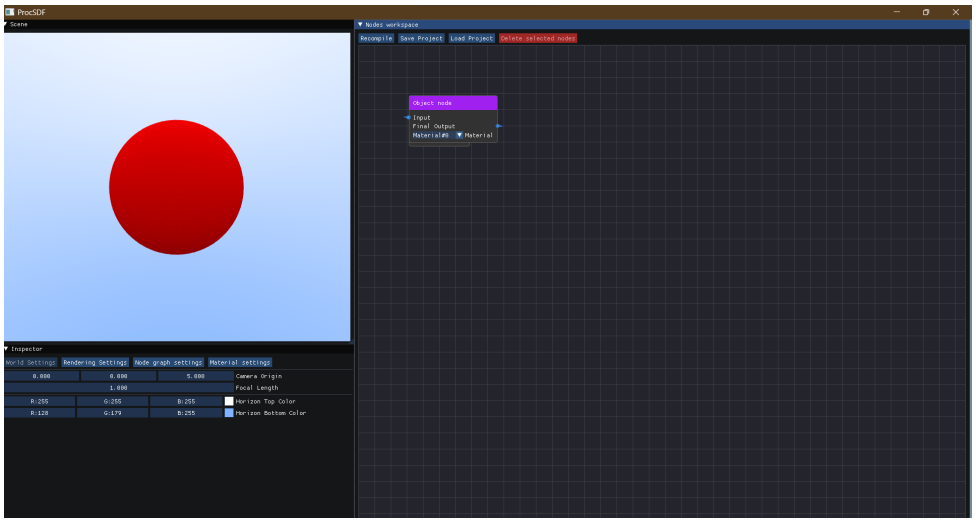


Figure 3.1: The default ProcSDF window

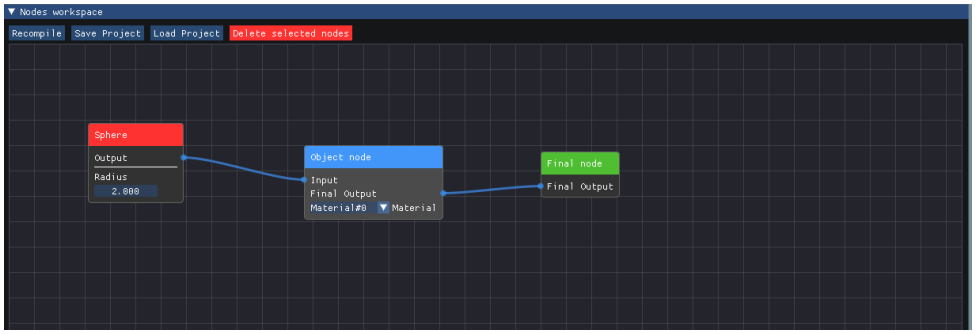


Figure 3.2: Node Workspace

However, changes to the node graph and changes in an object's selected material require regeneration of the shader to reflect in the render. After making such changes, a prompt appears that tells users to recompile the shader to view the effects of their changes. The shader generation algorithm is described in detail in section 3.2.3.

ProcSDF also allows saving and loading projects with the *Save Project* and *Load Project* buttons, respectively. These buttons save and load projects to a *.procsdf* file, a JSON-based description of the project, allowing for easy readability of project files. Nodes can be selected by clicking on them, and the *Delete selected nodes* button is enabled, allowing for the deletion of the selected node. To delete links between nodes, the user has to hold the *Ctrl/Cmd* key on the keyboard while clicking on the particular link.

Inspector

The inspector has several tabs dealing with different parts of the 3D scene described as follows. The World Settings tab, shown in Figure 3.3, contains various properties that affect the 3D world being rendered. The *Camera Origin* and *Focal Length* parameters define the position and

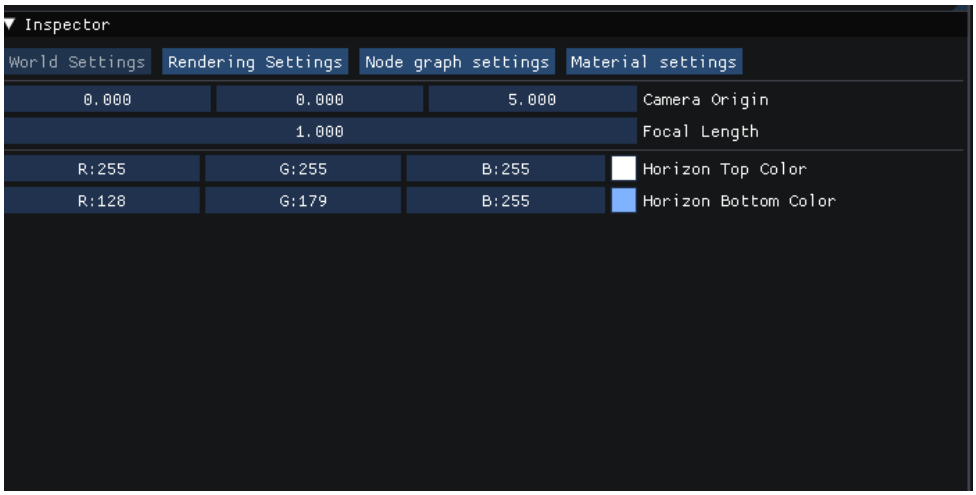


Figure 3.3: The World Settings tab in the inspector

focal length of the camera, respectively. The *Horizon Top Color* and *Horizon Bottom Color* parameters allow users to set the colors visible in the background of the rendered scene. By default, the background is set to a gradient with a blue top and a white bottom. The rendering

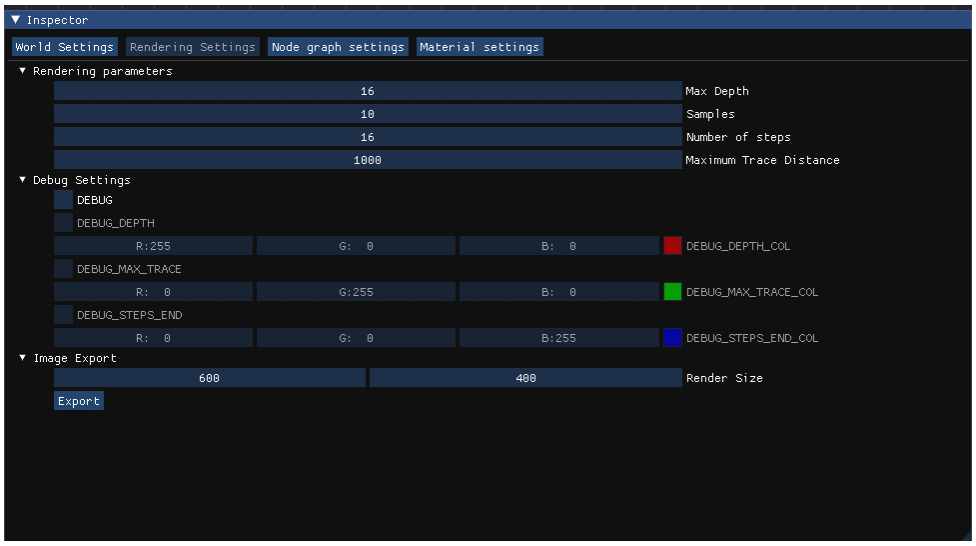


Figure 3.4: The Rendering Settings tab in the inspector

settings tab, as seen in figure 3.4, deals with the actual image rendering flow. The *Rendering parameters* include *Max Depth*, *Samples*, *Number of steps*, *Maximum Trace Distance* - increasing the value of which shall improve the quality of render at the cost of performance. *Max Depth* is the maximum number of times the ray can be scattered from an object before defaulting to a black color. *Samples* is the number of samples we take into consideration for each point on the screen. Fewer samples lead to a noisy render. *Number of steps* defines the maximum number of times a ray shall advance. *Maximum trace distance* defines the maximum distance that a ray shall advance.

Debug settings can be enabled with the *DEBUG* checkbox. This provides users with three debug settings that can be used for debugging custom code. Section 3.1.3 goes over these options in more detail. *Image Export* allows us to export renders into *.png* files. Users can set *Image Size* and click on *Export* to select the file name and location to export the render to. The node graph

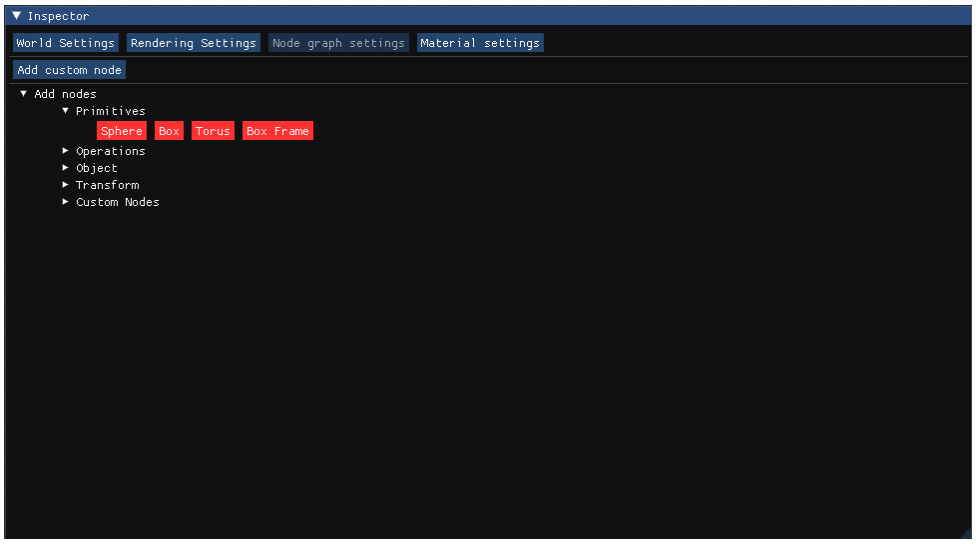


Figure 3.5: The Node Graph Settings tab in the inspector

settings tab, as seen in figure 3.5, allows users to add nodes to the node workspace. The different types of nodes are described and listed in section 3.1.1. The *Add custom node* button allows user to load their own nodes into ProcSDF. The material settings tab, as seen in figure 3.6,

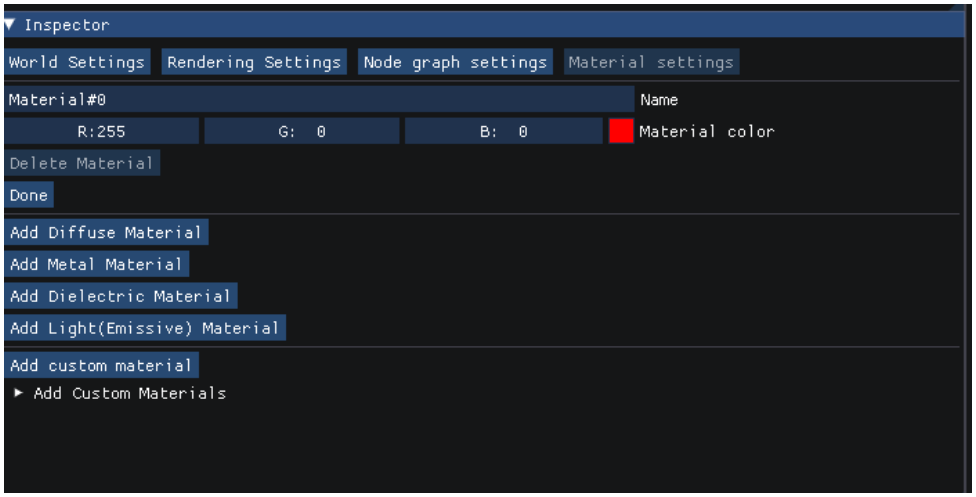


Figure 3.6: The Material Settings tab in the inspector

allows users to add different types of materials and modify their properties. This tab also houses the Custom Material flow. The different types of available materials are listed and described in section 3.1.2.

Viewport

The viewport is the area in the ProcSDF window where the 3D scene is displayed. It provides users with a real-time view of their work as they modify the scene in the node workspace or adjust settings in the inspector. The rendered image in the viewport updates automatically in response to changes made in the scene. Users can rotate and pan the camera by changing the required settings in the inspector, providing a way to inspect the scene from different angles visually. However, unlike the node workspace, there are no interactions possible with the viewport itself. Users cannot modify the scene by clicking or dragging in the viewport. Instead, any changes must be made in the inspector or node workspace.

3.1.1 Nodes

ProcSDF uses a node-based approach to create 3D scenes, where nodes represent different objects and operations in the scene. These nodes are manipulated in the *Nodes Workspace* and can be added from the *Node graph settings* tab located in the *Inspector*. Users can drag and drop nodes into the workspace, where they can be positioned and linked with other nodes using wires. ProcSDF provides different types of nodes, including Primitives, Operations, Object, and Transform Nodes, which can be combined to create complex 3D scenes. Additionally, ProcSDF allows users to create and use custom nodes to enhance their workflow further.

Primitive nodes are the fundamental building blocks of any 3D scene in ProcSDF. These nodes are a starting point for constructing complex shapes and objects within the 3D environment. ProcSDF offers a variety of primitives, such as Sphere, Box, BoxFrames, Torus, etc. These primitives can be easily operated upon and combined with other nodes to create more complex shapes and objects. By providing a set of primitive nodes, ProcSDF simplifies the process of constructing 3D models, allowing users to focus on the creative aspects of their work.

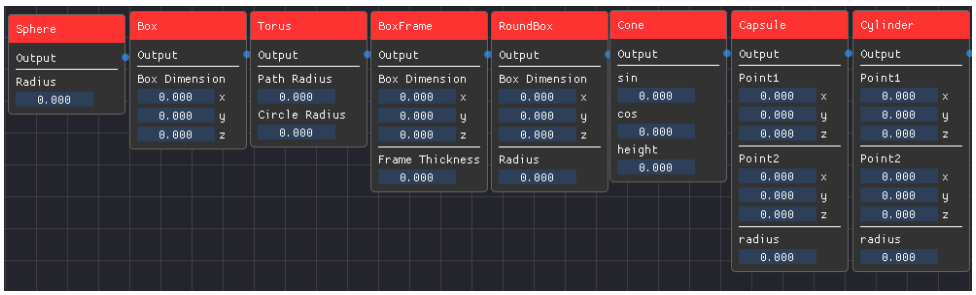


Figure 3.7: Primitive Nodes in the Node Workspace

Operation nodes in ProcSDF take multiple nodes as inputs and perform specific operations on them to achieve a desired effect. For example, the Union node combines two shapes to create

a new shape that includes both shapes, while the Intersection node outputs the part of the shapes that overlap. ProcSDF provides a range of operations that can be applied to the nodes, including Union and Intersection nodes, as well as other operations that can be used to create more complex shapes.

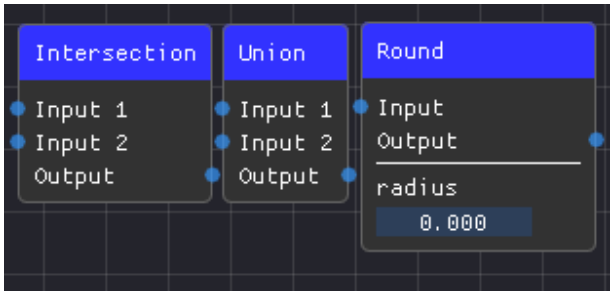


Figure 3.8: Operation nodes in the Node Workspace

ProcSDF offers a range of **Transform nodes** that enable users to modify the spatial coordinates of a model without changing its shape. The *Transform* nodes allow users to translate, rotate and scale a shape as per their requirements. Below are the various types of *Transform* nodes that ProcSDF provides:

- *Translation*: This node has three parameters: *x, y,* and *z*. It can be used to move a shape along the X, Y, and Z directions. The user can simply adjust the *x, y,* and *z* parameters to move the shape accordingly.

- *Rotation around X-Axis*: This node modifies the orientation of a shape with the X direction as its axis of rotation. The shape can be rotated by a specified parameter, which is measured in degrees.

- *Rotation around Y-Axis*: This node changes the orientation of a shape with the Y direction as its axis of rotation. The shape can be rotated by a specified parameter, which is also measured in degrees.

- *Rotation around Z-Axis*: This node changes the orientation of a shape with the Z direction as its axis of rotation. The shape can be rotated by a specified parameter, which is again measured in degrees.

In ProcSDF, **Object nodes** serve as the second-to-last step in the node-based workflow before feeding into the *Final Node*. These nodes are responsible for defining the various 3D objects in the scene, and each object has a dropdown menu that allows users to select the material that will be applied to it. Although the object node may seem to have only semantic value, it becomes functional with the addition of materials. Separating shapes into different objects enables users

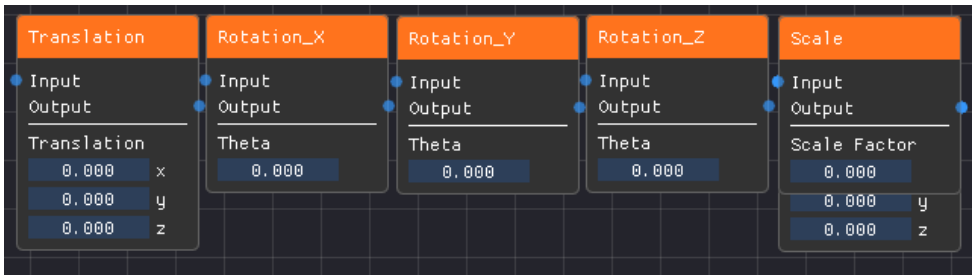


Figure 3.9: Transform nodes in the Node Workspace

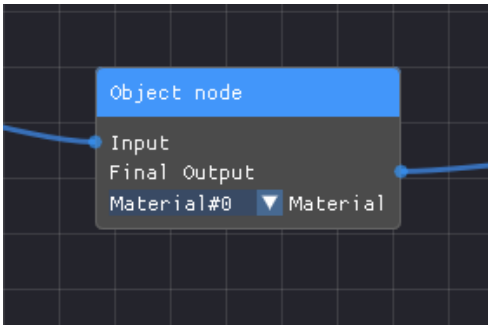


Figure 3.10: Object node in the Node Workspace

to apply different materials to different objects in the scene.

On the other hand, **Custom nodes** are nodes that users can define and use in the tool, allowing for more flexibility in the workflow. Users can create distance functions or operations that can be used as nodes in the tool. Adding custom nodes is simple and involves writing a GLSL function with a few comments that help ProcSDF parse the code. More information on the custom nodes workflow is provided in section 3.1.3.

3.1.2 Materials

Materials are a key component in ProcSDF as they describe how a ray will interact with an object in the scene and the color that the object will contribute to the final image.[19] ProcSDF provides a variety of built-in materials that can be modified to meet the user's needs, such as diffuse, metallic(specular), and dielectric(refractive) materials. Additionally, users can also create their own custom materials using the Custom Materials feature by writing their own GLSL code. The materials in ProcSDF are essentially reusable descriptions of how the ray will behave when scattered by an object, which is determined by the material's properties such as reflectance, transparency, and roughness. The color that an object contributes to the final image is also determined by its material properties, as the material specifies how the surface will interact with light sources and how the scattered light will affect the color and brightness of the object. Overall, materials play a crucial role in determining the appearance of objects in the scene and

can greatly affect the final output of the rendering process.

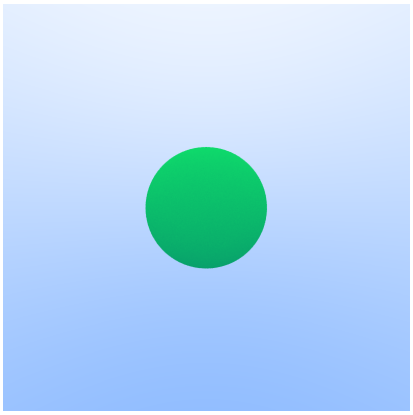


Figure 3.11: Diffuse Material on Sphere

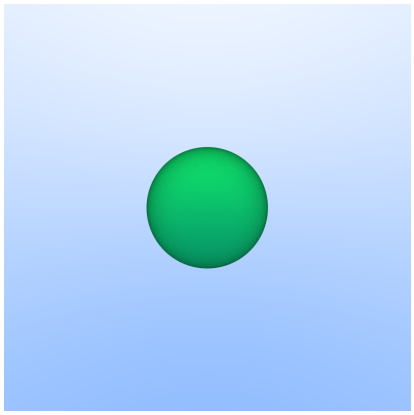


Figure 3.12: Metal Material on Sphere with roughness 0.70

The **diffuse material** is a simple material that scatters incident light rays in random directions, resulting in a rough appearance of the object. It is the most basic material available in ProcSDF. A diffuse material is a type of material that scatters incident light in all directions, making the object appear rough or matte. When light hits a surface with a diffuse material, it gets absorbed and re-emitted in random directions, making the surface appear evenly lit from any angle. This is in contrast to specular materials, which reflect light in a specific direction and create shiny or glossy surfaces. Diffuse material is the most basic type of material and is often used as a starting point for more complex materials. An example of the diffuse material is shown in Figure 3.11.

The **metal material** is used to simulate metallic surfaces in 3D scenes. It reflects the incident ray along the normal, which is the line perpendicular to the object's surface. The roughness property of the metal material determines how much the reflected ray is offset from the perfect reflection direction, giving the material a more or less shiny appearance. A perfect reflection occurs when the roughness is set to 0.0, meaning the reflected ray bounces off the surface at exactly the same angle as the incident ray. However, in reality, most metal surfaces are not perfectly smooth and have imperfections that cause the reflected light to scatter in different directions. By increasing the roughness of the metal material, the user can simulate these imperfections, resulting in a more realistic and believable appearance. The metal material in ProcSDF allows the user to adjust the roughness property to create a range of metallic surfaces, from highly polished to rough and gritty. Figure 3.12 shows an example of a sphere with a metal material applied, where the roughness parameter is set to a non-zero value, giving the sphere a rough, frosted appearance.

The **dielectric material** is a transparent material that can refract and reflect light. When a ray enters a dielectric material from air, the angle of incidence determines the amount of refraction that occurs. The *IOR* property is used to set the Index Of Refraction, which determines how

Figure 3.13: Dielectric Sphere placed before a diffuse sphere

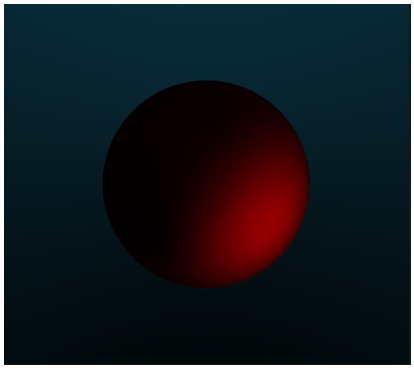


Figure 3.14: A sphere illuminated by a light source

much the ray is bent when it enters the material. The *Roughness* property is used to add imperfections to the surface of the dielectric material, simulating a rough surface that can distort the refracted light. The combination of reflection and refraction of light in the dielectric material produces the characteristic transparent and reflective appearance of glass-like materials.

Emissive materials are unique in the sense that they emit light and remain visible in the scene even without environmental lights. They can be used as a source of light to illuminate other objects in the scene, and they are especially useful for creating light sources in dark environments. Figure 3.14 provides an example of a scene that utilizes emissive materials as light sources.

Other than all the materials described above, users can also define their own materials. The custom material flow is defined in more detail in section 3.1.3.

3.1.3 Extending ProcSDF

As mentioned earlier, ProcSDF is designed around the idea of extensibility, so, ProcSDF offers two features to extend the functionalities in the form of Custom Nodes and Custom Materials. The following sections describe these features in detail.

Custom Nodes

Adding a custom node mainly includes adding one simple function that takes in a number of arguments and returns a *float* value. Since all nodes are basically SDFs, the inputs correspond to the position(in the case of primitive nodes), values of other nodes, or the parameters specific to the node. In contrast, the output corresponds to the result of the SDF that gives the distance of the input position from the closest point on the surface.

To aid the software in parsing the code and creating the nodes as required - the only thing required besides the function are a few minimal comments that can be put anywhere in the file,

listed below.

1. *type <node_type>*: Here *node_type* can be *Primitive*, *Operation* or *Transform* depending on the type of node you want to add.

2. *name <node_name>* : This is used to specify the name, the *node_name* should correspond with the name of the function.

3. *input_pins <... pins>* : All input pins need to be listed here. These will take in other nodes as input. The labels given here shall appear on the resulting node's inputs.

4. *float_params <... params>* : All float params need to be listed here. These will take in parameters on the node itself as input. The labels given here shall appear on the resulting node's inputs.

5. *vec3_params <... params>* : All vec3 params need to be listed here. These will take in parameters on the node itself as input. The labels given here shall appear on the resulting node's inputs.

Using this flow, a number of custom nodes can be added. Figure 3.16 showcases the custom nodes while Figures 3.15, 3.18, 3.17 show the code required to make a Mandelbulb[20], Smooth Union, Octahedron node respectively. The code for *Mandelbulb* was inspired by Pedro Tonini Rosenberg Schneider's shader-fractals repository[21]. The resulting image rendered from the *Mandelbulb* code can be seen in figure 4.1.

```
// type primitive
// name Mandelbulb
// float_params Iterations Bailout Power
float Mandelbulb(in vec3 pos, in float Iterations, in float Bailout, in float Power)
{   vec3 z = pos;
    float dr = 1.0;
    float r = 0.0;
    for (int i = 0; i < Iterations ; i++) {
        r = length(z);
        if (r>Bailout) break;

        float theta = acos(z.z/r);
        float phi = atan(z.y,z.x);
        dr =  pow( r, Power-1.0)*Power*dr + 1.0;

        float zr = pow( r,Power);
        theta = theta*Power;
        phi = phi*Power;

        z = zr*vec3(sin(theta)*cos(phi), sin(phi)*sin(theta), cos(theta));
        z+=pos;
    }
    return 0.5*log(r)*r/dr;
}
```

Figure 3.15: Minimal code to generate the Mandelbulb node

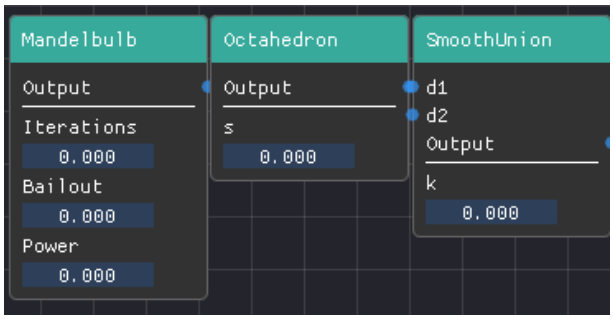


Figure 3.16: Custom nodes in the Node Workspace

```
// type primitive
// name Octahedron
// float_params s
float Octahedron( in vec3 p, in float
{
  p = abs(p);
  return (p.x+p.y+p.z-s)*0.57735027;
}
```

Figure 3.17: Minimal code to generate the Octahedron node

```
// type operation
// name SmoothUnion
// float_params k
// input_pins d1 d2
float SmoothUnion( float d1, float d2, float k ) {
    float h = clamp( 0.5 + 0.5*(d2-d1)/k, 0.0, 1.0
); return mix( d2, d1, h ) - k*h*(1.0-h);
}
```

Figure 3.18: Minimal code to generate the Smooth Union node

Custom Materials

Adding a custom material mainly includes adding one simple scatter function that takes in a number of arguments and returns a *scatter_info* object. The scatter function takes in the position vector, the normal vector, the incident ray, a boolean variable representing whether the ray is incident on the outer side or the inner, and the color followed by user-declared parameters. To aid the software in parsing the code and creating the materials as required - the only thing required other than the function are a few minimal comments that can be put anywhere in the file listed below.

1. *name <material_name>* : This is used to specify the name, the *material_name* should correspond with the name of the function. For example, a material named *Checkerboard* should have a function named *Checkerboard_scatter*.

2. *float_params <... params>* : All float params need to be listed here. The labels given here shall appear on the resulting material's inputs.

3. *vec3_params <... params>* : All vec3 params need to be listed here. The labels given here shall appear on the resulting material's inputs.

```
// name Checkerboard
// float_params divide_by
scatter_info Checkerboard_scatter(in vec3 position, in vec3 normal, in vec3 r_in, in bool is_front_face,
in vec3 color, in float divide_by)
{
  if(divide_by == 0.0)
  {
    divide_by = 0.01;
  }
  int x = int(r_in.x/divide_by);
  int y = int(r_in.y/divide_by);
  vec3 atten = vec3(1.0, 1.0, 1.0);
  if((x%2 == 0) && (y%2 == 0))
  {
    atten = vec3(0.0, 0.0, 0.0);
  }
  vec3 scattered = position + random_in_hemisphere(normal, position);
  //atten = color;
  return scatter_info(scattered, true, atten);
}
```

Figure 3.19: Checkerboard Material code

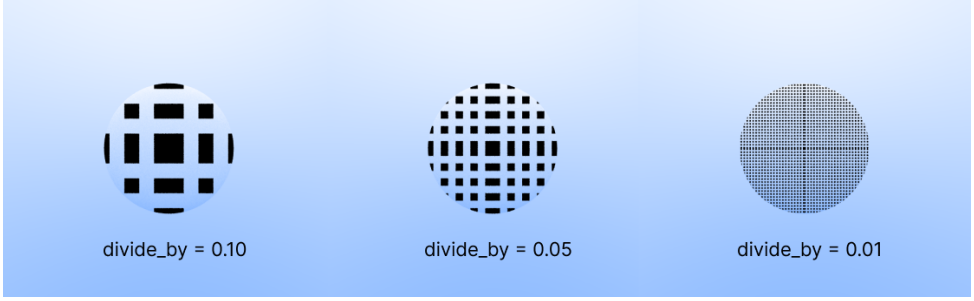


Figure 3.20: Checkerboard Material at various parameter values

Using this flow, a number of custom materials can be added. Figure 3.19 showcases the code while figure 3.20 the resulting materials.

Debugging your custom code

User-written code can add bugs to the flow. There's a chance that the code might not give the required results. Now, graphical debugging doesn't involve the line-based debugging that the coding community is used to, so we have to resort to visual solutions. Other than the debugging solutions that the user might resort to on his own by maybe outputting a specific color or something along that lines, ProcSDF offers a few properties that can be enabled to provide extra information about the rendering that might be useful in debugging. These properties are listed below.

- *DEBUG*: The user needs to enable this property before other debugging properties are enabled.

- *DEBUG_DEPTH*: If the number of scatters exceeds the max depth, the resulting color for that pixel shall be nearly black. However, the user can output a specific color for this case with this property.

- *DEBUG_MAX_TRACE*: If the ray has traveled more distance than the maximum trace distance, the horizon is multiplied by the existing color. However, with this property, the user can output a specific color directly for this case.

- *DEBUG_STEPS_END*: If the steps reach the maximum value, the user can output a specific color directly using this.

3.2 Design and implementation of ProcSDF

ProcSDF is implemented using the C++ programming language with the OpenGL graphics API for rendering. The graphical user interface (GUI) is rendered using the open-source library called "Dear ImGUI." In this section, we shall delve into the technical implementation details of ProcSDF.

3.2.1 Architecture

Figure 3.21 shows the class diagram for ProcSDF. From a high level, ProcSDF's functioning can be divided into three inter-dependent yet distinct classes that tackle three main tasks for ProcSDF - rendering the GUI, generating the shader from the node graph, and rendering the 3D scene that the user has created. To that effect, we have three classes **GUI**, **ShaderGenerator**, and **Renderer**. **NodeGraph** keeps track of all nodes added to the graph and their interconnection and provides an API for its modification. Other than this, **Node** and **Material** are two important abstract classes that are then inherited to create specialized nodes and materials.
The **GUI** class renders the Graphical User Interface (GUI). For this, we have used the "Dear ImGUI" library. The three main GUI components in ProcSDF are the viewport, the inspector, and the node graph editor. For this, the **GUI** class calls the *draw()* function of the **Inspector** and **NodeEditor**, which in turn call the *draw()* functions of their respective components. Such a *draw()* function hierarchy helps simplify and organize the code since any class that may have a GUI representation can define its GUI through a *draw()* call. The viewport is created directly by the **GUI**, and the rendered scene is accessed as a texture directly from the **Renderer**.
The **ShaderGenerator** deals with converting the data from the **NodeGraph** into a fragment

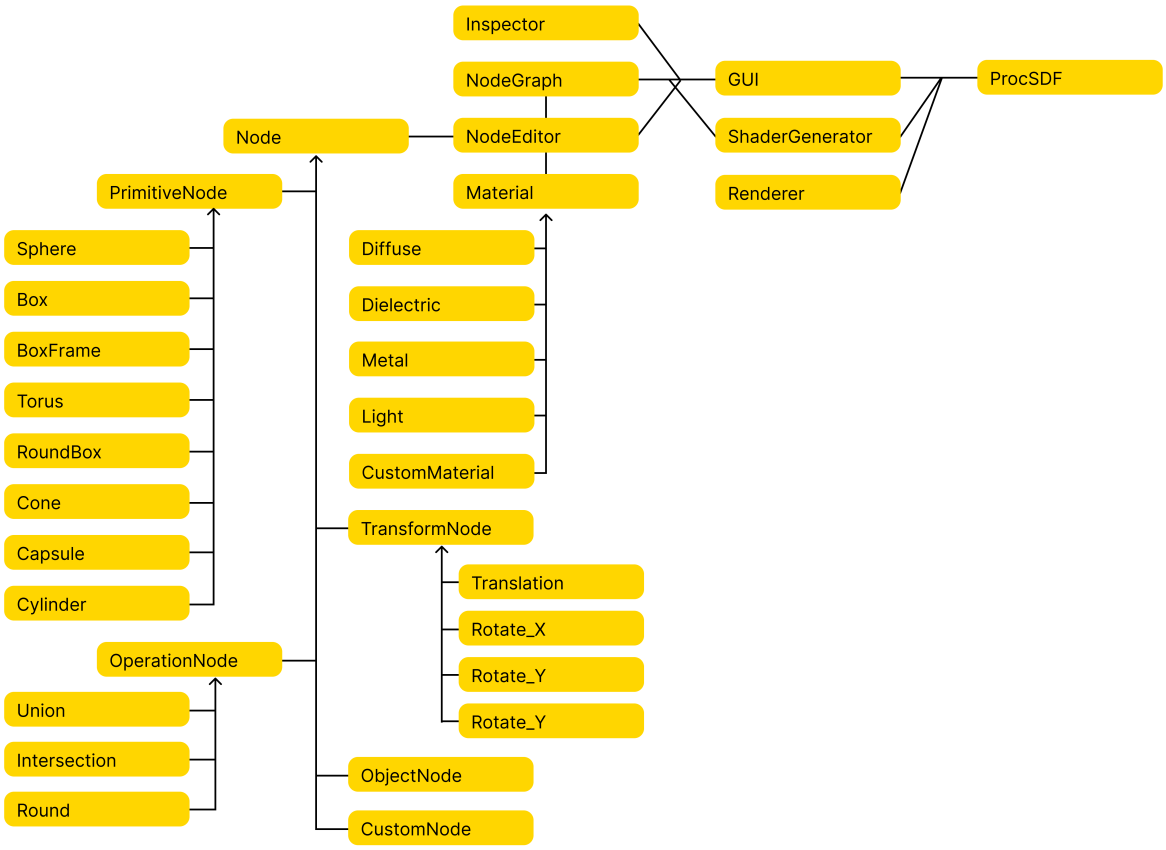


Figure 3.21: Class Diagram for ProcSDF

shader. The exact details of the algorithm involved in this are described in section 3.2.3. The user can ask for the shader to be recompiled from the **NodeEditor**, after which the **ShaderGenerator** gets data from the **NodeGraph**, runs the shader generation algorithm on it, and finally creates a fragment shader that is then used by the **Renderer**.

The **Renderer** sets up the rendering pipeline using OpenGL, renders the 3D scene when called upon, and provides an API for tasks like Image Exporting. It accesses the fragment shader from the **ShaderGenerator**. As explained before, both the hierarchy of the **Node** and **Material** start from abstract classes. All the specific nodes and materials inherit from these abstract classes. Since writing the *draw()* function for each such specific node and material would have been too cumbersome and a considerable impediment to the speed of adding newer nodes and materials to the tool, we made it a point to make the *draw()* and *init()* functions of the top-level, abstract classes generalized enough so that, all that adding a new node or new material involves is inheriting a class from the required base class, defining a few member variables like *Input Pins*, *Output pins*, *Input Labels*, etc. - and finally adding the required GLSL code that shall make these nodes useful. This also helps keep the code for **CustomNode** and **CustomMaterial** clean and organized.

3.2.2 Rendering flow

The rendering pipeline in ProcSDF is designed to be lightweight and efficient. The vertex shader, responsible for transforming vertices, is minimalistic and primarily serves to pass geometry to the fragment shader. In turn, the fragment shader generates the image by raymarching through signed distance functions (SDFs) and shading the surfaces accordingly. The geometry passed to the GPU comprises two triangles that cover the entire screen. The resulting image is then rendered to a texture, which is directly displayed on the GUI. This approach not only allows for fast on-screen rendering but also simplifies the process of exporting images. By resizing the texture to the desired dimensions and issuing a draw call, the scene is rendered at the required resolution. The rendered image is then retrieved from the GPU and saved to a file.

3.2.3 Shader Generation Algorithm

The Shader Generation algorithm is the heart of our raymarching tool. The algorithm generates dynamic GLSL code corresponding to the state of the node graph, which is then passed on to the GPU for image rendering. **Dynamic code generation for realtime shaders** deals with programmatic generation of GLSL code[22].

The algorithm generates a separate function for every non-transform node in the node graph, which is of the form f_{node name}{node id}. The {node name}{node id} part ensures that the function names are unique for each node, and the prefix "f" prevents any conflicts between function names and variable names used. The nodes are processed in a topological order since the nodes that occur later in the topological order build upon earlier ones.

We use an acyclic-directed graph to store the nodes and their connections as described in the node editor[23]. Following is a line-by-line description of the algorithm 1 :

- **Line 2 :** The Shader Generation algorithm takes a list of nodes that have been topologically sorted in a node graph. The nodes are then iterated over in an outer **for** loop, with each node id being processed by the *get_node* function on **line 2** to return the corresponding node object.

- **Lines 4-5 :** The algorithm skips over nodes that are either a transform node or a final node on **lines 4-5**.

- **Lines 7-12 :** Variables such as *function_name*, *function_body*, and *return_variable* are initialized on **lines 7-12**.

- **Line 13 :** The **for** loop on **line 13** iterates for each of the current node's input pins.

Algorithm 1 Algorithm for Shader Generation

```
1: for node_id in TOPOLOGICAL_SORTING do
2:     Node node = get_node(node_id)
3:     index = 0
4:     if (node is transform node or final node) then
5:         continue
6:     end if
7:     function_name = "f_" + node.variable_name()
8:     function_body = ϕ
9:     return_variable = node.variable_name()
10:    if node is an object node then
11:        function_name = "object_" + node.object_id
12:    end if
13:    for ordering_information in node.operation_information do
14:        reverse_transform_ordering = reverse(ordering_information)
15:        intermediate_distance_storage_statement = ϕ
16:        for transform_node in reverse_transform_ordering do
17:            function_body.append(apply(transform_node))
18:        end for
19:        if node is a transform node then
20:            distance_variable = node.previous_non_transform_node[index].get_variable()
21:            call_function = "f_" + distance_function + index
22:            return_variable = distance_variable + index
23:            function_body.append(create_distance_storage_statement(distance_variable,call_function)
24:        else
25:            return_variable = node.variable_name()
26:            function_body.append(node.get_string())
27:        end if
28:        index = index + 1
29:    end for
30:    function_definition =create_function_definition(function_name, function_body, return_variable)
31: end for
32: return function_definition =0
```

- **Line 14 : Line 14** reverses the *ordering_ information* since the operations need to be applied in reverse order to undo the effect of those operations and return the object to the origin.

- **Line 16 :** The algorithm then iterates over the operation nodes between the current node and the previous non-transform node in the **for** loop on **line 16**.

- **Line 14-28 :** Depending on the type of node being processed, **lines 14-28** append the necessary shader code to the *function_ body*.

- **Line 30 :** Finally, **line 30** takes *function_ name*, *function_ body*, and *return_ variable* and returns its function definition.

- **Line 32 :** The shader is returned by the Shader Generation function on **line 32**.

Every generated function corresponding to a primitive or an object node takes a 3D point called 'position' as input and returns a floating point number as output. The returned floating point number signifies the closest distance between the input point and the node itself. This way, we build upon primitive nodes to get the closest distance between the combination of nodes described in the node graph.

Functions generated for Operation Nodes take at least two floating point numbers which are distances, as inputs and transform the distances according to the functions which were predefined and appended during preprocessing.

Transform nodes can occur in series between any two non-transform nodes. As explained before, transform nodes change the spatial properties of the nodes. We apply the transform in reverse order to the 'position' parameter and then call the previous non-transform with the transformed parameters. This way, we mimic the behavior of the previous non-transform node being transformed according to the transformations specified. Transformations are realized essentially by a transformation matrix that yields the transformed point when multiplied by a three-dimensional vector.

Following code is generated by our Shader Generation Algorithm and renders the very basic red ball on blue horizon image. Utility functions, primitives and scatter functions that are pasted as-is in the shader are omitted in the given code for the sake of brevity. Similarly, the *main()* function and the raymarching loop has been omitted as it is pasted as-is and not generated by our algorithm. All functions in the following code are thus, procedurally generated by our algorithm. These functions are then used by the raymarching code to render an image. The image rendered by this can be seen in figure 3.22.

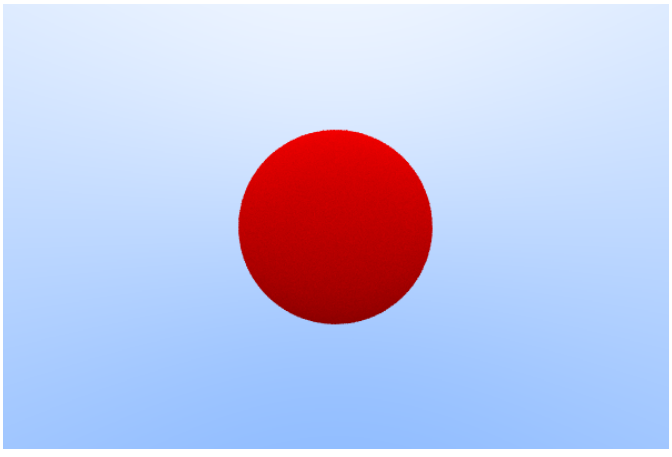


Figure 3.22: Default Render generated by the sample code

Listing 3.1: Fragment Shader (Generated by the algorithm)

```
// PREAMBLE
#version 330 core
uniform vec3 u_camera_origin;
uniform vec2 u_viewport_size;
uniform float u_focal_length;
uniform vec3 u_r_horizon_bottom_color;
uniform vec3 u_r_horizon_top_color;
uniform int u_r_Max_Depth;
uniform int u_r_Samples;
uniform int u_r_Number_of_steps;
uniform int u_r_Maximum_Trace_Distance;
out vec4 FragColor;
uniform bool DEBUG;
uniform bool DEBUG_DEPTH;
uniform bool DEBUG_MAX_TRACE;
uniform bool DEBUG_STEPS_END;
uniform vec3 DEBUG_DEPTH_COL;
uniform vec3 DEBUG_MAX_TRACE_COL;
uniform vec3 DEBUG_STEPS_END_COL;
vec2 random_val = vec2(0.0);
// END OF PREAMBLE

vec3 get_color(int object_index);

// STRUCTS
struct closest_object_info
{
    float closest_distance;
    int object_index;
};
struct scatter_info
{
    vec3 scattered_ray;
    bool is_scattered;
    vec3 attenuation;
};
// END OF STRUCTS
```

```
uniform float u_Sphere_0_Radius;
uniform vec3 color_0;

vec3 get_color(int object_index)
{
switch(object_index)
{
case 1:
return color_0;
break;
}
return vec3(1.0, 0, 0.0);
}

bool is_light(int object_index)
{
bool res = false;
switch(object_index)
{
case 1:
res = false;
break;
}
return res;
}

float f_Sphere_0(vec3 position)
{
mat3 rotation_transform_x = mat3(1.0);
mat3 rotation_transform_y = mat3(1.0);
mat3 rotation_transform_z = mat3(1.0);
vec3 position_dup = position;
position = position_dup;
position = position/(1.0);
float Sphere_0 = Sphere(position, u_Sphere_0_Radius) * 1.0;
return Sphere_0;
}

float object_1(vec3 position)
```

```
{
mat3 rotation_transform_x = mat3(1.0);
mat3 rotation_transform_y = mat3(1.0);
mat3 rotation_transform_z = mat3(1.0);
vec3 position_dup = position;
position = position_dup;
position = position/(1.0);
float Sphere_00 = f_Sphere_0(position) * 1.0;
return Sphere_00;
}

float get_distance_from(vec3 position, int object_index)
{
float d = 0.0;
switch(object_index)
{
case 1:
d = object_1(position);
break;
}
return d;
}

closest_object_info get_closest_object_info(vec3 position)
{
float dist_1 = object_1(position);
float min_dist = 3e+38;
int object_index = 1;
min_dist = min(min_dist, dist_1);
if(dist_1 == min_dist)
{
object_index = 1;
}
return closest_object_info(min_dist, object_index);
}

vec3 calculate_normal(vec3 position, int object_index)
{
const vec3 small_step = vec3(0.001, 0.0, 0.0);
```

```
vec3 normal = vec3(1.0, 1.0, 1.0);
float g_x, g_y, g_z;
switch(object_index)
{
case 1:
g_x = object_1(position + small_step.xyy) - object_1(position - small_step.xyy);
g_y = object_1(position + small_step.yxy) - object_1(position - small_step.yxy);
g_z = object_1(position + small_step.yyx) - object_1(position - small_step.yyx);
normal = normalize(vec3(g_x, g_y, g_z));
break;
}
return normal;
}

scatter_info get_target_ray(vec3 position, int object_index, vec3 normal, vec3 r_in,
    bool is_front_face)
{
scatter_info target = scatter_info(vec3(0.0,0.0,0.0), true, vec3(1.0, 1.0, 1.0));
switch(object_index)
{
case 1:
target = diffuse_scatter(position, normal, object_index);
break;
}
return target;
}
```

Chapter 4

Results

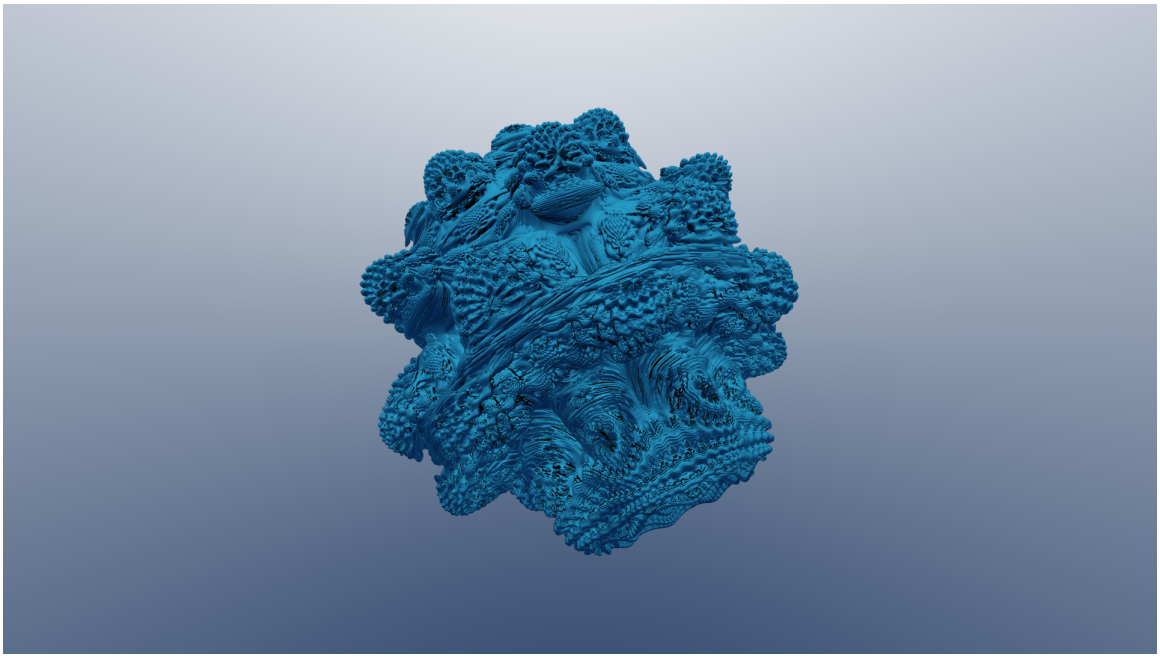
4.1 Paper Submission

We have successfully submitted a paper titled "ProcSDF: An Extensible Node-Based Raymarching Playground" to Wiley's "Software: Practice and Experience" journal, which focuses on practical experiences related to software systems and practices. The paper shares a similar structure to this report and provides detailed insights into the features and implementation of ProcSDF. It aims to serve as a reference for anyone interested in implementing similar software. Our paper has been submitted as an Experience Report, which emphasizes explaining the software's implementation, the interesting challenges we faced, and the results obtained. We are hopeful that our paper will be published in the journal.

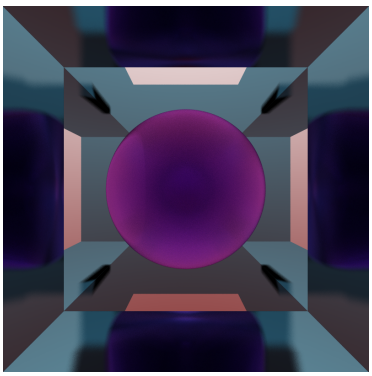
4.2 Renders

This section showcases a few renders created using ProcSDF and their render times. A few things are to be noted before we get into the details. Bench-marking for these renders was done in a non-isolated desktop environment, so several factors might have affected the render times.

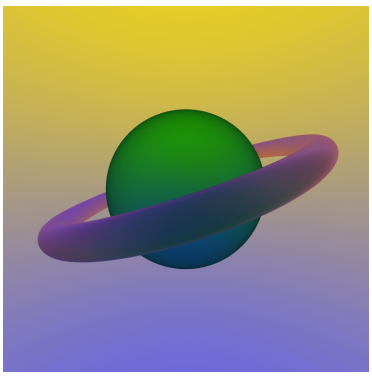
Figure 4.1 shows some renders made using ProcSDF while table 4.1 illustrates details for these renders. $I.R.T$ is the Initial Render Time while $A.R.T$ is the Average Render Time. Every time an image is rendered for the first time, the time it takes for the same is much higher than the subsequent renders, so we have included both the initial time taken (I.R.T) and the average time taken (A.R.T) in the subsequent renders. The render times remain a decent representative of the relative load that certain renders might put on the software while not being extremely accurate. The render times for "Two metal balls" show this with the unexpected reduction in render time with an increase in samples which might result from bench-marking the software in a



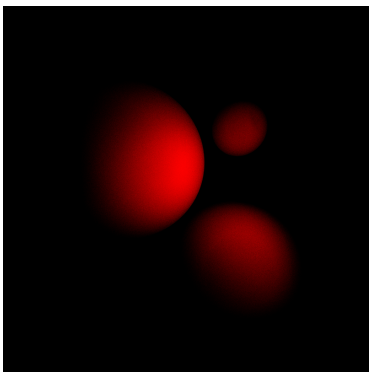
(a) "Mandelbulb"



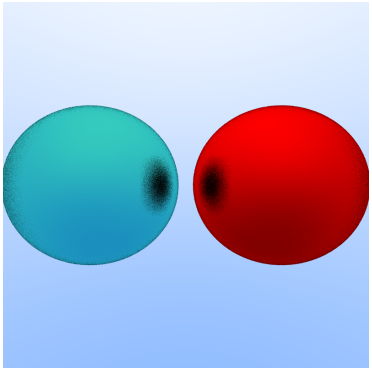
(b) "Mirrors and Glasses"



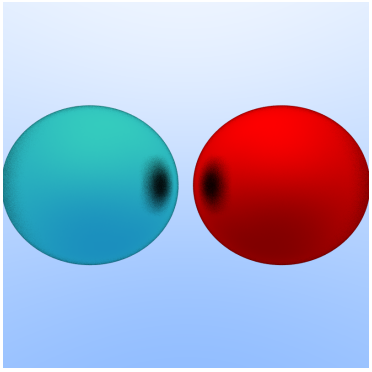
(c) "Green Planet"



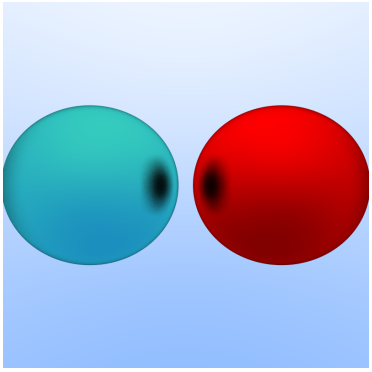
(d) "Lights"



(e) "Two metal balls" (10 samples)



(f) "Two metal balls" (50 samples)



(g) "Two metal balls" (250 samples)

Figure 4.1: A collection of images rendered using ProcSDF

Name	Samples	Max Depth	Size	I.R.T(ms)	A.R.T(ms)
Mandelbulb	50	500	(1920, 1080)	916	1.5
Mirrors and Glasses	200	25	(800, 800)	923.95	1.7
Green Planet	500	50	(800, 800)	413.7	2.03
Lights	250	25	(800, 800)	1.7	1.4
Two metal balls	10	25	(800, 800)	422	2.06
Two metal balls	50	25	(800, 800)	455	1.55
Two metal balls	250	25	(800, 800)	422	1.27

Table 4.1: Rendering details for the example renders provided

non-isolated environment. The benchmarking was done on a HP Pavillion Gaming Laptop with processor specifications: Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz 2.30 GHz, 8 GB RAM, and a NVIDIA GeForce GTX 1050 Ti.

Mandelbulb showcases a Mandelbulb rendered with all its intricate randomness. The Mandelbulb itself was added to the scene using the Custom Nodes feature of ProcSDF. *Smoke and Mirrors* showcases a dielectric sphere surrounded by a metal box frame. ProcSDF's ability to render striking images replete with materials full of character is showcased here. The jump in rendering time due to increased max depth should be noted here. *Lights Experiment* showcases the lighting capabilities of ProcSDF while *Two metal balls* was just a simplistic render to illustrate the resulting difference in the quality of the render to changes in the number of samples. Following subsections shall illustrate case studies on a few of these renders.

4.3 Case Study: Mandelbulb

The setup process for the Mandelbulb render using ProcSDF was straightforward. The Mandelbulb is a complex 3D object with an intricate random shape, conceptualized as a higher dimensional extension of the Mandelbrot set, which exhibits randomness in only two dimensions. This case study aimed to demonstrate the implementation of such esoteric Signed Distance Functions (SDFs) in ProcSDF. The following steps were involved in the process:

1. Import the necessary code for the Mandelbulb as shown in Figure 3.15 [21] and integrate it as a custom node in the ProcSDF system.

2. Set up the node graph for the Mandelbulb object as depicted in Figure 4.2, using the custom node directly in the graph.

3. Configure the metal material with a *roughness* value of *0.0* to produce a smooth metallic appearance.

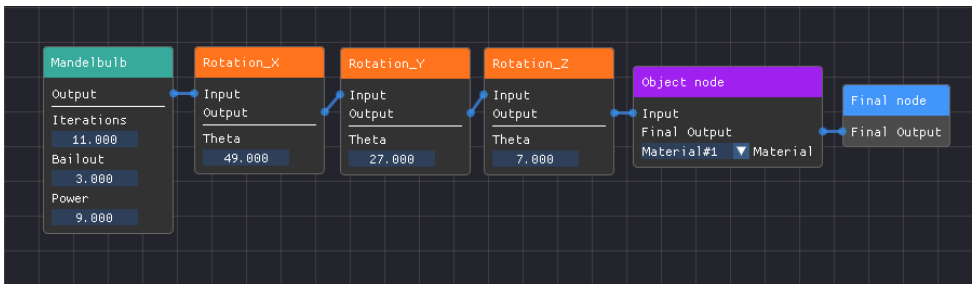


Figure 4.2: Node Graph for Mandelbulb

This exercise demonstrates the straightforward process of integrating custom code for Signed Distance Functions (SDFs) into ProcSDF and configuring them with the appropriate environment and material properties to achieve visually appealing renders with minimal effort. This feature is expected to be advantageous for researchers and enthusiasts who aim to rapidly generate proofs of concept for their SDFs by importing them into ProcSDF.

4.4 Case Study: Mirrors and Glasses

This case study demonstrates ProcSDF's ability to render realistic materials such as metals and dielectrics. The objective was to examine if ProcSDF could generate believable renders by placing metallic objects resembling mirrors near dielectric objects and allowing them to interact. The following steps were taken to set up the render:

1. The node graph was established, as depicted in Figure 4.3. To distinguish between the objects, the *BoxFrame* and *Sphere* were supplied to separate *Object* Nodes, as distinct materials were to be applied to them.

2. The *metal* material was established with a *roughness* value of *0.03* for the *BoxFrame*, while the *sphere* was provided with a *dielectric* material with an *IOR* of *1.5* and *roughness* of *0.28*. This gave the sphere a glass-like appearance.

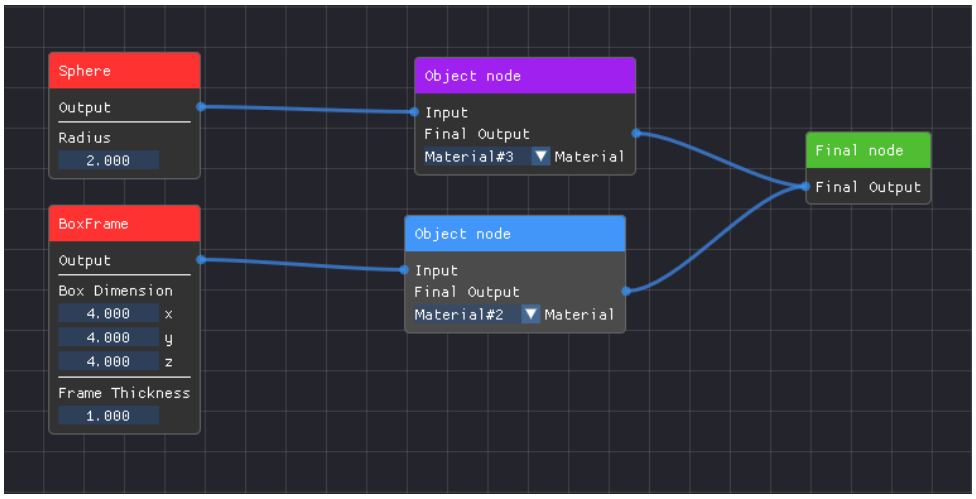


Figure 4.3: Node Graph for "Mirrors and Glasses"

This example demonstrates ProcSDF's ability to create visually appealing scenes with realistic interactions between materials such as metals and dielectrics. The process highlights the intuitive workflow of creating and using various materials to achieve a scene's desired look and feel. The simple node graph used in this exercise demonstrates how one can easily produce impressive and visually stunning renders with the appropriate materials and settings.

4.5 Case Study: Lights

This case study involves setting up emissive objects in a dark environment to see how lights behave in ProcSDF. Following are the steps involved in setting up the render :

1. Set up the node graph as seen in figure 4.4. The top three spheres are the ones that are visible in the scene, while the last one makes up the off-screen light that illuminates these spheres.

2. The three spheres use a basic *diffuse* material colored red.

3. The last sphere uses an *emissive* material. Had it been onscreen, it would have acquired a flat feel similar to how lights look when looked at directly. However, off-screen, it illuminates nearby objects, creating a sense of light in the dark environment.

Using emissive materials to create lights in ProcSDF demonstrates the simplicity of lighting in raymarching. However, it also highlights the limitations of the lighting in terms of realism. The probabilistic nature of raymarching, which uses random sampling of rays, can result in lights having less palpable effects on objects onscreen than in realistic settings. To address this, researchers have proposed methods such as iRay [24], which uses importance sampling to reduce

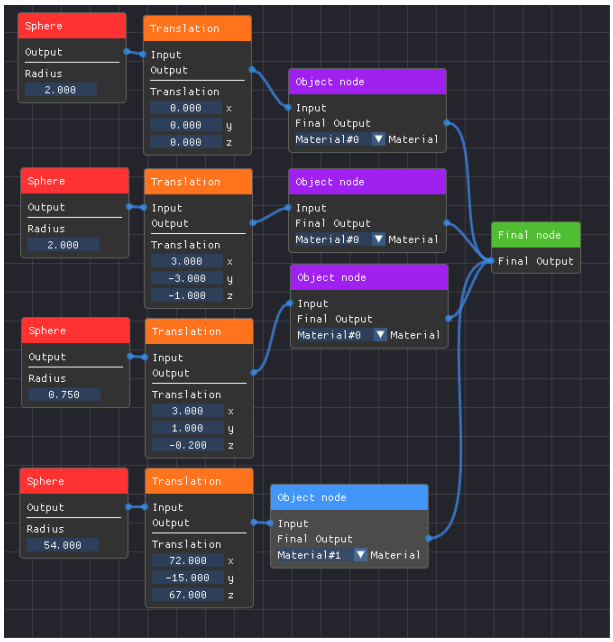


Figure 4.4: Node Graph for "Lights Experiment"

the noise in the lighting, as well as bidirectional light transport [25, 26] and Metropolis light transport [27], which aim to improve the accuracy of lighting simulations. Future iterations of ProcSDF could potentially incorporate such techniques to enhance its lighting capabilities.

Chapter 5

Conclusions and Future Work

Going forward, additional rendering features such as volumetric rendering and advanced lighting models can be added to ProcSDF to expand its capabilities. Additionally, the performance of the raymarching algorithm can be optimized. The raymarching algorithm is not error-proof in terms of rendering materials and surfaces as per the expectations all the time, and work can be done towards ironing out such errors.

ProcSDF can prove to be useful in furthering ML research surrounding SDFs. For instance, neural networks can be used to generate and refine SDFs, resulting in more efficient and accurate rendering.[28] [29]

Furthermore, the interface is as experimental as it gets and has a lot of scope to improve in terms of UI/UX. Additionally, support for importing and exporting scenes and models from other 3D software applications can be added.

Continued engagement with the Raymarching community can provide valuable feedback and suggestions for future developments. By collaborating with artists and researchers, ProcSDF can remain at the forefront of raymarching software development and continue to advance 3D rendering technology.

In conclusion, ProcSDF is a node-based raymarching tool that provides an intuitive workflow for exploring the rendering algorithm. By creating custom nodes and experimenting with different features, ProcSDF has the potential to expand the capabilities of raymarching and drive further innovation in 3D rendering. Continued collaboration with the Raymarching community can help gather valuable feedback and suggestions for future developments. In this way, ProcSDF represents a promising tool for artists and researchers looking to explore the possibilities of raymarching and contribute to the advancement of 3D graphics.

Bibliography

[1] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*. Addison-Wesley, 1996.

[2] S. R. Buss, *3D computer graphics: a mathematical introduction with OpenGL*. Cambridge University Press, 2003.

[3] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-time rendering*. AK Peters/CRC Press, 2008.

[4] M. Pharr and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016.

[5] L. J. Tomczak, "Gpu ray marching of distance fields," 2012.

[6] W. 3D, "Womp : The new way to 3d." Last accessed 1 January 2023.

[7] M. Walczyk, "Sdfperf." Last accessed 1 January 2023.

[8] J. Hart, "Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces," *The Visual Computer*, vol. 12, 06 1995.

[9] I. Quilez, "Sdf code." Last accessed 8 November 2022.

[10] J. Cole, "Sdfs." Last accessed 8 November 2022.

[11] M. Walczyk, "Raymarching." Last accessed 8 November 2022.

[12] T. Reiner, G. Mückl, and C. Dachsbacher, "Interactive modeling of implicit surfaces using a direct visualization approach with signed distance functions," *Computers Graphics*, vol. 35, no. 3, pp. 596–603, 2011. Shape Modeling International (SMI) Conference 2011.

[13] B. Zhang and K. Oh, "Interactive indirect illumination using mipmap-based ray marching and local means replaced denoising," in *Proceedings of the 25th ACM Symposium on Virtual Reality Software and Technology*, VRST '19, (New York, NY, USA), Association for Computing Machinery, 2019.

[14] C. Sun and E. Agu, "Many-lights real time global illumination using sparse voxel octree," in *Advances in Visual Computing* (G. Bebis, R. Boyle, B. Parvin, D. Koracin, I. Pavlidis, R. Feris, T. McGraw, M. Elendt, R. Kopper, E. Ragan, Z. Ye, and G. Weber, eds.), (Cham), pp. 150–159, Springer International Publishing, 2015.

[15] M. Csongei, L. Hoang, U. Eck, and C. Sandor, "Clonar: Rapid redesign of real-world objects," in *2012 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, pp. 277–278, 2012.

[16] O. Rouiller, "Real time rendering of skeletal implicit surfaces." Last accessed 11 April 2023.

[17] R. Vergne and P. Barla, "Designing gratin, a gpu-tailored node-based system," *Journal of Computer Graphics Techniques (JCGT)*, vol. 4, pp. 54–71, November 2015.

[18] P. B. Silva, P. Müller, R. Bidarra, and A. Coelho, "Node-based shape grammar representation and editing," *Proceedings of PCG*, pp. 1–8, 2013.

[19] H. P. A. Lensch, M. Goesele, Y.-Y. Chuang, T. Hawkins, S. Marschner, W. Matusik, and G. Mueller, "Realistic materials in computer graphics," in *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, (New York, NY, USA), p. 1–es, Association for Computing Machinery, 2005.

[20] T. Letz, "Raymarching the mandelbulb fractal in vr," 06 2018.

[21] P. T. R. Schneider, "shader-fractals." Last accessed 5 April 2023.

[22] N. Folkegård and D. Wesslén, "Dynamic code generation for realtime shaders," in *The Annual SIGRAD Conference. Special Theme-Environmental Visualization*, no. 013, pp. 11–15, Linköping University Electronic Press, 2004.

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT press, 3 ed., 2009.

[24] A. Keller, C. Wächter, M. Raab, D. Seibert, D. van Antwerpen, J. Korndörfer, and L. Kettner, "The iray light transport simulation and rendering system," 2017.

[25] E. Veach and L. Guibas, "Bidirectional estimators for light transport," in *Photorealistic Rendering Techniques* (G. Sakas, S. Müller, and P. Shirley, eds.), (Berlin, Heidelberg), pp. 145–167, Springer Berlin Heidelberg, 1995.

[26] V. Frolov, A. Voloboy, S. Ershov, and V. Galaktionov, "Light transport simulation and realistic rendering: State of the art report," pp. 1–12, 01 2021.

[27] D. Cline and P. Egbert, "A practical introduction to metropolis light transport," 04 2005.

[28] J. J. Park, P. Florence, J. Straub, R. Newcombe, and S. Lovegrove, "Deepsdf: Learning continuous signed distance functions for shape representation," 2019.

[29] S. Lombardi, T. Simon, J. Saragih, G. Schwartz, A. Lehrmann, and Y. Sheikh, "Neural volumes," *ACM Transactions on Graphics*, vol. 38, pp. 1–14, jul 2019.