SDF (Signed Distance Field) functions visual editor in Unity

Wizualny edytor SDF (funkcji odległości ze znakiem) dla silnika Unity

Maksymilian Polarczyk

Praca inżynierska

Promotor: dr Łukasz Piwowar

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

Wrocław 2024

Abstract

This thesis presents a comprehensive tool for the Unity engine that simplifies the creation and editing of scenes represented using Signed Distance Fields (SDFs). By providing an intuitive, interactive user interface, the tool eliminates the need for manual shader programming, enabling users to design complex geometries visually.

Key features of the tool include:

- An interactive user interface for creating and manipulating SDF scenes, allowing users to design complex geometries visually.
- An extendable API for generating Shaderlab and HLSL files using Abstract Syntax Trees (ASTs), facilitating structured shader code representation.
- A tree-based system for representing shaders as networks of connected primitive nodes, enabling flexible and modular shader development.
- Editor widgets for controlling SDF scene primitives, enhancing user interactivity and control.
- A library of pre-built primitives and operators, offering ready-to-use components for efficient SDF scene construction.
- Example scenes demonstrating the tool's capabilities and providing a starting point for users.

In addition to the tool's implementation, this thesis provides brief comparisons with existing similar tools, an overview of the technologies leveraged in the tool's development, and discusses potential directions for future work.

By streamlining the SDF scene creation process and providing powerful, extensible APIs, this tool aims to lower the barrier to entry for developers and artists, fostering greater experimentation and innovation in SDF-based rendering within the Unity engine.

Niniejsza praca przedstawia kompleksowe narzędzie dla silnika Unity, które upraszcza tworzenie i edytowanie scen reprezentowanych za pomocą funkcji odległości ze znakiem (SDF). Dzięki intuicyjnemu, interaktywnemu interfejsowi użytkownika, narzędzie eliminuje konieczność ręcznego programowania shaderów, umożliwiając użytkownikom wizualne projektowanie złożonej geometrii.

Kluczowe elementy pracy obejmują:

- Interaktywny interfejs użytkownika do tworzenia i manipulowania scenami SDF, umożliwiający wizualne projektowanie złożonej geometrii.
- Otwarte API do generowania plików w językach Shaderlab i HLSL za pomocą Drzew Składni Abstrakcyjnej (AST), ułatwiające strukturalne przedstawienie kodu shaderów.
- System oparty na drzewach do reprezentowania shaderów jako sieci połączonych prymitywnych węzłów, umożliwiający elastyczne i modularne tworzenie shaderów.
- Elementy interfejsu do kontrolowania prymitywnych obiektów sceny SDF, zapewniające użytkownikom wysoką interaktywność tworzonej sceny.
- Własnoręcznie stworzoną bibliotekę gotowych do użycia prymitywów i operatorów, oferującą komponenty do efektywnej konstrukcji scen SDF.
- Przykładowe sceny demonstrujące możliwości narzędzia i stanowiące dobry punkt wyjścia dla użytkowników.

Oprócz implementacji narzędzia, niniejsza praca zawiera krótkie porównanie z istniejącymi, podobnymi narzędziami, przegląd technologii wykorzystanych w rozwoju narzędzia oraz omawia potencjalne kierunki przyszłych prac.

Usprawniając proces tworzenia scen SDF i oferując rozwinięte, rozszerzalne API, narzędzie to ma na celu obniżenie bariery wejścia dla deweloperów i artystów, wspierając eksperymentowanie i innowacje w renderowaniu opartym na SDF w ramach silnika Unity.

Contents

1 Introduction **7**

1.1 Motivation 7

1.2 Existing tools 8

2 Raymarching **11**

2.1 Overview 11

2.2 Theory 13

2.2.1 Geometry 14

2.2.2 Examples of SDF primitives 15

2.2.3 SDF operators 15

2.2.4 SDF normals 16

2.2.5 SDF shading 17

3 Project overview **21**

3.1 Components 21

3.2 SDF scene generation 22

3.3 Adding objects to SDF scenes 22

3.4 Custom controller editors 23

3.5 SdfScene inspector 24

3.6 Instantiating SDF scenes 25

3.7 Configuring generated materials 26

3.7.1 Debug modes 26

3.7.2 Gizmos 29

3.8 Examples 31

4 Implementation **35**

4.1 Programming environment and tools 35

4.2 Architecture and design 35

4.2.1 AST 35

4.2.2 Generation 39

4.2.3 Generators 42

4.2.4 Problems 42

5 Summary and conclusions **47**

5.1 Future work 47

Bibliography **49**

Chapter 1

Introduction

1.1 Motivation

Rendering using Signed Distance Fields (SDFs) is a technique known in many fields related to computer graphics. It is often used by the demoscene community for achieving complex effects and drawing complicated scenes with high flexibility and ease that could be otherwise hard or inefficient to achieve with regular mesh or raytracing based techniques. Signed Distance Fields have been applied in various industries, including web development (notably in font rendering), the robotics and computer vision industry (representing the world using SDFs), art, and the game development industry (for real-time Global Illumination, Constructive Solid Geometry, dynamic particle systems, and more).

Even though SDFs provide easy access to complicated effects (e.g., soft shadows, Boolean operations, infinite repetition, cheap ambient occlusion), they also come with some trade-offs:

- They often require developers to have specialized, hard-to-find knowledge of non-trivial graphics programming, raising the barrier to entry for novices.
- Furthermore, without the availability of visual tools, developers are often forced to partake in a tedious process of adjusting shader parameters by trial and error. Tools like Shadertoy [1] provide only the most basic tools for creating shaders, without much support for user or developer interactivity.
- Oftentimes, the optimized SDF routines are obscure, undocumented, copy pasted code snippets, which leads to even more confusion for beginners.
- There is no single source of truth for the definitions of SDF-based functions, leading to a common practice of copy-pasting similar undocumented implementations scattered all over the Internet.

- When creating the SDF scene, adjusting the components manually leads to shader recompilation, which slows down development even further. Optimizing this process to use widgets and data passed to shaders would require considerable effort from anyone trying to achieve it.

This work has been motivated by the aforementioned shortcomings and aims to provide a simpler and more robust way of generating and editing SDF scenes.

1.2 Existing tools

Several software solutions address similar challenges in rendering using Signed Distance Fields (SDFs), each with its strengths and weaknesses.

Shadergraph, integrated within Unity, is a stable and actively maintained tool that offers good performance and a live, real-time preview. However, it is limited to simple evaluation model without treating SDF functions as proper values, it is semi-open with native bindings to closed-source libraries, and lacks support for user generated port data or a support for scene-view interactivity and extensibility using gizmos. Integrating the Shadergraph and the work of this thesis can be a subject of the future work.

Womp [2] is a cloud-only, proprietary tool known for its intuitive user experience but suffers from performance issues as scene complexity increases. It requires a constant online connection, and on poor connections, the delay can make the tool less interactive. Additionally, Womp is non-extensible, and many basic features are only available with a subscription, making it less accessible for users needing advanced functionalities. Models created in Womp can't be easily used in other tools such as Shadertoy or in game engines, unless a baked mesh were to be used. The set of operations available in Womp is limited and fixed, thus composition of advanced SDF operators may turn out to be difficult.

Unbound [3], yet to be released, promises to combine web functionality with addons for integrating with other game engines such as Unity Engine, Unreal Engine, Blender and Godot Engine, potentially offering high extensibility and an intuitive, albeit somewhat simplistic, user experience. The scope of this project is however yet unknown. It is unknown if this tool will generate shaders that can be modified or only export scenes as static meshes.

uRaymarching [4] is a discontinued, open source SDF shader generator for Unity. It has more features and supports more rendering pipelines than the work of this thesis. However, it is intended for use with older versions of Unity and has compatibility issues with modern Unity technologies. uRaymarching uses a custom string templating language to construct shaders and provides several templates for generating them. Despite this, it lacks interactivity and the ability to compose scenes

dynamically. Defining SDF scenes with uRaymarching involves manually writing all the shader code, which means that while it helps bootstrap a raymarching renderer, it does not offer interactive scene controls. The final shaders depend on a single, manually written SDF function. Some techniques and examples from this package have been adapted in the current work, primarily in relation to interacting with the Unity Engine API.

MudBun [5] is a premium, paid Unity package derived from the older, open-source Clayxels [6] package, designed for working with SDF and volumetric geometry in the Unity engine. While MudBun is a mature tool, its cost can be a significant barrier to entry for novices. It is extensible and provides APIs for defining custom geometry brushes, though it is unclear if the package allows access to and modification of the source code. It provides an intuitive graphical interface for working with the geometry. MudBun supports four drawing modes (smooth mesh, flat mesh and two splatting variants) and several meshing algorithms. Without obtaining the tool it is not entirely transparent how the rendering process works, as the author mentions the use of meshing algorithms, compute shaders, and the necessity to sample SDFs multiple times. It appears that the tool itself does not generate flexible shaders; instead, it likely uses compute shades to generate meshes directly on the GPU by sampling the SDF or volume data and renders the generated mesh using regular shading techniques.

In contrast, this work aims to provide a new, experimental, interactive and highly extensible solution with a more flexible and powerful API for generating shader code. It is open-source under the MIT license and developed in C# with a Unity base. Unlike the aforementioned tools, this project offers all of the following features: interactive, offline and non-destructive editing directly in Unity, simple generation model, flexible data types, easy to use GUI elements, capability to generate readable, reusable shader code and partial scripting support. Though still experimental and supporting only a simple shading model which doesn't utilize full capabilities of the Built In Render Pipeline in Unity, this solution offers a robust alternative serving as a base with potential for high customizability and free availability, addressing many limitations observed in existing tools and lowering the barrier of entry.

It is worth noting that the development of this tool began prior to the discovery of the other tools mentioned in this section. However, as the project progressed, certain design decisions and features were influenced and inspired by the capabilities and approaches observed in these existing tools. This integration allowed for a more comprehensive and user-friendly implementation, incorporating some of the best practices from the field.

Chapter 2

Raymarching

2.1 Overview

Raymarching is a rendering technique that traces rays through a scene to determine the distance to the nearest surface and uses this information to compute the geometry and other image effects. This technique was known since at least the 1980s [7] as a method for rendering implicit surfaces and has since been popularized by the demoscene real-time graphics communities and prominent individuals like Inigo Quilez [8] due to its flexibility and efficiency in rendering complex, stunning, artistic scenes.

Unlike traditional ray tracing, which calculates intersections with explicit geometric objects, raymarching renders geometry using implicit geometry definition by simulating rays coming out of the virtual camera and using Signed Distance Fields (SDFs) estimating the distance to the nearest surface, continuing until it is determined that the ray has hit a surface. This allows for the rendering of highly detailed and mathematically defined shapes with smooth surfaces, complex lighting, and effects such as soft shadows and cheap (in terms of GPU memory and computation budget) ambient occlusion. The term ”raymarching” itself describes a process of interactively performing consecutive ”steps” along the ray until it is determined that a surface has been hit.

Raymarching provides some advantages over other methods like rasterization and traditional ray tracing. Rasterization is fast and well-suited for hardware acceleration but struggles with representing complex geometries and often requires the geometry to be defined explicitly a priori. Things like Constructive Solid Geometry (CSG) can be problematic for traditional raster renderers. Traditional ray tracing, while capable of producing high-quality images with accurate reflections and refractions, can be computationally expensive and difficult to implement efficiently for scenes with intricate details. Raymarching, by leveraging SDFs, allows for the representation of complex and procedurally generated surfaces with relatively simple

and efficient mathematical descriptions. Another major advantage of SDFs is their ability to represent seemingly infinite (up to the numerical limits) detail without the loss of quality, i.e. something not easily achievable in mainstream raster renderers without complex approximation techniques such as various kinds of normal, bump and parallax mapping, hierarchical Level Of Details (LODs) and others.

The main trade-offs of raymarching include the need for specialized knowledge in mathematical functions and optimizations, as well as skills necessary for optimizing shaders for extremely detailed scenes due to the iterative nature of distance field evaluations. Usually Signed Distance Field shaders are additionally static scenes, or parametrized scenes where the number and definitions of primitives don't dynamically change, simply because it would either involve dynamic branching in the shader code leading to very poor performance or recompiling shaders at runtime. In addition, some graphic effects may depend on strict mathematical properties of SDFs leading to rendering more or less noticeable rendering artifacts when these properties are violated. Recent technological advancements in rendering and graphic APIs, for example support for HLSL classes and interfaces, may however soon help in resolving some of these issues. Despite these challenges, raymarching continues to gain traction in various fields, including computer graphics, game development, and interactive art installations, due to its versatility and the high quality of the visual results it can achieve and the growing compute capabilities of the mainstream hardware.

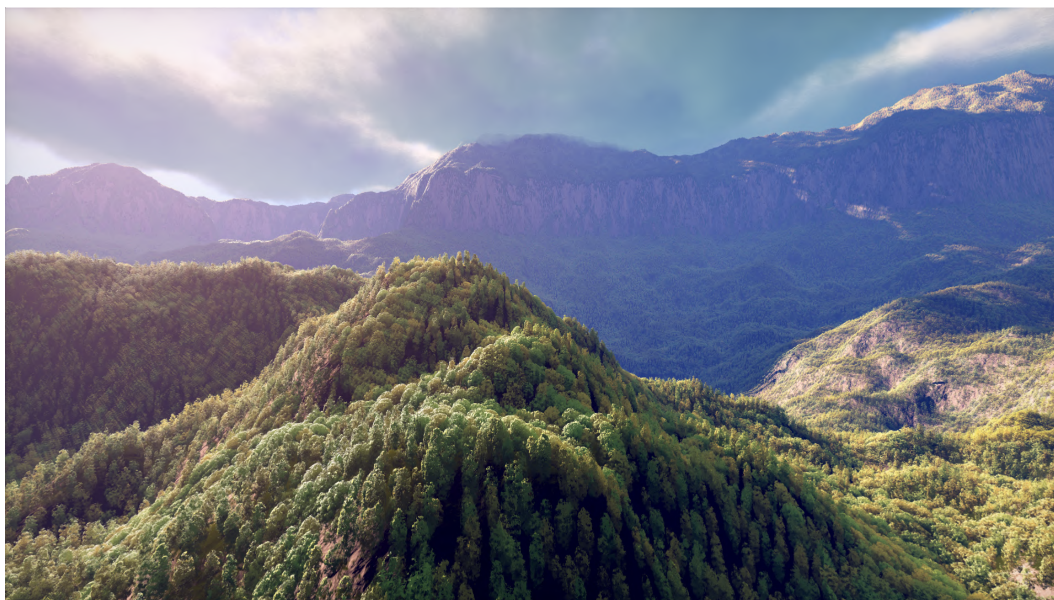

Figure 2.1: Render of the 'Rainforest' shader using SDF raymarching,
author: Inigo Quilez [9]

Figure 2.2: Render of the 'Snail' shader using SDF raymarching,
author: Inigo Quilez [9]

2.2 Theory

The theory of SDF raymarching is extensively researched and explained in existing works [10] [11] and is not within the primary scope of this thesis. However, a brief description is provided here for clarity.

Unlike regular raster renderers, SDF raymarching uses implicit surfaces defined by scalar fields called Signed Distance Fields. For any point p in space, the value of

$$signed_distance = SDF(p)$$

describes the signed, shortest distance to the surface of the implicit object from that point. A positive value indicates that the point is outside the geometry, while a negative value indicates that the point is inside the geometry. This definition implies that the surface of the implicit object is a set of points where the SDF evaluates to 0. Using this definition, we can infer that within a sphere of radius $|SDF(p)|$ centered at point p, there are no points belonging to the surface (simply provable by contradiction).

Rendering geometry defined using SDFs involves simulating the travel along rays outgoing from each "pixel" of the camera until the surface is hit. We start at the ray origin (usually the camera's near plane in world space) and iteratively step forward along the ray. Unlike regular raymarching or raytracing, which perform fixed steps, we use the properties of SDFs to accelerate the process by taking a step of size $SDF(p)$. We stop when the ray hits the object, i.e., when $SDF(p) = 0$. In

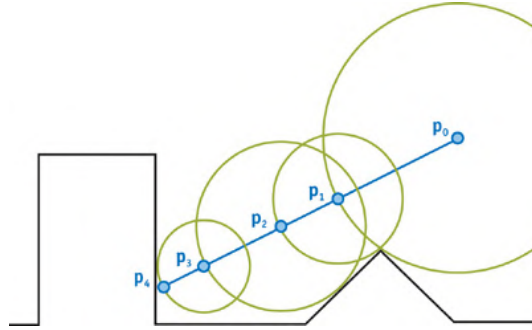

Figure 2.3: Raymarching, from GPU Gems 2: Chapter 8 [11]

practice, due to numerical errors, we stop when we are close enough to the surface, using a very small distance ϵ, i.e., when $|SDF(p)| < \epsilon$.

Given this SDF definition, traversing along the ray becomes straightforward. Instead of performing fixed steps, as in regular raymarching or raytracing, we can safely traverse forward by stepping a distance equal to $SDF(p)$ at each iteration.

2.2.1 Geometry

One of the primary challenges of SDF raymarching is the necessity to define functions describing the geometry of primitives. This process can be intricate and impractical compared to traditional 3D modeling and rendering techniques, which rely on explicit geometry typically represented by triangles. Triangular surfaces are straightforward to manipulate and render using rasterization or ray tracing. In contrast, SDF-based techniques require mathematical descriptions of surfaces, which can pose difficulties in deriving and optimizing complex shapes.

Alternative methods exist for rendering implicit surfaces from pre-existing data, such as randomly sampling the mesh and precomputing an SDF into a low-resolution 3D texture meant for sampling during raymarching. While this approach simplifies the representation of complex geometries, it requires an additional precomputation step, some memory and performance overhead and finally can sometimes lead to interpolation artifacts and a loss of detail. However, these techniques, though valuable, fall outside the scope of this thesis, which concentrates on real-time generation and manipulation of SDFs.

Despite the challenges, there are comprehensive online resources available for common SDF primitives and operations. For example, the `hg_sdf` library [12] and Quilez's work [13] provide extensive collections of SDF functions and transformations for describing various geometric shapes and effects. These resources serve as invaluable assets for developers aiming to implement SDF-based rendering without having to derive all functions from scratch.

2.2.2 Examples of SDF primitives

Below are derivations of SDFs for a sphere and a box in pseudo-HLSL code. These primitives are defined relative to the center of the Cartesian space at $(0,0,0)$.

```
float sdf_sphere(float3 p, float radius) {
    return length(p) - radius;
}

// b are 3 half-sizes of the box
float sdf_box(float3 p, float3 b) {
    float3 d = abs(p) - b;
    return length(max(d, 0)) + min(max(d.x, d.y, d.z), 0);
}
```

Listing 1: Sphere and box SDF primitives written in HLSL pseudocode.

These, along with other primitives, can be found in the source code of the program within the `primitives.hlsl` include file.

2.2.3 SDF operators

One of the key advantages of SDFs is the ability to combine and transform geometry in nontrivial ways using various operators. These operators facilitate the creation of complex, artistic scenes by employing simple primitives and transformations or combinations using operations like union, intersection, difference, and smooth blending, to name a few. Below are definitions of some commonly used SDF operators. These operators, which often require expensive, numerically inaccurate computations or complicated algorithms and compute shaders executed on GPU in conventional raster renderers, are typically simple and inexpensive to add in SDF raymarchers.

- Union: Combining two SDFs results in a smaller distance of the two:

$$\text{union}(p) = \min(sdf_1(p), sdf_2(p))$$

- Intersection: Taking the maximum of the results from two SDFs results in an intersection of the two:

$$\text{intersection}(p) = \max(sdf_1(p), sdf_2(p))$$

- Flipping the signed distance field "inside out" is simply flipping the sign:

$$\text{flipped}(p) = -sdf(p)$$

- Subtraction: Subtracting SDF_2 from SDF_1 can be performed by taking the maximum of the first SDF and the inverted second SDF:

$$\text{subtract}(p) = \max(sdf_1(p), -sdf_2(p))$$

- Transformation: To translate and rotate the primitive from the world origin $(0, 0, 0)$, perform an inverse transformation of the point where the SDF is evaluated. Transformation is represented using a single translation and rotation matrix T:

$$\text{transformed_sdf}(p, T) = sdf(T^{-1}p)$$

More advanced operations easily achieved with SDFs are smooth combination operators, such as smooth union, smooth intersection, and smooth difference. They are commonly performed by smoothly interpolating between the two returned distances, for example using a cubic polynomial interpolation. However, some interpolation methods may produce incorrect SDFs, leading to rendering artifacts or unnecessary raymarcher steps. More about smooth interpolation of SDFs, their variants, caveats, and comparisons can be found in [14].

Other operations that can be performed on SDFs include extrusion, onion skinning, elongation, scaling, transformation, bending, twisting, repetition, symmetry, revolving around an axis, rounding, change of metric, and others. Their implementations can be found in the source code of the program and in online resources such as [13] and [12].

Due to the implicit nature of SDFs, there are limitations to the capabilities of the operators and their methods of operation. For example, a common operation of scaling in regular raster renderers isn't as straightforward in raymarching. Scaling SDFs uniformly stretches and squishes the space itself, so it should be accounted for during raymarching. Non-uniform scaling is rarely seen in SDF raymarchers due to frequent occurrences of rendering artifacts. Instead, scaling is often applied to individual objects separately by directly changing SDF properties.

2.2.4 SDF normals

An essential aspect of rendering SDFs is shading, which often requires determining surface normals and calculating the influence of lights.

The surface normal of the implicit SDF surface f at point p can be found by evaluating the gradient of the function at that point:

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)$$

Defining a normal for a surface represented with SDF can be achieved in several ways. One approach involves finding analytic solutions by solving derivative equations, but these are often very challenging or even impossible to calculate. Another, more robust method is to approximate the gradient using finite central differences by sampling the SDF in a very close neighborhood around the point p:

$$\vec{n} = \nabla f(p) \approx \begin{bmatrix} f(\ x+\epsilon, & y, & z\ ) - f(\ x-\epsilon, & y, & z\ ) \\ f(\ x, & y+\epsilon, & z\ ) - f(\ x, & y-\epsilon, & z\ ) \\ f(\ x, & y, & z+\epsilon\ ) - f(\ x, & y, & z-\epsilon\ ) \end{bmatrix}$$

It's important to note that the result of such a calculation should be normalized to provide a true, unit-length normal in the shader.

This equation can be implemented in various ways, some more performent and others more accurate. More information about different implementations can be found in [15].

The following approximation holds true for exact SDFs. For approximate SDFs, such as the results of smooth blending, it may produce invalid values and rendering artifacts.

2.2.5 SDF shading

Given the normal definition above, shading an SDF can be performed using universally known shading techniques such as Lambert, Phong, or BRDF shading. Required material properties such as albedo, roughness, index of refraction, and specular can often be provided by using ray direction, hit point, normal, and additional material data returned by the hit surface.

Applying shadows to shaders is often as simple as casting secondary rays from the hit point on the surface towards the lights. If the secondary ray hits another geometry, it indicates that the hit point is in shadow. This technique can be simply extended to render soft shadows at no additional cost. One of the simplest methods involves tracking the minimal distance encountered during the traversal. This distance is then used to calculate the umbra and penumbra shadows. More about this technique, its caveats, implementation details, and variants can be found in [16].

Ambient Occlusion (AO) is yet another effect that can be easily computed in SDF raymarching shaders. AO is often used to make shading more realistic by calculating the influence of nearby geometry on shading, so details like creases or concave surfaces would look believable.

To approximate the AO of a certain point p of SDF, we can use the Monte Carlo method of randomly sampling the neighborhood around p and counting how many samples fall inside the geometry ($sdf(p) < 0$) and how many are outside

($sdf(p) > 0$). The more points fall inside the surface, the more probable it is that the geometry around point p is concave. Counting and calculating the occlusion ratio with appropriate falloff gives an occlusion factor. Details of this technique, optimizations and improvements can be found in [17].

An important part of shading is applying textures to SDFs. Compared to regular renderers, texturing implicit surfaces isn't as easy. In traditional rendering, UV mapping provides a straightforward way to map 2D textures onto 3D surfaces defined by mesh geometry. Each vertex of a mesh has corresponding UV coordinates that directly map to the texture space, allowing for precise control over how the texture is applied.

However, with SDFs, the surfaces are defined mathematically rather than through explicit mesh geometry, which complicates the texturing process. The challenge lies in generating appropriate texture coordinates for points on the implicit surfaces. There are several techniques for texturing SDF objects.

One approach is to use procedural texturing, where the texture is generated procedurally based on the spatial coordinates of the points on the surface. This method does not rely on UV coordinates and can seamlessly cover the entire surface, avoiding issues such as texture seams or distortions. This technique is often used to texture organic surfaces like rocks, dirt, stone, foliage, water, snow and other noisy materials.

Triplanar mapping is another popular technique for texturing SDFs. It blends textures based on the projection from three orthogonal planes (XY, XZ, and YZ), reducing the dependency on UV coordinates and providing a smooth transition between different projections. More about triplanar and biplanar mapping can be found in [18].

For more complex shapes, surface parametrization techniques can be employed to generate UV coordinates. These techniques involve mapping the surface points to a 2D domain in a way that minimizes distortion, but they are more computationally intensive and challenging to implement for arbitrary SDFs.

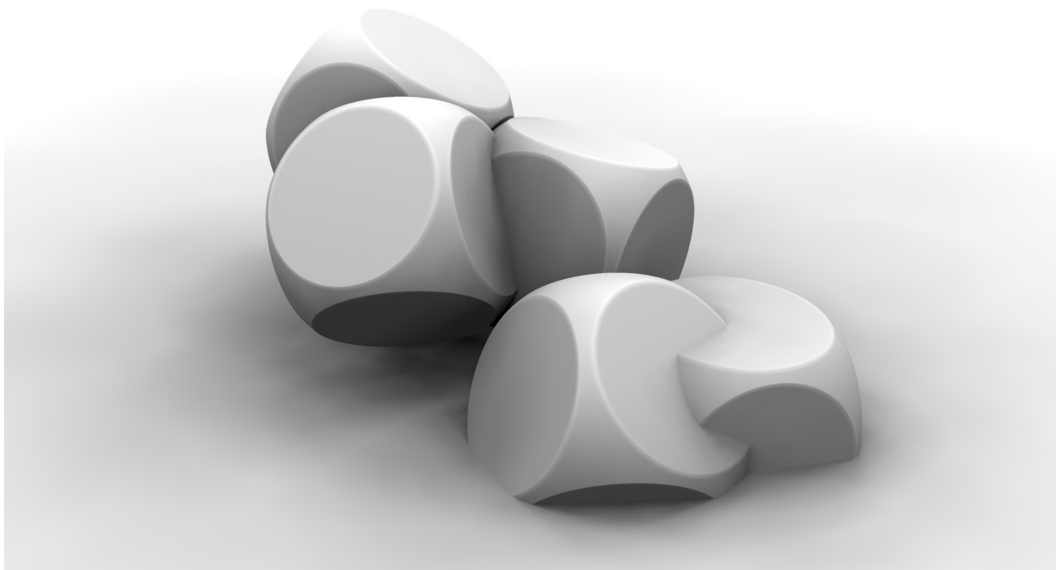

Figure 2.4: AO and soft shadows visualized, a scene by mhnewman [19]

Chapter 3

Project overview

3.1 Components

The program is designed as an example Unity project and a single, embedded Unity Editor package. It is open-source and available online on a publicly hosted GitHub repository [20].

The project includes:

- A sample project with exemplary SDF scenes demonstrating the capabilities of this tool.
- A collection of versatile C# classes for working with Abstract Syntax Trees of HLSL and Shaderlab (Unity's own Shader DSL) languages. The supplied classes model a subset of grammars and contain only the necessary utilities for working with the aforementioned languages.
- An original, embedded sub-project implementing a syntax generator using the Roslyn C# compiler API, used to automatically generate necessary AST classes based on the aforementioned, properly annotated C# grammar classes.
- A collection of Unity C# component scripts for generating and controlling shaders, interacting with and composing primitives using operators, and handling assets in the Unity editor.
- A set of documented, ready-to-use HLSL include files defining functions and utilities for working with SDF scenes. This set of tools can be used with the provided components or as standalone functions for writing custom shaders and extending the editor with new components, operators, and effects.

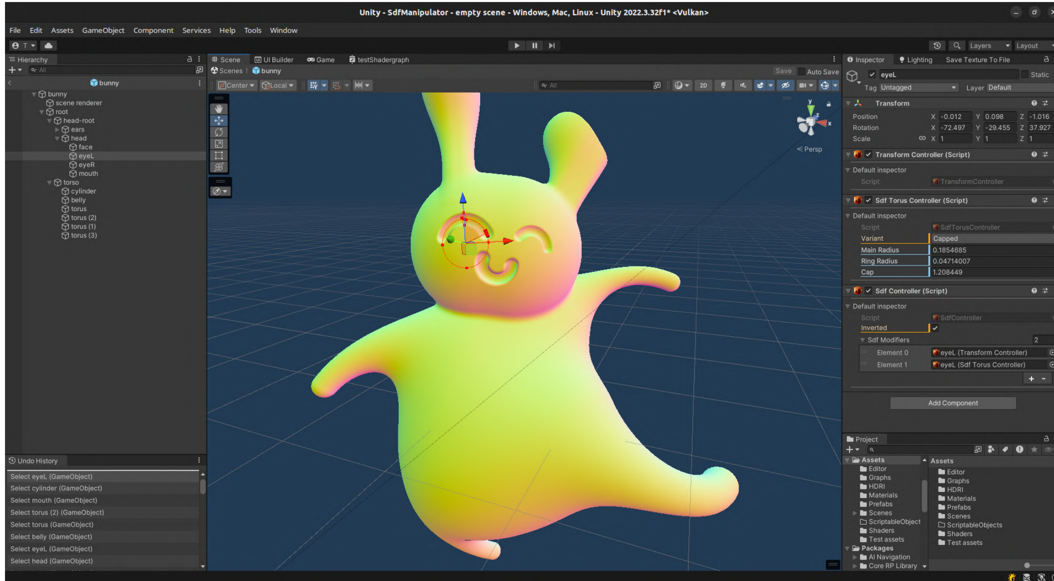

Figure 3.1: Prefab stage serves as an editing environment for the SDF scene. Game object hierarchy and attached modifier components define the scene structure. The scene view displays gizmos of the currently selected object's modifiers.

3.2 SDF scene generation

SDF shaders are generated by an `SdfScene` component attached to the root game object of a prefab. The structure of an SDF scene can only be generated and edited within the prefab. To create SDF scenes, use the `Create > SDF > Scene Asset` context menu option in the Project view.

The generated SDF scene prefab asset will include:

- A material and shader attached as a sub-asset to the prefab, which is regenerated when the scene requires regeneration.
- A main game object with an `SdfScene` component, a `root` child that serves as the origin for the SDF scene, and a `scene renderer` game object with a single mesh using the generated material. The `scene renderer` displays the game object and can be resized independently to encompass the whole scene without distorting the raymarched space.

3.3 Adding objects to SDF scenes

Adding new objects to scenes can be achieved by selecting options from the `SDF` context menu in the hierarchy window when editing a prefab. SDF primitives and operators are represented using subclasses of the `Controller` component, which

handles unidirectional communication with the `SdfScene` by sending events when properties change.

Basic primitives should be automatically recognized by the scene if they contain a subclass of the `SdfController`. A basic, positioned primitive in the scene will be detected if it contains the following components:

- A `TransformController` for transforming the space to position a primitive relative to the root. The purpose of this component is to react to the changes of a game object position and emit events used by `SdfScene` to update shader properties and position primitives.
- A subclass of `SdfPrimitiveController` which defines SDF primitives. Some examples include `SdfBoxController`, `SdfSphereController` or variants and `SdfTorusController`.
- An `SdfController` which references an ordered list of SDF operations, in the simplest case referencing attached aforementioned components in this exact order. Additional controls in the inspector allow control of several properties of the SDF, for example if it should be inverted or not. This component provides a definition of an SDF function a generator can attach to the generated shader.

Adding modifiers and primitives is achieved by adding a subclass of the `SdfController` to the game objects, filling in the required data, and referencing them in the appropriate components. The order of operations influences how the operators affect the generated scene. Controllers which implement the `IModifier` interface must agree on the input and output data types. Errors should be reported to the console when requirements are not met or the application of modifiers is impossible.

3.4 Custom controller editors

Custom editors for controllers provide responsive inspectors for controllers and functions for controlling modifier properties using scene gizmos. Inspector properties are divided into:

- Structural properties signalling when a shader regeneration is required, for example when a torus type has been changed from regular to capped, or when a primitive has been inverted.
- Runtime properties signalling the need to update material uniform values in real-time.

The generated material exposes controllable properties in the material inspector. Raymarching settings such as step count, surface hit epsilon, ambient occlusion steps, debug rendering modes, and other shader properties can be adjusted there.

Basic inspectors are generated automatically for subclasses of `Controller` and they autonomously bind to observable properties in controllers if properties are properly defined. Detailed requirements are documented in the `ControllerEditor.cs` file.

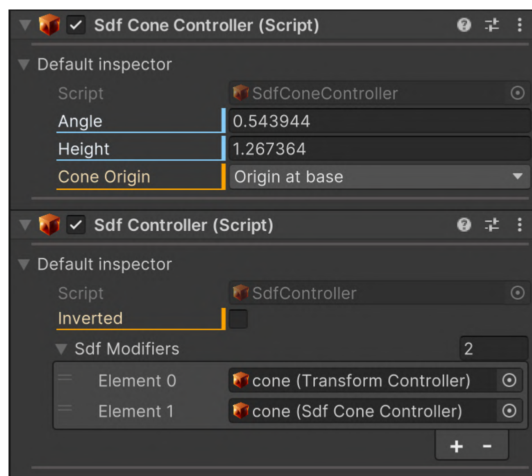

Figure 3.2: Two automatically generated controller inspectors. Orange fields indicate structural properties, blue fields indicate runtime properties.

3.5 SdfScene inspector

The `SdfScene` inspector depicted in Figure 3.3 includes:

- `Rebuild Shader` button allows you to manually rebuild the shader in the prefab editing environment, useful for debugging. This feature is inspired by the similar capability of the `ShaderGraph` asset from the Unity Shader Graph package.
- `Open Generated Shader` button displays the generated shader code. This code is built using a similar approach to the Shader Graph.
- `Diagnostics` section, displaying important information, errors and warnings.
- `Control` section, referencing asset used by the `SdfScene` to control material properties in real time.
- `Asset` section allows you to reference assets, which will be regenerated by the script whenever shader code changes. Asset in this section can be left empty.
- `Generation` section, allowing the user to select used shader preset and generator settings. Presets should be detected automatically during Unity domain

reload if they are derived from the `ShaderPreset` class. Generator settings are detected and displayed automatically by Unity, and can be initialized by shader presets.

Interactivity is achieved by exposing specific shader variables as uniforms, with the `SdfScene` component updating these uniforms as needed. Shader specific properties are handled and automatically detected using the `com.unity.properties` [21] package.

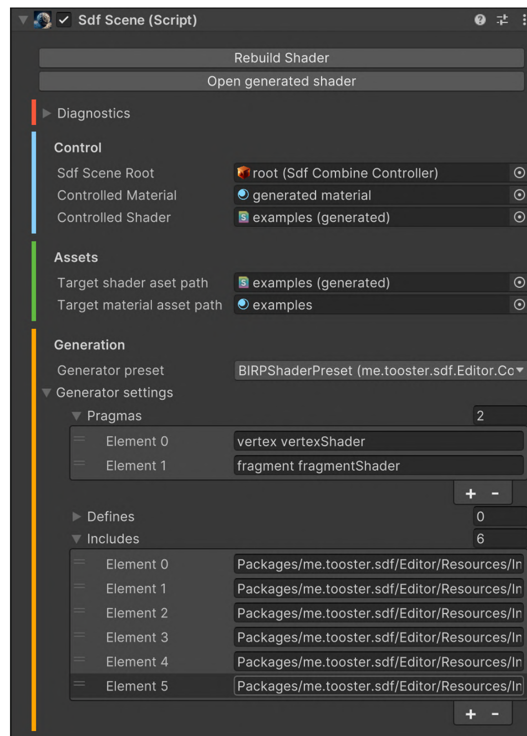

Figure 3.3: `SdfScene` component inspector, the root of any SDF scene.

3.6 Instantiating SDF scenes

Instantiating SDF scenes can be done by instantiating the SDF scene prefab. Gizmos are provided to show helpful information visually, such as green and red gizmos indicating normal or inverted primitives, and colored lines indicating combined controllers and their operations.

Multiple scenes can be instantiated at the same time and integrated into regular Unity scenes with normal mesh geometry thanks to proper handling of the depth reads and writes. Users can extend default generators and include files to add support for advanced image effects, alternative rendering pipelines and others.

3.7 Configuring generated materials

Generated shaders provide various customization options through the adjustment of material properties. The simplest way to modify these properties is by using the inspector of the generated material. Additionally, material properties can be controlled via standard Unity APIs, such as updating uniforms or toggling keywords from C# scripts. The default shader generator included in this tool exposes several useful properties for adjusting material settings such as raymarching steps and distance limits, enabling depth buffer writes and Z-tests, and controlling ambient occlusion strength to name a few.

3.7.1 Debug modes

To assist with the debugging of raymarching shaders, several debug modes have been added to the default generated shader. The functions are defined in the `raymarching.hlsl` and `debug.hlsl` include files. Controls for switching modes are available through the inspector of the generated material.

The debug modes include visualizations for:

- **Default Material Mode**: Displays the standard material shading.

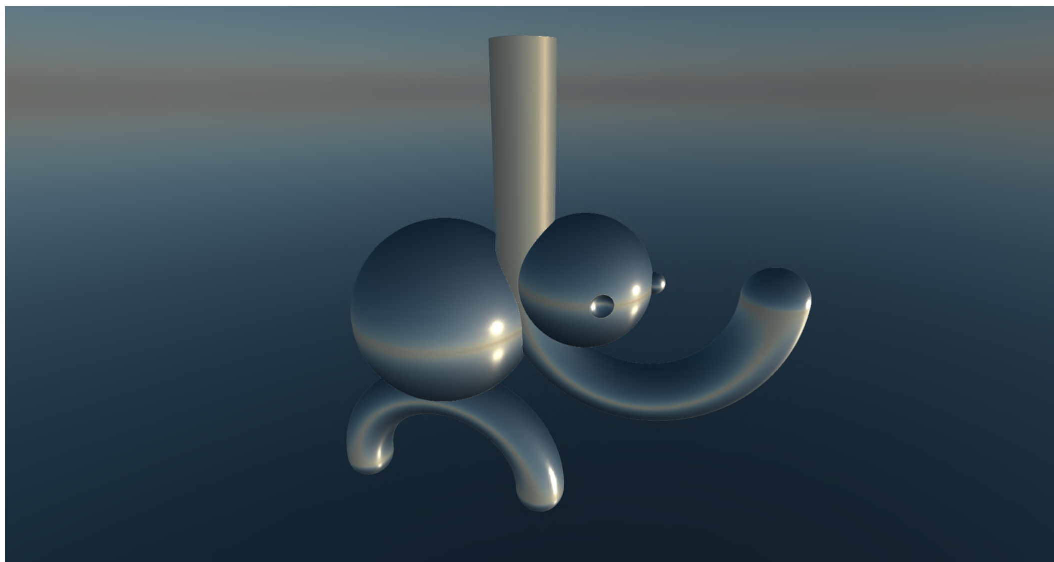

Figure 3.4: A figure model with material visualized.

- **Albedo**: Shows the base color of the material.
- **Primitive ID**: Visualizes the unique identifier for each SDF primitive.

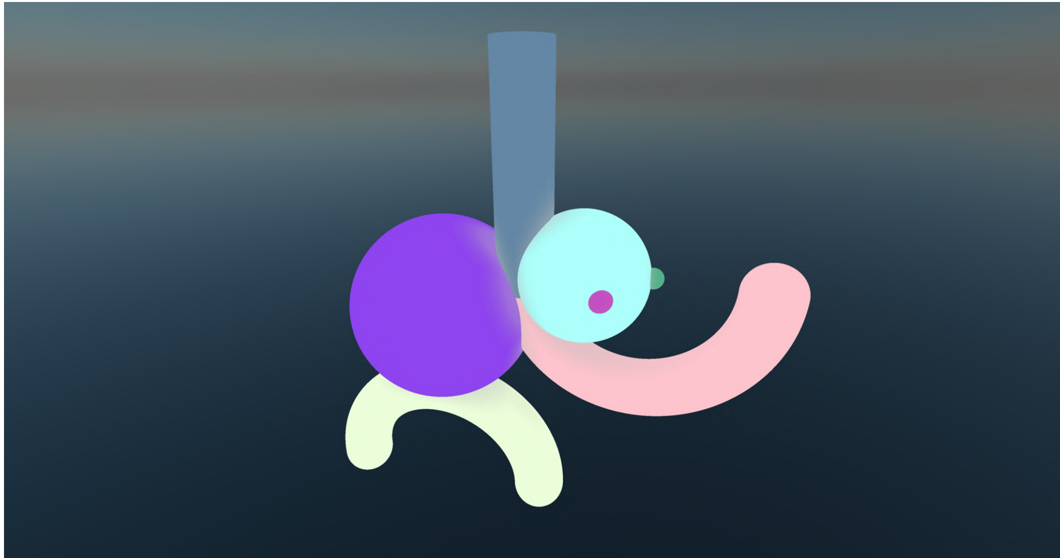

Figure 3.5: A figure model with primitive ID visualized.

- **Skybox Data**: Displays the skybox data used in the scene.
- **Calculated Normal**: Shows the surface normals calculated during raymarching.

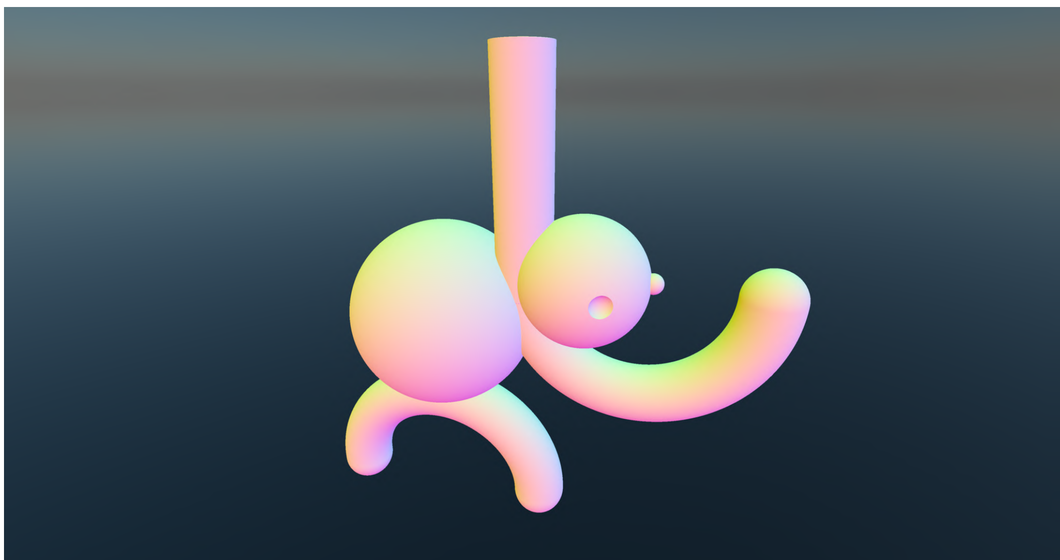

Figure 3.6: A figure model with calculated normal visualized.

- **Amount of Raymarching Steps**: Visualizes the number of steps taken by the raymarching algorithm.

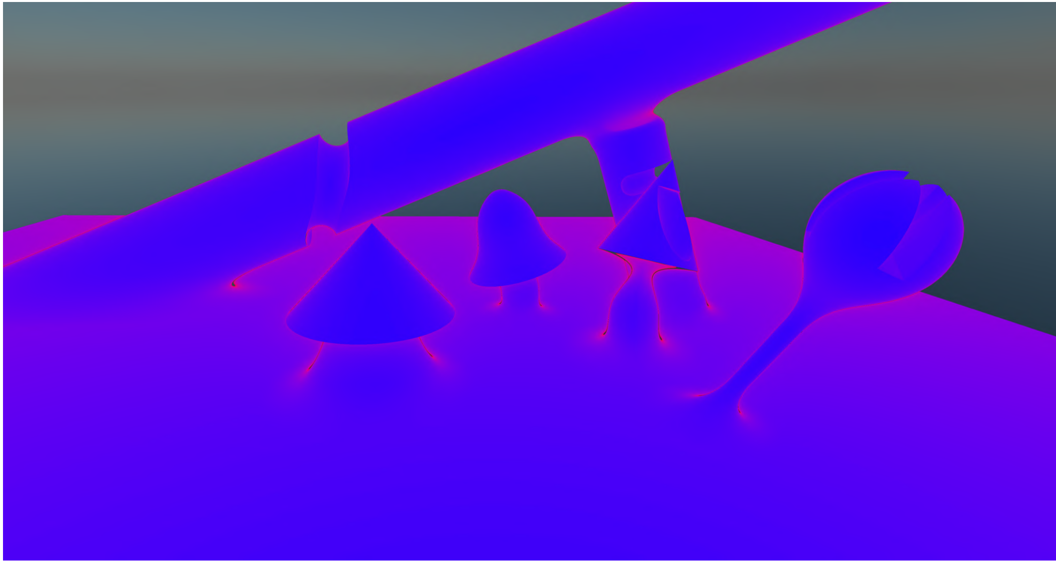

Figure 3.7: Visualization of the step count of the raymarcher. Blue indicates fewer steps, while red indicates more.

- **Depth**: Shows the depth information of the rendered scene.
- **Occlusion Factor**: Displays the ambient occlusion factor, indicating the shading influence of nearby geometry.

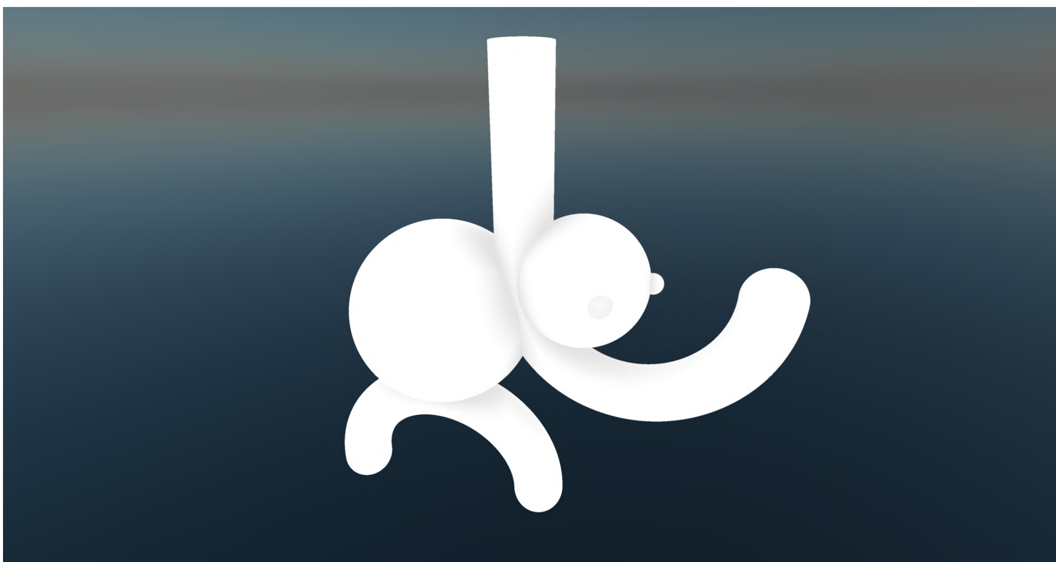

Figure 3.8: A figure model with ambient occlusion visualized.

- **World Space Grid**: Displays a grid of world space positions for better spatial understanding.

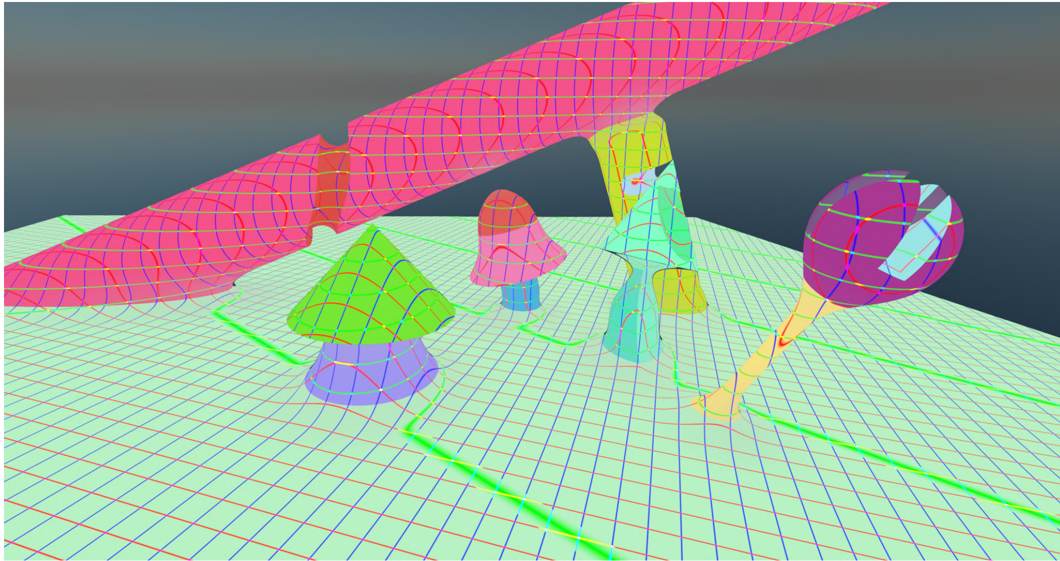

Figure 3.9: A complex scene with many primitives and operators, visualized with albedo debug mode and a world position grid.

3.7.2 Gizmos

To assist in the process of creating scenes interactively, a selection of gizmos and handles have been implemented in the package. An example gizmo for controlling a capped torus is displayed below in Figure 3.10. When an object with an `SdfTorusController` is selected, a gizmo for the major radius, minor radius, and cap angle is shown. Dragging the gizmo with the mouse or modifying the values in the inspector updates the values in the material in real-time, providing instant visual feedback.

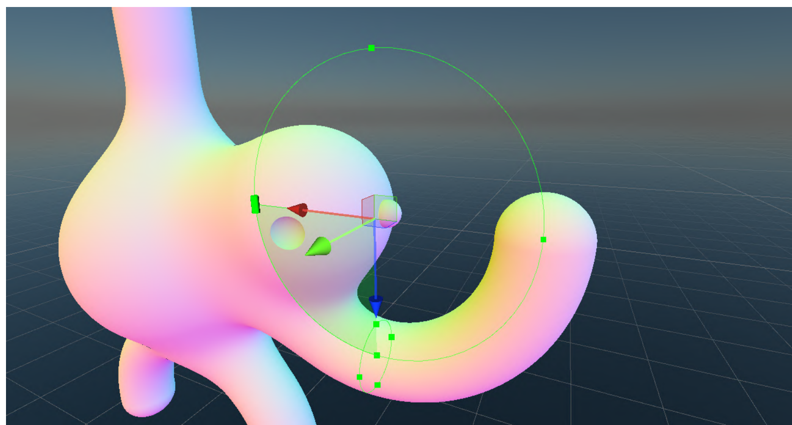

Figure 3.10: A capped cone gizmo. Dragging handles allow adjusting major radius, minor radius and the cap (cutout) angle.

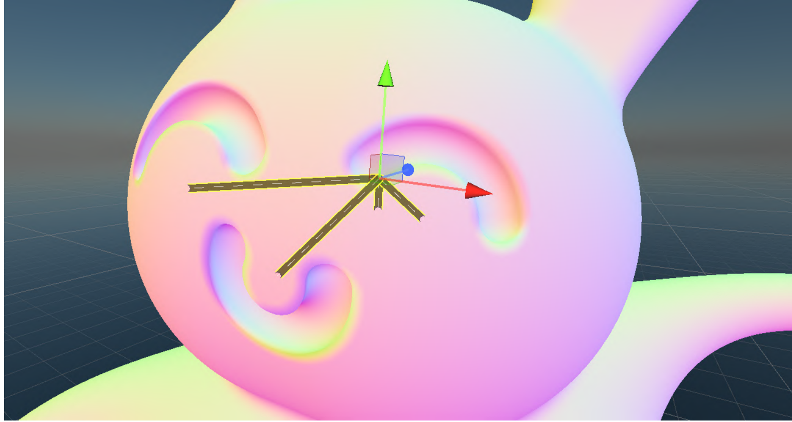

Figure 3.11: `SdfCombineController` in intersection mode displays yellow lines pointing to combined children.

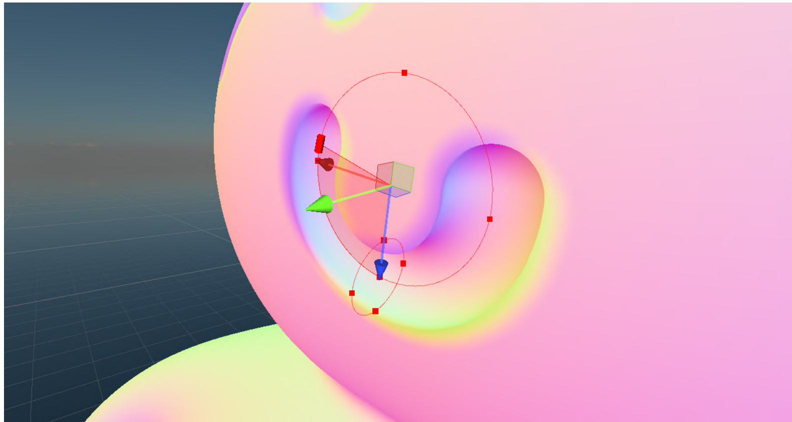

Figure 3.12: A capped torus gizmo turns red if the primitive is inverted.

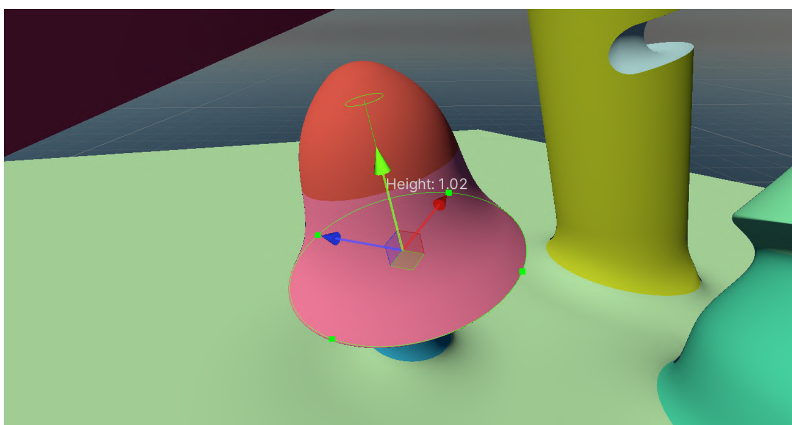

Figure 3.13: A cone scene gizmo. The tip and base radius handles can be dragged to adjust cone sizes.

3.8 Examples

This section features a collection of scenes crafted with the assistance of this tool. All scenes have been created manually and rendered in 2K inside the editor. Additionally, some scenes have been recreated based on publicly available online resources such as womp.com [2].

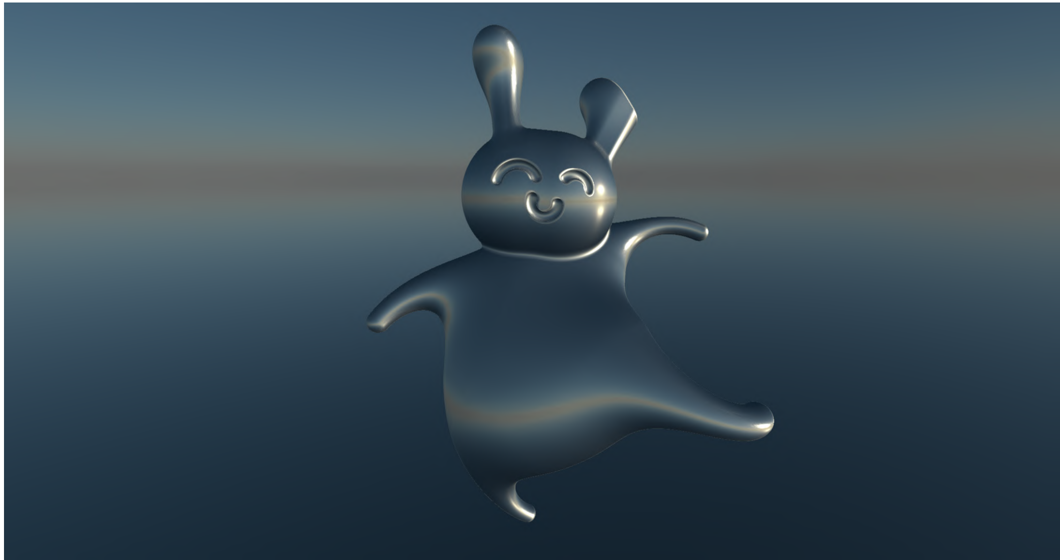

Figure 3.14: HDRI skybox shading applied to a bunny model.

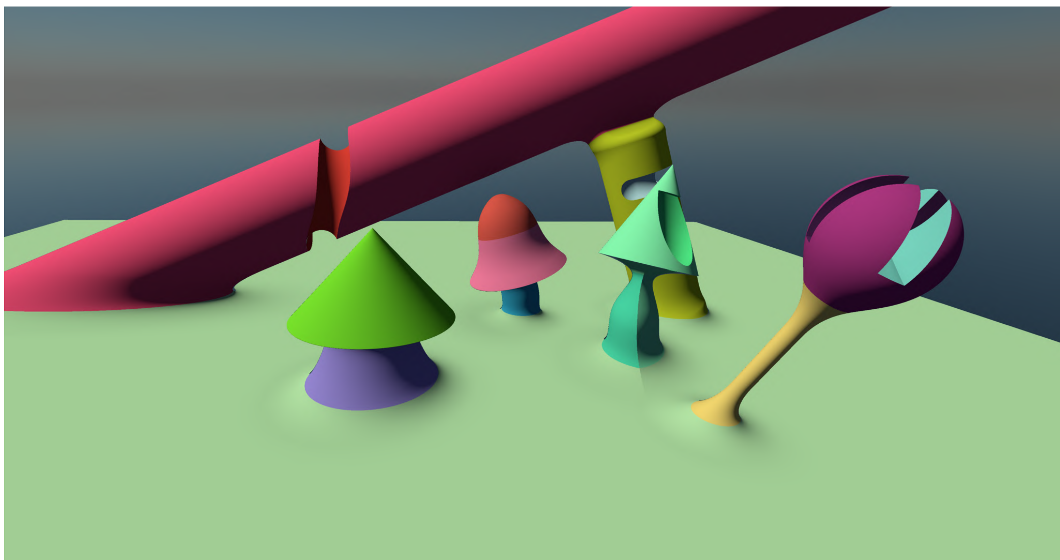

Figure 3.18: A complex scene shaded using primitive ID and basic Lambert shading model respecting Unity light.

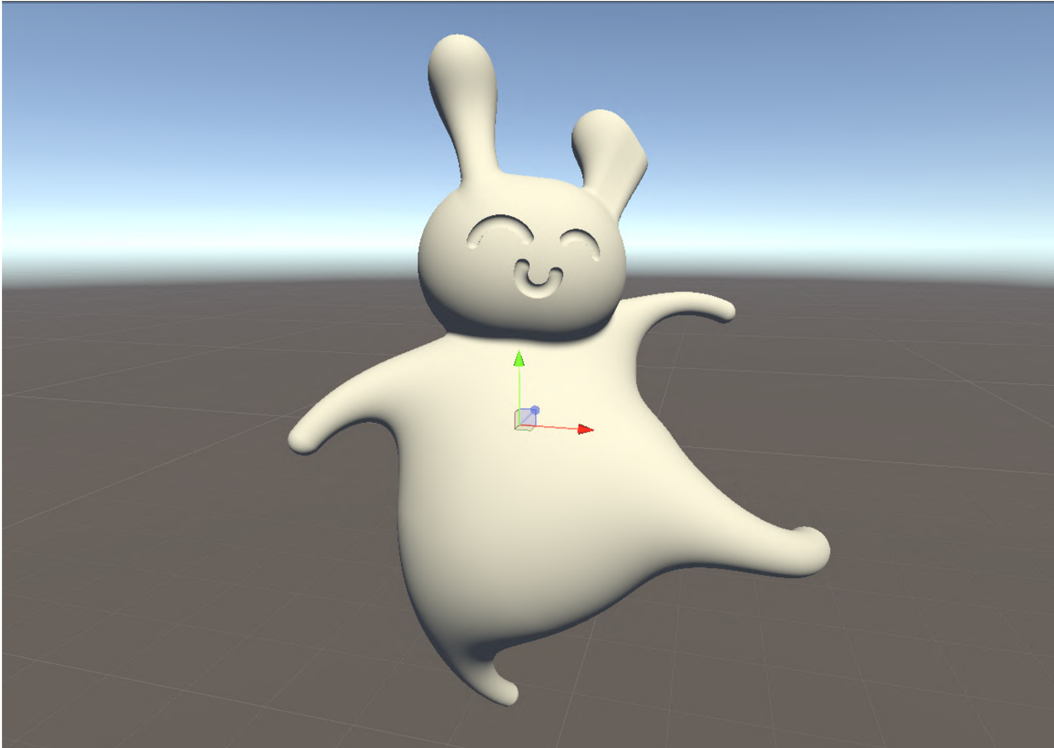

Figure 3.15: A bunny model created with the help of this tool.

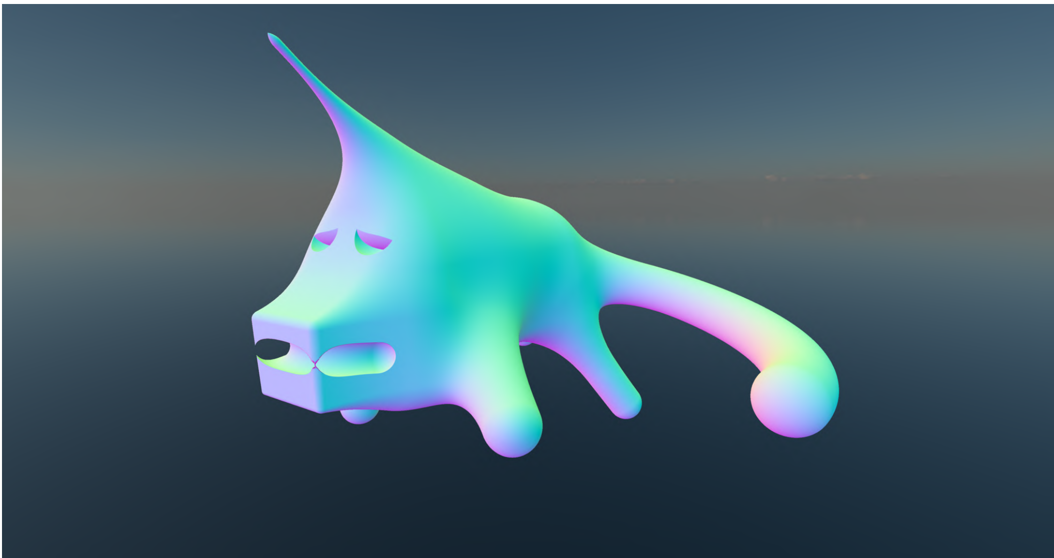

Figure 3.16: An uncanny depiction of a unicorn created in under five minutes.

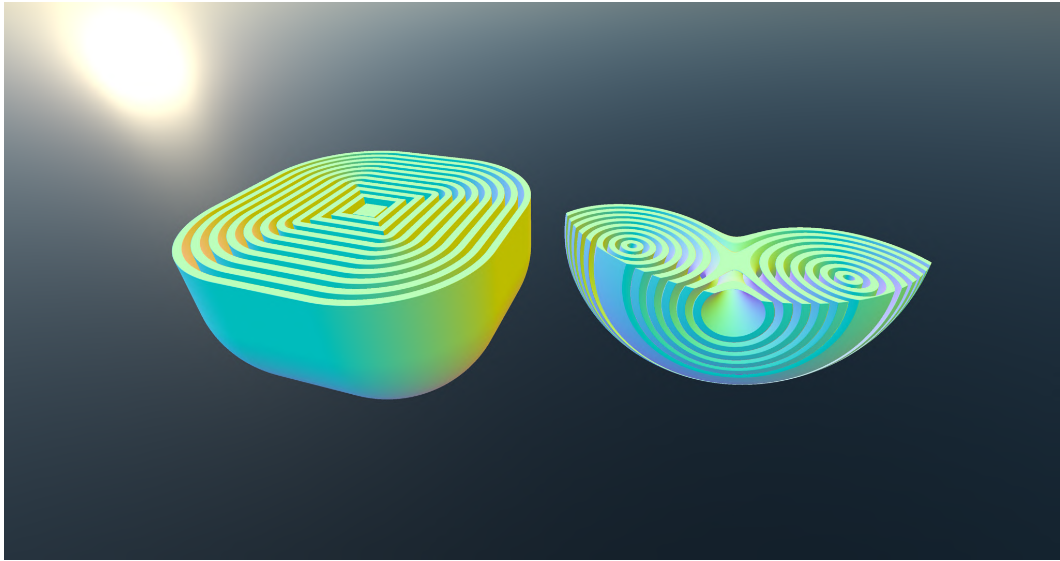

Figure 3.17: Layered onion operator applied to a rounded box and a cut torus

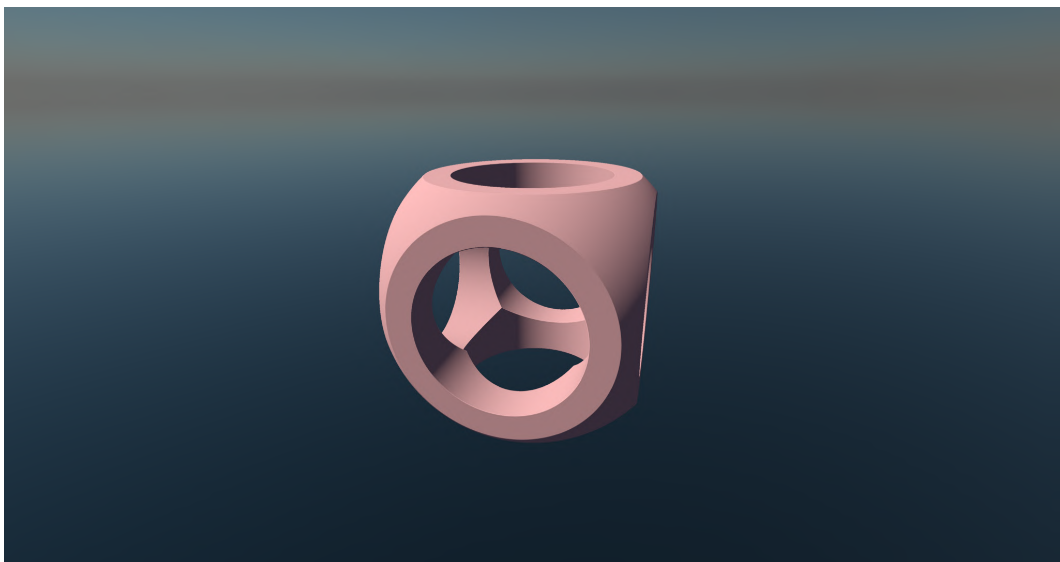

Figure 3.19: Constructive Solid Geometry example: rounded cube with cutout rounded cross.

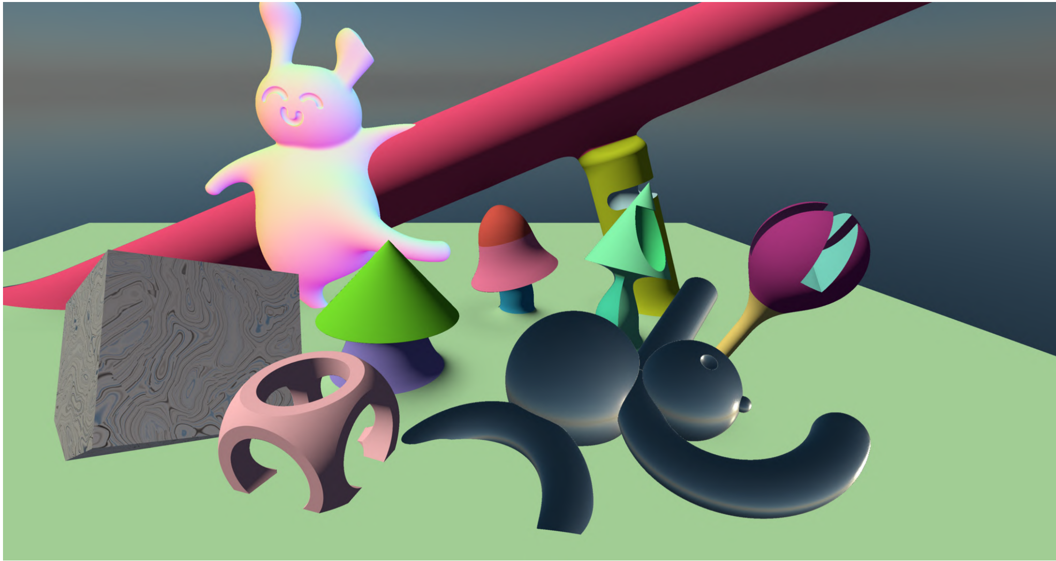

Figure 3.20: Multiple SDF scenes and regular unity mesh geometry (gray box on the left) rendered in a single Unity scene using one camera.

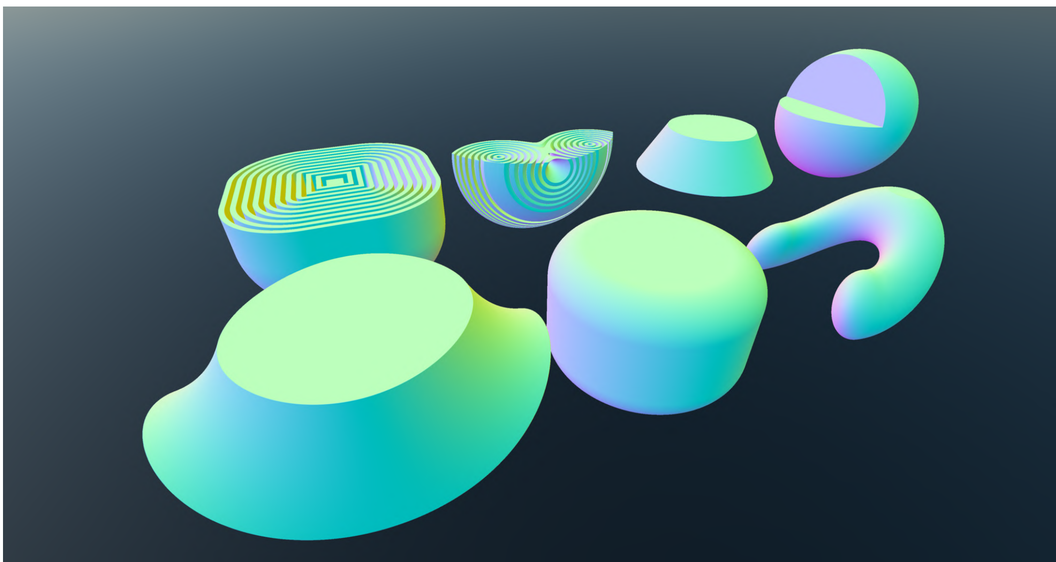

Figure 3.21: Primitives and operators can be combined, to create complex shader effects and behaviors.

Chapter 4

Implementation

4.1 Programming environment and tools

The program was created using the JetBrains Rider IDE, version `2024.1.2`. The project was developed and tested in Unity version `2022.3.32f1`. An original, embedded `Generators.dll` compiled shared library was used for generating AST helper classes. This library was built from the sources of the original sub-project `SDF/SyntaxGenerators` using the `msbuild` tool included with Rider IDE. Git was used for version control to efficiently manage and track changes throughout the development process.

4.2 Architecture and design

The tool is designed as a Unity Package and can be divided into two main components: the AST library and the Unity-specific classes and components for working with SDF scenes.

4.2.1 AST

To support the generation, assembly, formatting, and transformation of shaders, an AST library inspired by Roslyn's [22] Syntax API was developed. It is located in the `SyntaxGenerators` directory of the package. The design of the AST tools was influenced by the following decisions:

- The AST should be modeled as an immutable tree, providing means for transforming and rewriting parts of it easily through specialized syntax node visitors, such as syntax rewriters, visitors and walkers. This approach enables modifying the generated shader in an idiomatic way without crude string matching, commonly found in other shader generation tools.

- Immutable AST nodes facilitate the creation of declarative and functional code with fewer state-related problems.
- Formatting of the generated code is achieved using `HlslFormatter` and `ShaderlabFormatter` implementing a tree walker and visitor pattern operating on the AST.
- Strongly typed AST nodes help in generating valid HLSL and Shaderlab source code, reducing the likelihood of runtime errors.
- The library focuses on constructing syntax trees, not parsing them, as parsing is not a primary concern.
- AST trees are constructed bottom-up, but a lazy ancestor chain is created using `Anchors` during downward traversal. This ensures AST immutability while providing the flexibility of accessing parent nodes on demand. An alternative approach using the functional Zipper pattern may be considered for future work.
- Common AST structures, such as `SyntaxList` or `Literal`, are shared between languages without repetition, but are tagged with different languages to prevent mixing syntax nodes of different languages at compile time. This is achieved using generic node types and marker language interfaces.
- Source Generators are used to automatically generate the required syntax visitors and utilities based on minimal syntax node definitions.
- The syntax tree preserves most of the source information, including trivia (whitespaces, comments, preprocessor directives). The tree tries to support full fidelity on the best effort basis.
- Unlike Roslyn but similar to TypeScript, syntax nodes do not have trailing trivia, only leading trivia. This decision helped in mitigating numerous edge cases encountered when trying to implement syntax rewriters for a model with both leading and trailing trivia.
- C# records are used to provide better developer experience for immutable modifications using the `with` construction. Even though the current design may have small performance issues due to the usage and allocations of reference data types, the API is designed to allow easy migration to the more efficient `record structs` in more modern versions of C# yet to be supported by the Unity engine.
- Optimizations should be performed mainly when a bottleneck is detected. As long as the source code generation doesn't degrade interactivity, the optimization shall be a secondary priority.

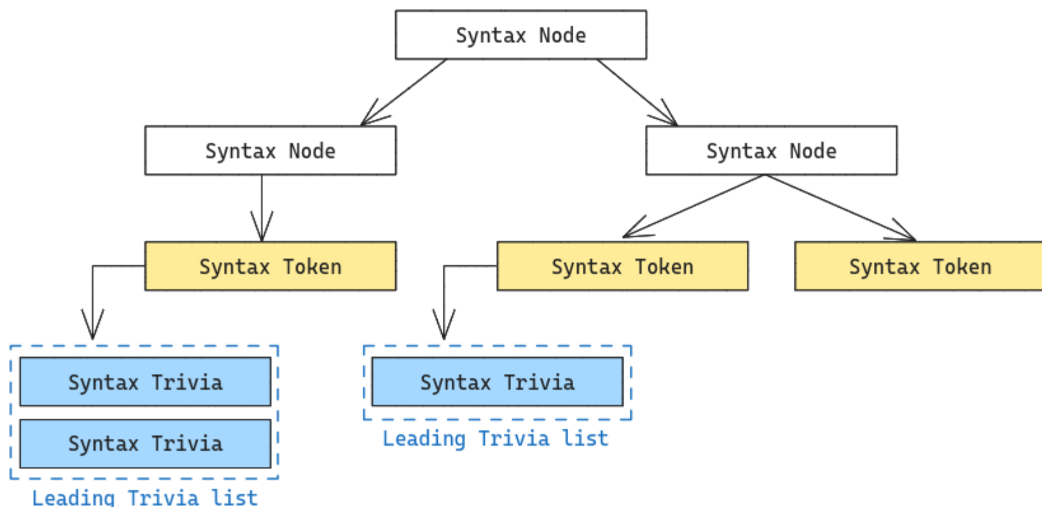

Figure 4.1: Conceptual model of a syntax tree

- Syntax nodes should closely mirror grammar rules of the languages, and should be structurally similar to nodes present in other HLSL tools such as compilers and parsers, allowing for easier future integration of such tools.

The structure and architecture patterns of the AST for HLSL and Shaderlab were inspired by Microsoft's Roslyn compiler design [22] and HlslTools by Tim Jones [23]. It is also worth mentioning a new, standalone parser library tailored for HLSL and Shaderlab [24], although this tool has been made public too recently to have influenced or been used in this work.

Design of AST nodes

AST nodes are designed as simple, immutable C# record data types. Any mutations to nodes are achieved using C# record class `with` syntax. Tree nodes are divided into syntax nodes (representing abstract parts of the syntax such as function definition statements or binary expressions), token nodes (representing concrete sets of tokens recognized by the language), and trivia nodes (representing auxiliary parts of the concrete syntax tree such as preprocessor directives, whitespaces, and comments).

This design was directly influenced by existing and widely used tools, such as the C# Roslyn compiler [22], HlslTools by Tim Jones [23], and the TypeScript compiler. Detailed explanations of this pattern can be found in the cited works.

Syntax nodes are internal tree nodes that do not directly represent the text — they only hold references to other syntax nodes or token nodes. Token nodes are the leaves of the tree and describe how tokens are transformed into text. Trivia nodes are owned solely by token nodes.

Listing 2 is an example definition of a syntax node representing a function definition of the HLSL language.

```
/// function definition with body, for example
/// int foo(float x, row_major float y : VPOS = 7.0f) {
///     return x + y;
/// }
[SyntaxNode] public partial record FunctionDefinition :
↪   Statement<hlsl>
{
    public Type                      returnType     { get; init; }
    public Identifier                id             { get; init; }
    public ArgumentList<Parameter> paramList      { get; init; }
        = new();
    public Semantic                  returnSemantic { get; init; }
    public Block                     body           { get; init; }
}
```

Listing 2: Syntax node defining a HLSL function definition.

Language grammars are defined using C# classes inside `AST/<language>` directories. The `AST/Syntax` directory defines base AST classes and common syntax, token and trivia nodes. A single, lowercase marker interface is generated for each child directory of `AST` by the `ShaderGenerator.dll` source generator. For details on source C# source generators refer to Roslyn and Unity documentation [25].

For the syntax generator to function correctly, records must be partial and annotated with either `[SyntaxNode]` or `[TokenNode]`. There are two types of trivia: `StructuredTrivia` and `SimpleTrivia`. Simple trivia only define text (for example, whitespace), while structured trivia hold a node to a syntax, such as a preprocessor `ifdef` syntax node.

Generated partial classes define acceptors for syntax visitors and a utility function to return children nodes in the exact order they are defined in the class. Syntax node children are defined using public properties returning subclasses of AST nodes and are init-only.

Some syntax nodes define implicit conversion operators to reduce the verbosity of the tree-building API. A universal "Mapper" (a tree rewriter) and "Formatter" inheriting from it are defined for the common language, which are subclassed by `HlslFormatter` and `ShaderlabFormatter`, allowing for a unified, centralized formatting logic.

A selection of utility extension functions is defined in `Navigation` and `Extensions` for working with AST. An `Anchor` class, an overly simplified version of a functional Zipper, is defined to act as a stateful wrapper for syntax nodes, created

by rewriters and walkers during the descent.

The majority of the code uses C# pattern matching to write AST transformation rules.

```
new FunctionDefinition() {
    returnType = (Predefined)Constants.ScalarKind.@float,
    id = (Identifier)"sdfFunction",
    paramList = new Parameter {
        type = SdfData.pData.typeSyntax,
        id = SdfData.pParamName
    },
    body = new Block { /*...*/ },
}
```

Listing 3: Example of creating a syntax node representing a HLSL function definition using the AST API.

4.2.2 Generation

Shader generation is performed mainly by the following types of components:

- Modifiers inheriting from `Controller` and implementing `IModifier` interface, acting as modular, composable blocks and exposing C# API for controlling and updating material properties in runtime.
- `SdfScene`, which keeping track of the scene's state, listens to events emitted by Controllers, and updates the materials or shaders when needed.
- `RaymarchingShaderGenerator` derived from `Processor` which serves a disposable, stateful generator composing the final shader source code by evaluating modifiers and processing required `SdfScene` data.
- `ShaderPreset`, which plays a role of a configurable generator factory, enabling users to easily hook into the generation pipeline by providing their own generators and configuring them.

`SdfScene`

The `SdfScene` has been modeled using prefabs. This design choice was dictated by the need to keep shaders unique per SDF scene, as each one generates different shader code, and shaders are compiled during project build time. Using prefabs allows for modular and reusable architecture while maintaining the uniqueness required for each shader. Additional benefit of using prefabs is exposing ”a controllable skeleton”

of the scene whenever it's instantiated. This is conceptually similar to exposing and using rigged meshes when animating models.

The `SdfScene` acts as a scene root and collects important data, taking care of assigning unique identifiers to each component, tracking properties, and sending uniform updates to the controlled material when needed. It serves as the central hub for managing and orchestrating the various elements that make up the SDF scene, ensuring consistency and proper functionality.

`Controller` and `Modifier`

Controllers are components inheriting from the base class `Controller` and implementing a `Modifier` interface. The `Modifier` interface is used to recognize the input and output types, and they are modeled to be like modular bricks, which can be stacked and evaluated serially by the processor. This modular approach allows for flexible and dynamic composition of SDF operations.

Modifiers define a single `Apply` method, which defines the behavior of the modifier. This method accepts a single input object and returns a single output object. There are no restrictions on the data type aside from It is up to the modifier to correctly interpret and handle this data. Unlike Unity's ShaderGraph, which can handle only a fixed set of types (HLSL primitive types only), this tool can bundle structures and even functions as data. This means that functions and operations are treated as first-class citizens in this model, providing greater flexibility and power in shader creation.

Modifiers are designed to be mostly stateless, which significantly simplifies the logic. By minimizing statefulness, the code becomes significantly simpler, resulting in fewer bugs and making maintenance easier. This stateless nature allows for more predictable behavior and simpler debugging. Due to the limitations of the Unity engine it wasn't possible to implement true stateless architecture, nevertheless it was used on a best-effort basis.

During processing, modifiers can emit additional requirements which must be handled by the processor. Some examples include the definition of a function that has to be included in the shader to make the evaluation work or an inclusion of a specific HLSL file. This allows for complex dependencies and functionalities to be managed seamlessly.

Modifiers can be combined into stacks, which themselves are modifiers. Modifiers in a stack must agree on input and output types. This model is more powerful than the a simple evaluation model used by Shader Graph, because it allows using modifiers as values. This stacking capability allows for complex operations to be built up from simpler ones, enhancing modularity and reusability.

Modifiers can depend on and reference other modifiers to apply advanced mod-

ifications. For example, an Elongate modifier can take a modifier that transforms VectorData into ScalarData and, assuming it's the SDF function, applies elongation. This dependency system allows for intricate shader effects to be created.

The decision for that architecture was dictated by the definition of some SDF operators. For example, a mathematically exact elongation operator is defined as a circumfix operation that transforms the space before evaluating an SDF and modifies the returned distance value:

```
// elongation is a circumfix operation on a primitive.
float elongate_exact(float3 p, float3 size) {
    float3 q = abs(p) - size;
    return SDF(max(q, 0.0)) + min(max(q.x, q.y, q.z), 0.0);
}
```

Listing 4: An implementation of exact elongation operator in HLSL pseudocode.

Without the capability to treat SDFs as first class citizens, this operation would have to be defined as two nodes in a Shader Graph evaluation model: the first one evaluated before and the second one evaluated after an SDF.

Event driven architecture of Controllers

Controllers define two important events used for communicating with the SdfScene: PropertyChanged and StructureChanged. The first event indicates that a non-structural shader change occurred to a property, meaning that the updated value should be sent to the shader. The second event informs that an underlying shader definition of the operator changed, requiring regeneration and recompilation. These events ensure that the shader remains up-to-date with the current state of the scene.

Controllers implement the standard INotifyPropertyChanged interface to emit property change events. This interface is commonly used in .NET for data binding, making it a familiar and robust choice for notifying the system of property changes.

The com.unity.properties [21] package is used to collect and update properties generically during runtime. Compatible properties must be annotated with [CreateProperty] and either [ShaderProperty] or [ShaderStructural], and the controller must be a partial class annotated with [GeneratePropertyBag] for compile-time generation of class property visitors. This allows for efficient access and visitation of properties by SdfScene, streamlining the property management process.

Controllers send required events when they detect changes. These changes can include modifications to children or parents, renaming, or changing the order or data of components. Due to the multitude of events that Unity can handle, it is possible that some events have not been handled, but regular use of the tool has not shown

any major errors. This robust event-handling system prevents tight component coupling and ensures that the SDF scene remains responsive and up-to-date.

To make default Controller inspectors properly bind UI to controller data, properties and their backing fields must be named the same, with the backing field starting with a lowercase letter and the property with a capital letter. This naming convention allows for the automatic generation of inspector classes. If the need arises, a custom inspector can be created, but for the most part, a default generated Controller inspector should suffice. This convention simplifies the code and reduces the likelihood of naming conflicts or errors. Several C# attributes have been implemented to improve the process of automatic inspector generation.

Handling errors

Unmet modifier requirements are reported as errors to the console. For example, when a combine controller does not have any children, an error is logged. This error reporting helps identify and resolve issues quickly, improving the development process.

Some errors are displayed in the inspector of the `SdfScene` in the "Diagnostics" foldout. This provides a convenient and accessible way for developers to view and address issues directly within the Unity Editor, enhancing the debugging and development experience.

4.2.3 Generators

Generators implement the `Processor` interface to consume the data collected by the `SdfScene`, provide an evaluation context for modifiers, and create the final shader code. Generators are instantiated on demand by classes derived from `ShaderPreset` and are disposed of after generation. Presets define methods and data necessary for instantiating generators. Unity detects and registers shader presets automatically after a domain reload. Users can implement their own presets and generators to target alternative rendering paths, such as forward or deferred rendering, and rendering pipelines, such as Unity's URP (Universal Render Pipeline) or HDRP (High Definition Render Pipeline). This modularity and extensibility make it straightforward to adapt the tool to a wide range of rendering requirements.

4.2.4 Problems

During the implementation, numerous problems were encountered and resolved. Some noteworthy issues are listed below:

String based generation

The initial approach to generating shaders involved solutions similar to uRaymarching and Shadergraph—simple string concatenation or a simple string templating language. This method proved difficult for several reasons:

- The need to develop, integrate, and learn an additional templating language.
- String concatenation is challenging when dealing with multiple dependencies, as HLSL SDF functions must be treated as first-class citizens during generation. The data dependencies are often bidirectional, and string concatenation is not powerful enough without a lot of messy code.
- The code is either littered with formatting directives or the generated shader source is unreadable.

The AST library was created due to the lack of existing, standalone HLSL and Shaderlab tools for parsing, syntax, and semantic models at the time of developing this project.

Roslyn's red-green tree model

An attempt was made to implement an AST model similar to Roslyn's red-green trees [26]. However, the complexity of implementation and the redundancy of maintaining internal and public syntax for representing immutable bottom-up and lazy top-down syntax trees proved too difficult to manage. Instead, lazy references to parents were used during downward tree traversal to reduce the amount of maintenance code and make the traversal explicit.

Ray generation

The implementation of the raymarching shader required a careful design to accommodate both perspective and orthographic projections. While numerous online sources provide guidance on raymarching, their explanations often do not translate universally to orthographic projection. The goal was to create a raymarching solution that works seamlessly for both projection types.

Several approaches were tested for generating rays from the camera. The first approach used projection and view matrices and their inverses between the vertex and fragment shader, but this method required calculating expensive matrix inverses and sometimes produced invalid results.

Another approach interpolated direction vectors between orthographic and perspective rays based on the active projection type. Although this method worked

well in the fragment shader, it produced artifacts around triangle seams when implemented in the vertex shader to leverage automatic GPU interpolation for calculating fragment ray directions.

The final solution was an improvement to the previous technique. It involved disabling perspective correction on the generated ray direction in the vertex shader using the HLSL `noperspective` directive and performing ray direction normalization only in the fragment shader after the GPU performed rasterization. The final version of this technique is present in the `vertexShader` and `fragmentShader` functions inside the `debugBaseShading.hlsl` and `raymarching.hlsl` include files found in the package.

Lack of documentation and problems with the Unity engine

The lack of proper Unity documentation and internal engine bugs significantly hindered the work on implementation of several features. Some examples include:

- Poorly documented asset pipeline with difficult to work with APIs: The current design of modelling the scenes as Prefabs could have been implemented with custom asset types, importers and a standalone editor window, similar to Shader Graph. However, the scope of the project was so vast that it would have been unrealistic for a single developer to complete within a reasonable timeframe.

- Lack of detailed documentation for rendering APIs, reliance on implicit shading knowledge and a collection of seemingly disjoint but stateful shader files: Unity doesn't fully explain what its core shading functions do and how to hook into the rendering pipeline in the shader. Some things, like integrating with the standard unity shading model and material pipeline in vertex and fragment shaders, were either very hard to understand, integrate or they were skipped altogether.

- Support for controlling the shader compilation pipeline: Due to the lack of explicit API for manually compiling shaders, the recompilation process depends solely on internal unity event events. The stateful design of Unity engine can sometimes lead to weird problems when the state in memory is not synchronized with the state in the editor. One such example is occasional problems with refreshing of the prefab editing stage when a prefab updates and stale references to non-existing assets in the inspector windows.

- Scene picking: Unity doesn't explain how to integrate custom scene picking for manually rendered objects. This forced the design of the tool where objects can be picked only at their center by their icon gizmo in the scene. Several approaches were tested for true pixel-based scene picking, but there has been no success in implementing it.

- Editor problems on a Linux platform: Editor often encountered fatal crashes due to internal bugs when developing under Linux. The lack of support for integration with rendering debuggers (for example RenderDoc) proved to be a major difficulty in the initial stages of the development process.

- Occassionally, several internal Unity bugs may trigger false error message reports, which do not affect the tool's behavior.

Chapter 5

Summary and conclusions

This thesis presented a comprehensive tool designed for the Unity engine, simplifying the creation and editing of scenes using Signed Distance Fields (SDFs). By providing an intuitive and interactive user interface, the tool eliminates the need for manual shader programming, allowing users to design complex geometries visually. This work has aimed to lower the barrier to entry for developers and artists, fostering greater experimentation and innovation in SDF-based rendering within the Unity engine.

5.1 Future work

There is a vast space for improvement and exploration regarding the tool. Future work could include improving the performance of the tool and reducing the memory footprint, implementing rendering optimizations such as constant buffers and batching, integrating with existing Hlsl and Shaderlab parser libraries to facilitate fully featured AST support, supporting more rendering pipelines and paths, integrating with Shadergraph and VFX graph, exporting scenes as meshes, better lighting support, support for GLSL language, improving the scene editor tools and error reporting, optimizing asset generation process, implementing alternative techniques, primitives and operators, integrating VR rendering or even decoupling the tool from Unity and releasing it as a standalone web app. Due to difficulty, engine bugs and an already broad scope of this thesis, proposed topics should instead become the subject of the future work.

Bibliography

[1] Shadertoy.com, online community and tool for creating and sharing shaders through webgl.
https://shadertoy.com/
Accessed: 2024-06-10.

[2] Womp.com, free and beginner-friendly 3d modeling software to design easy 3d art in real time.
https://womp.com/index
Accessed: 2024-06-10.

[3] Unbound.io, a new engine built around the magic of creating with sdf shapes.
https://www.unbound.io/
Accessed: 2024-06-10.

[4] Hecomi. uraymarching, a discontinued tool with raymarching shader templates for rendering sdfs.
https://github.com/hecomi/uRaymarching
Accessed: 2024-06-10.

[5] Long Bunny Labs. Mudbun, a volumetric vfx mesh tool for unity.
https://longbunnylabs.com/mudbun/
Accessed: 2024-06-10.

[6] Ming-Lun ”Allen” Chou. Clayxels, a discontinued, experimental raymarching sandbox for raymarching in unity.
https://github.com/TheAllenChou/unity-ray-marching
Accessed: 2024-06-10.

[7] J. C. Hart, D. J. Sandin, and L. H. Kauffman. Ray tracing deterministic 3-d fractals. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH ’89, page 289–296, New York, NY, USA, 1989. Association for Computing Machinery.

[8] Inigo Quilez ”iq”. Rendering worlds with two triangles.
https://web.archive.org/web/20240503035829/https://iquilezles.org/articles/nvscene2008/rwwtt.pdf
Accessed: 2024-06-10.

[9] Inigo Quilez "iq". 'snail' shader on shadertoy.com.
https://www.shadertoy.com/view/ld3Gz2
Accessed: 2024-06-12.

[10] Ray marching and signed distance functions.
https://web.archive.org/https%3A//jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/
Accessed: 2024-06-10.

[11] NVIDIA. Gpu gems 2.
https://developer.nvidia.com/gpugems/gpugems2/part-i-geometric-complexity/chapter-8-pixel-displacement-mapping-distance-functions
Accessed: 2024-06-10.

[12] Johann Korndörfer, Benjamin Keinert, Urs Ganse, Michael Sänger, Simon Ley, Konstanze Burkhardt, Mario Spuler, and Jörn Heusipp. Hg_sdf: A glsl library for building signed distance functions, 2015.
https://mercury.sexy/hg_sdf
Accessed: 2024-06-10.

[13] Inigo Quilez "iq". 3d sdf functions.
https://iquilezles.org/articles/distfunctions/
Accessed: 2024-06-10.

[14] Inigo Quilez "iq". Smooth minimum for sdfs.
https://iquilezles.org/articles/smin/
Accessed: 2024-06-10.

[15] Inigo Quilez "iq". Numerical sdf normals.
https://iquilezles.org/articles/normalsSDF/
Accessed: 2024-06-10.

[16] Inigo Quilez "iq". soft shadows in raymarched sdfs.
https://iquilezles.org/articles/rmshadows/
Accessed: 2024-06-10.

[17] Anatole Duprat. Hemispherical signed distance field ambient occlusion.
https://www.aduprat.com/portfolio/?page=articles/hemisphericalSDFAO
Accessed: 2024-06-10.

[18] Inigo Quilez "iq". biplanar mapping.
https://iquilezles.org/articles/biplanar/
Accessed: 2024-06-10.

[19] mhnewman. 'sdf ambient occlusion' shader on shadertoy.com.
https://www.shadertoy.com/view/XlXyD4
Accessed: 2024-06-10.

[20] Maksymilian Polarczyk. Sdfmanipulator, a unity package for generating and manipulating raymarching shaders using sdfs (signed distance fields).
`https://github.com/T3sT3ro/SdfManipulator`
Accessed: 2024-06-10.

[21] Unity technologies. Unity properties package.
`https://docs.unity3d.com/Packages/com.unity.properties@2.1/manual/index.html`
Accessed: 2024-06-10.

[22] Microsoft. Roslyn, a .net compiler api.
`https://github.com/dotnet/roslyn`
Accessed: 2024-06-10.

[23] Tim Jones. Hlsltools, a visual studio extension providing enhanced support for editing high level shading language (hlsl) files.
`https://github.com/tgjones/HlslTools`
Accessed: 2024-06-10.

[24] pema99. Experimental unity shader parser.
`https://github.com/pema99/UnityShaderParser`
Accessed: 2024-06-12.

[25] Unity technologies. Unity source generators documentation.
`https://docs.unity3d.com/Manual/roslyn-analyzers.html`
Accessed: 2024-06-12.

[26] Eric Lippert. Persistence, façades and roslyn's red-green trees.
`https://ericlippert.com/2012/06/08/red-green-trees/`
Accessed: 2024-06-10.