



Assignment of bachelor's thesis

Title:	Ray Marching Scene Creator
Student:	Yevhenii Basov
Supervisor:	Ing. Petr Pauš, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Graphics 2021
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2024/2025

Instructions

The aim of this thesis is to create an application in Unity 3D to easily create and display 3D scenes using the Ray Marching method with a suitable user interface. The application will provide the ability to create parameterizable geometric primitives and various functions to combine them.

Assignment points:

1. Analyze the Ray Marching algorithm.
2. Analyze the possibilities of implementation in Unity3D.
3. Design a prototype application including user interface.
4. Implement the solution.
5. Perform appropriate testing.

Bachelor's thesis

RAY MARCHING SCENE CREATOR

Yevhenii Basov

Faculty of Information Technology
Department of computer grafics
Supervisor: doc. Ing. Petr Pauš, Ph.D.
May 16, 2024

Czech Technical University in Prague
Faculty of Information Technology

© 2024 Yevhenii Basov. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Basov Yevhenii. *Ray Marching Scene Creator*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

Acknowledgments	vii
Declaration	viii
Abstract	ix
List of abbreviations	x
Introduction	1
I Theoretical part	3
1 Analyzing the Technology	4
1.1 Introducing to Ray Marching	4
1.1.1 What is 3D rendering?	4
1.1.2 Rendering techniques	4
1.1.3 Ray Marching Features	7
1.2 Terminology and notions	10
1.2.1 Hardware and Software	10
1.2.2 Common visual effects	12
1.3 Game Engines	18
1.3.1 Game engine specifications	18
1.3.2 Engine selection	19
1.3.3 Unity Fundamentals	20
1.3.3.1 Interface	20
1.3.3.2 Programming part	22
2 Existing solutions	23
2.1 Ray Marching Tools	23
2.1.1 Raymarcher — Game toolkit for Unity	23
2.1.2 FERM	24
2.1.3 Raymarching Toolkit for Unity	26
2.1.4 Other projects	26
2.2 Shadertoy	27
2.3 Conclusion	28

II	Practical part	29
3	Project development	30
3.1	Project Features	30
3.2	Methods	31
3.2.1	Interface	31
3.2.2	Primary input processing	32
3.2.3	Shader	32
3.3	Design	33
3.4	Implementation	35
3.5	Testing	65
4	Conclusion	66

List of Figures

1.1	Wolfenstein 3D [4]	6
1.2	Ray Marching 2D distance approximation with SDF in 5 iterations [5]	7
1.3	Soft shadows are very easy to implement using Ray Marching. [6]	8
1.4	With the module function, it is possible to create a repetition of an object without computational expense. [6]	8
1.5	The minimum and maximum functions applied to the SDFs of different objects allow us to simulate the operations of union, subtraction and intersection. [6]	8
1.6	The smooth minimum and smooth maximum functions, create smooth transitions between objects. [6]	9
1.7	A variety of functions can be applied to the SDF result or to the position of the ray in space at each iteration to obtain the desired results, such as displacement, twist or bend. [6]	9
1.8	One of Inigo's Ray Marching works [7]	10
1.9	Variables declaration in each type of memory and their scope. [8]	11
1.10	Access penalty of each type of memory. [8]	12
1.11	Combinations of roughness and metalness. From bottom to top the metallic value is increasing, roughness is increasing left to right. [10]	14
1.12	Quadratic attenuation [11]	15
1.13	Point light example [11]	15
1.14	Hard (a) and smooth (b) transitions of spot light [11]	16
1.15	Spot light cones [11]	17
1.16	The principle of soft shadows [13]	17
1.17	Ambient Occlusion example [14]	18
1.18	Unity 3D 2022.3.11f1 interface.	21
1.19	Unity's template of C# script, screenshot of Visual Studio 2022.	22
2.1	Screenshot from Fractal Sailor game, which was created with Raymarcher tool. [17]	24
2.2	Fractal rendered with FERM tool. [17]	25
2.3	Screenshot of created in Character Creator character [20]	26
2.4	Example of one of Shadertoy user's shader. The code on right side is available to edit. [23]	28

3.1	Example of custom interface created with Unity’s Inspector. . .	32
3.2	Project structure. Created with [24].	33
3.3	First version of shape’s interface. Slider was created with ”Range” attribute.	36
3.4	Properties of Group.	37
3.5	Unity object’s hierarchy and its representation in the form of a tree and its array. Created with [24].	38
3.6	Two renders of an ellipsoid and a plane with 300 and 100 iterations. Equation of exact distance to an ellipsoid is too demanding, so we use its approximation. It causes artifacts when the number of iterations is small. [27]	42
3.7	Illustrations of three SDF operations. These are screenshots from [31]	45
3.8	Example of smooth minimum function. Red and blue are initial functions, purple is the minimum of them, orange is the smooth minimum. Created in Desmos [33].	46
3.9	Different smooth minimum functions comparison in 2D space. Green is the circular smooth minimum [32].	47
3.10	”Bad” tree for naiv tree traversal. Created with [24].	48
3.11	A pawn, created with hierarchical shapes combinations.	55
3.12	Example of using triplanar mapping [35].	56
3.13	Different powers of norm comparison. 1, 4, 10 and infinite powers are demonstrated.	57
3.14	Example of non-symmetric object repetition in 2D space. Lines shows space with the same distance to the closest object (”distance lines”). [36]	59
3.15	Example of implemented domain repetition feature. Checkerboard and pawns are created with domain repetition.	60
3.16	Artifacts on shadow [37].	61
3.17	Implemented soft shadows.	63
3.18	Implemented ambient occlusion.	64

List of Tables

List of code listings

3.1	Example of "Dropdown" and "ConditionalField" attributes. The CubeRounding field appears in Inspector only if Shape variable's value is equal to "CUBE".	36
3.2	Custom structure to store shapes data. otherParameters is another structure which simulates behaviour of float array of 12 elements. We can't use an array directly because of Unity. This "array" is for storing a specific to each shape data.	39
3.3	First frustum corners corners are calculated. Then they are converted to global coordinates with CurrentCamera.transform.TransformVector(), normalized, and set to 4×4 matrix as rows. At the end matrix with frustum corners is sent to the shader.	40
3.4	Compute buffer creating and setting.	41
3.5	getNormal function. getDist function returns distance from the point to an object.	43
3.6	Circular smooth minimum function code [32].	46
3.7	Shader's function, which performs common tree traversal. Tree is constructed in an optimal way already. objProcQueue is array allocated in GPU's shared memory. calcObjectDist calculates distance from point to object, using its special SDF (depends on object). applyFunc applies combining function (min, max, smooth_min, smooth_max) to distances.	51
3.8	Modified tree traversal for materials' coefficients calculation. . .	54
3.9	Function, which returns texture color at specific point.	58
3.10	Soft shadows function. Taken and modified from [37]	62
3.11	Ambient occlusion function. Taken and modified from [38] . . .	64

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Czech Technical University in Prague has the right to conclude a licence agreement on the utilization of this thesis as a school work pursuant of Section 60 (1) of the Act.

In Prague on May 16, 2024

Abstract

This work aims to create a tool for easy creation of 3D scenes, which are rendered with a Ray Marching shader. The tool will be an addon for Unity 3D game engine with a user-friendly GUI for creating parametric 3D objects. The work describes the principles and features of Ray Marching technology, the advantages of its use, explores the complexities and possibilities of rendering miscellaneous objects.

Keywords Ray Marching, sphere tracing, signed distance function (SDF), Unity engine, HLSL, compute shader, rendering

Abstrakt

Cílem této práce je vytvořit nástroj pro snadné vytváření 3D scén, které jsou vykreslovány pomocí shaderu Ray Marching. Tento nástroj bude doplňkem pro herní engine Unity 3D s uživatelsky přívětivým grafickým rozhraním pro vytváření parametrických 3D objektů. Práce popisuje principy a vlastnosti technologie Ray Marching, výhody jejího použití, zkoumá složitosti a možnosti vykreslování různých objektů.

Klíčová slova Ray Marching, sledování koule, funkce vzdálenosti se znaménkem, Unity engine, HLSL, compute shader, vykreslování

List of abbreviations

3D	Three-Dimensional
2D	Two-Dimensional
SDF	Signed Distance Function
CPU	Central Processing Unit
GPU	Graphic Processing Unit

Introduction

Rendering technologies play a pivotal role in the modern digital landscape, shaping the way we interact with virtual environments, immersive experiences, and digital content across various industries. From video games and architectural visualization to film production and scientific simulations, rendering technologies are at the forefront of delivering realistic and captivating visuals.

In the field of computer graphics, the desire for photorealism and the creation of immersive virtual environments has driven the evolution of rendering techniques. Probably everyone has heard about Ray Tracing rendering model — a method that simulates the behavior of light rays in space to achieve photorealistic images. I would like to dedicate this work to a similar but less popular Ray Marching model. This technology is also based on guessing the distance from the camera to the object in space, but unlike Ray Tracing it does it on the principle of "binary search" — at each iteration the length of the path traveled by the ray is less by about two times from the previous iteration. This feature allows to draw complex 3D shapes of objects without approximation by polygons using mathematical formulas, combine them in various ways and even change them in dynamics without using large computing power, which is problematic to do in other rendering models. However, this also means that conventional 3D models are not quite suitable and require a special approach when rendering with Ray Marching.

All of the above will be demonstrated in this thesis using the Unity engine, on which will be written a program to easily configure the Ray Marching shader to render the desired 3D scenes without having to make manual changes to the shader code.

Objectives

This thesis is aimed primarily at creating the main application.

The first part of the thesis will describe the main Ray Marching technology — its advantages, disadvantages and other differences from other rendering techniques.

Then we will investigate the capabilities of this technology. The peculiarities of their implementation in the selected Unity 3D game engine will be described. The general principles of rendering realistic 3D graphics will also be touched upon.

In the end a tool will be implemented, which will include two parts — a graphical interface for creating parametric objects/scenes and a shader, which will render the final image. The final version of the program will include as many features described in the second part as possible.

Part I

Theoretical part

Analyzing the Technology

This chapter covers the basics of rendering, the main rendering techniques of the moment, and the features and differences of the Ray Marching technique.

1.1 Introducing to Ray Marching

1.1.1 What is 3D rendering?

3D rendering in computer graphics is the process of creating 2D images or image sequences (animation) based on 3D models. [1] We can say 3D rendering is a way of displaying 3D graphics. Today 3D graphics is used in many industries, such as computer games, animation, advertising and others. At the same time, each industry has its own requirements for the final picture. For example, in advertising, marketers often choose a photorealistic style to demonstrate their products, and the time of rendering the image or animation does not play a major role. Due to this, hired artists can spend hours to render a single image to create a picture indistinguishable from reality. Whereas in video games, one of the main criteria is a high frame rate, which with modern technology is difficult to comply with rendering photorealistic images on each frame. Thus at the moment we have many techniques of 3D rendering, known and less known, to meet any criteria of the user.

1.1.2 Rendering techniques

Rasterization and scanline rendering This technique is based on the projection and rasterization of 3D objects onto a 2D image. 3D models consisting of polygons (usually triangles) are projected onto a 2D plane (the final image), taking into account the depth, in order to create the effect of overlapping the rear objects with the front ones. [2] Rasterization — splitting triangles that make up objects into separate pixels - also takes place. Then the final color of a pixel is calculated. This takes into account light

sources, object materials and other parameters to get the desired image. [3]

This technique allows you to render a satisfactory picture in a short time, because of which it is widely used in computer games and supported by most modern graphics cards. The minus, however, is that with this approach is very difficult to create a picture that will thoroughly copy the reality. In other words, the quality of the image is questionable.

Ray Based Techniques There are several rendering techniques based on beaming rays from the camera in the direction of each pixel in the image. Their key feature is to calculate the distance from a certain point in space to the nearest object along a certain direction. These techniques have a lot in common with each other, but at the same time each of them has its own features. As it is easy to guess from the name, one of them is Ray Marching, which is the subject of this thesis.

Ray Tracing This technique is based on tracing rays from the camera in the direction of each pixel on the screen. The distance to objects is calculated through complex formulas, which is one of the main differences from other similar techniques. The disadvantage is that it is very difficult, if not impossible, to derive a formula for the distance to complex objects.

This method works similarly to how rays of light propagate in the real world, except that the rays are not traced from the light source to the camera, but from the camera to the light source. When calculating the color of a pixel, not only the color of the object that is encountered in the path of the ray is taken into account. All objects reflected by the ray on its way to the light source contribute to the final pixel color. [2]

This method, simulating physical processes, allows to achieve high quality photorealistic images with such optical effects as reflection, refraction, chromatic aberration, soft shadows, depth of field and others, which are difficult to achieve with other techniques, such as rasterization and scanline rendering. On the other hand, it is obvious that such detailed computations for each pixel cannot be cheap in terms of computational resources. Therefore, Ray Tracing is rarely used for real-time rendering.

Ray Casting Ray Casting is commonly referred to as a technique where rays are not produced for each pixel of an image, but for each column of pixels. Also, the ray is spread over a 2D scene rather than a 3D scene. The distance to the object is calculated over many iterations — at each iteration the ray takes a step whose length is equal to a constant value — usually the diameter of the smallest unit of which the scene is composed.

Then the depth is used to calculate the height of the column of pixels — the greater the depth, the smaller the column. This method allows you

to quickly render an unrealistic image, but with a visible depth effect. Because of this Ray Casting was used in the past, when computers were not very powerful, to create games. An example is the game Wolfenstein 3D.



■ **Figure 1.1** Wolfenstein 3D [4]

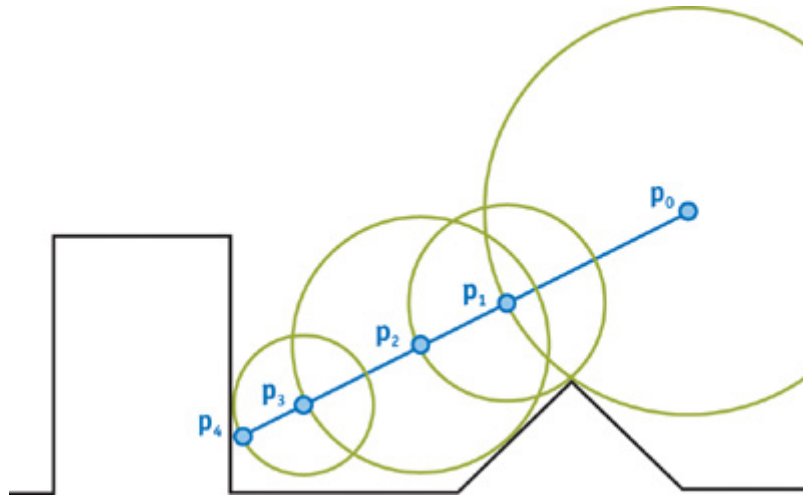
Ray Marching The Ray Marching technique is similar to Ray Casting in that the distance to the object is calculated iteratively. However, unlike Ray Casting, ray propagation is performed for each pixel on the screen in the traditional 3D space.

The step size at each iteration can be calculated in two ways. The first is similar to Ray Casting, where the ray takes a step by a constant amount. The second is that at each iteration the distance from current ray endpoint to all objects in the scene (in all directions) is calculated first. Then the ray takes a step in its direction by the value of the minimum distance among all distances to objects. This method has a special term — sphere tracing. Usually, when people talk about Ray Marching on the Internet, they mean sphere tracing. In this thesis we will also use sphere tracing — it is with the help of this method it is possible to achieve interesting visual effects, which will be discussed further.

1.1.3 Ray Marching Features

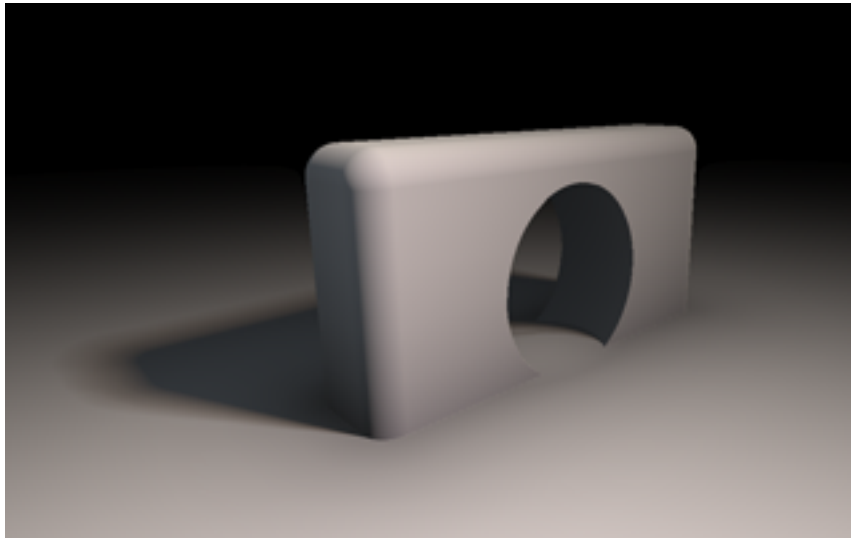
It is time to tell you about the features of Ray Marching technology and in what cases it can be used.

As it was described earlier, ray propagation occurs with calculation of distance to the nearest object to the point. This is done with the help of signed distance function (SDF). This function returns the distance from a given point to a certain object. Each object has its own signed distance function. However, you should not confuse Ray Tracing distance calculation with Ray Marching distance calculation. In the first case, a function is used to calculate the distance from a point in space to an object along the ray direction. In the second case, the distance from the point to the object is calculated — without taking into account the ray direction. This is the fundamental difference between these two approaches.



■ **Figure 1.2** Ray Marching 2D distance approximation with SDF in 5 iterations [5]

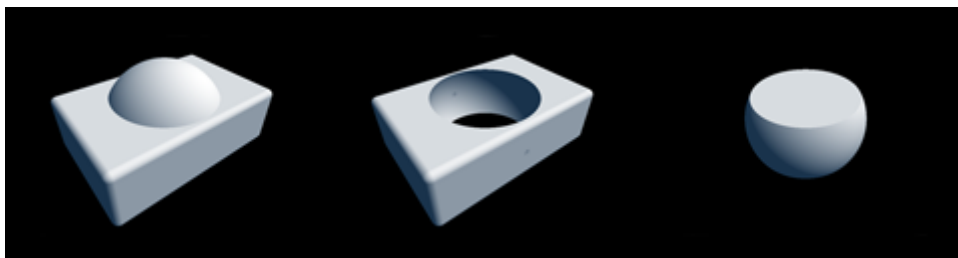
The following will demonstrate visual effects that are difficult or impossible to reproduce in a short time using other techniques. Details of their implementation will be described in the practical part of this thesis.



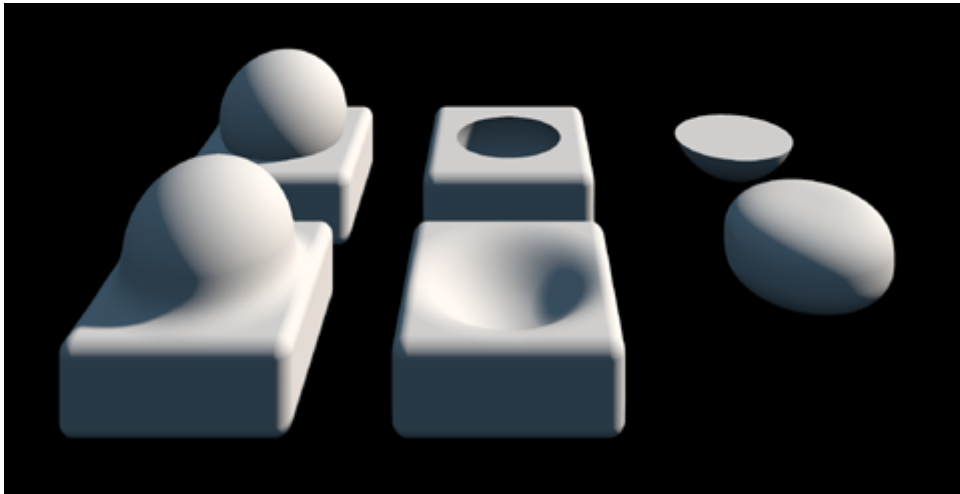
■ **Figure 1.3** Soft shadows are very easy to implement using Ray Marching. [6]



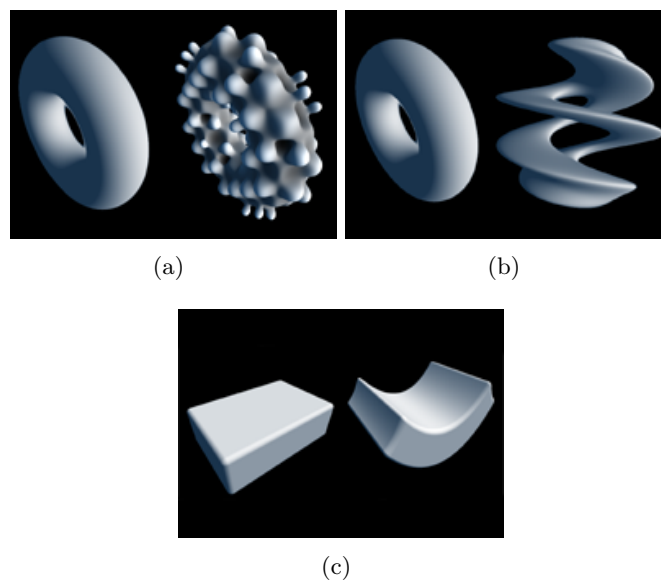
■ **Figure 1.4** With the module function, it is possible to create a repetition of an object without computational expense. [6]



■ **Figure 1.5** The minimum and maximum functions applied to the SDFs of different objects allow us to simulate the operations of union, subtraction and intersection. [6]



■ **Figure 1.6** The smooth minimum and smooth maximum functions, create smooth transitions between objects. [6]



■ **Figure 1.7** A variety of functions can be applied to the SDF result or to the position of the ray in space at each iteration to obtain the desired results, such as displacement, twist or bend. [6]

With these simple functions it is possible to create a wide variety of images. A good example would be the work of Inigo Quilez, who at one time greatly popularized the Ray Marching technique.



■ **Figure 1.8** One of Inigo's Ray Marching works [7]

1.2 Terminology and notions

1.2.1 Hardware and Software

Shaders Nowadays, all 3D graphics are rendered using shaders. Shaders are programs that use the power of the graphics card (GPU) rather than the computer's central processing unit (CPU) for their calculations. What is the difference between graphics cards and CPUs? CPUs are designed to quickly execute a chain of simple machine instructions (which make up any algorithm). Modern processors also have multiple cores, which allows them to execute multiple chains of operations in parallel. This is usually sufficient for most tasks. However, problems start when we encounter 3D graphics. When rendering 3D graphics, in order to calculate the final image we need to perform calculations for each pixel. And although individually the complexity of these calculations is not so high, when totaled for all pixels the amount of resources required becomes impressive. To render a single image in FULL HD format, which has already become a standard among monitors, requires $1080 \times 1920 = 2,073,600$ pixel calculations. This is too much even for modern processors. Despite this large number, graphics computing has a special feature — the computation of each pixel is independent of the computation of other pixels. This means that these computations can be performed simultaneously. As described above, today's processors have several cores each, which could be used to realize parallel computations. However, this is still not enough for rendering graphics. This is where graphics processors come to the rescue.

GPUs, aka graphics cards, are designed to perform thousands of calcula-

tions simultaneously. In simplified form GPU can be represented as thousands of relatively weak CPUs combined in a place. The power of one core of a graphics card is large enough to quickly perform calculations for a single pixel, and at the same time these cores are many enough to quickly perform calculations for millions of pixels. Due to the peculiarities of its architecture, not every task can be efficiently solved on a graphics processor. Only those tasks that can be broken down into a large number of small tasks whose computations will not depend on each other are suitable. Nevertheless, graphics cards are used not only for rendering graphics, but also in other areas. For example, in the modern field of artificial intelligence, graphics processors are used to multiply large matrices with thousands of rows and columns.

For such special devices as video cards it is necessary to write special programs. Whereas the usual code is a large algorithm of a long chain of actions, the code for graphics processors is a relatively small algorithm that takes some information from the CPU (e.g. data about the scene in the form of models and their materials), and based on it, with specificity for each pixel, performs calculations and outputs the result (e.g. the color of the pixel). Such programs are called shaders. In this paper, one of the main parts will be writing a Ray Marching shader.

GPU A detail still worth mentioning about the GPU is its memory structure. Its memory is categorized into registers, local, shared, global and constant. Each of them has its own limitations. You can see their differences in the following tables:

Variable declaration	Memory	Scope	Lifetime
<code>int localVar;</code>	register	thread	thread
<code>int localArray[10];</code>	local	thread	thread
<code>__shared__ int sharedVar;</code>	shared	block	block
<code>__device__ int globalVar;</code>	global	grid	application
<code>__constant__ int constantVar;</code>	constant	grid	application

■ **Figure 1.9** Variables declaration in each type of memory and their scope. [8]

Variable declaration	Memory	Performance penalty
<code>int localVar;</code>	register	1x
<code>int localArray[10];</code>	local	100x
<code>__shared__ int sharedVar;</code>	shared	1x
<code>__device__ int globalVar;</code>	global	100x
<code>__constant__ int constantVar;</code>	constant	1x

■ **Figure 1.10** Access penalty of each type of memory. [8]

Here it is worth explaining what thread, block and grid are. A thread is the smallest branch of calculations. It can be represented as one branch of calculations performed for one particular pixel on the screen. A group of threads is combined into a block, and a group of blocks is combined into a grid.

As you can see from the tables, the memory that is available only to each thread is only registers and local. All variables that we will declare when writing the shader will be registers, because each thread (for each pixel) will have its own values of these variables. However, pay attention to local memory. Local memory is all local arrays. As you can see from the table, this memory is terribly slow. However, when writing programs, we often have to use arrays to store temporary data. Fortunately, there is a way to avoid performance drops when using arrays in shaders, namely to use arrays declared in shared memory [9]. We will talk more about this in the implementation part.

1.2.2 Common visual effects

Since the development of 3D graphics, people have tried to reproduce the picture of the real world on the monitor screen as accurately as possible. And for better performance, it often had to be done in a roundabout way — “faking” various visual effects caused in the real world by the complex behavior of light through simple approximations. Many of these visual effects have gotten their own names and their own methods of calculation in different 3D rendering techniques. In the following, we will talk about some of them, which will later be implemented in the final application.

Materials Materials play an important role in rendering an object. They set the object’s color, gloss, reflectivity, opacity, luminosity, transparency, and other parameters. In the real world, how a material looks depends on the chemical compounds it is made of and their structures, inside and outside the object. Light rays can be reflected at different angles, can penetrate the material to different degrees, and can pass through it. All of this affects how the material will ultimately look. Physically based rendering techniques,

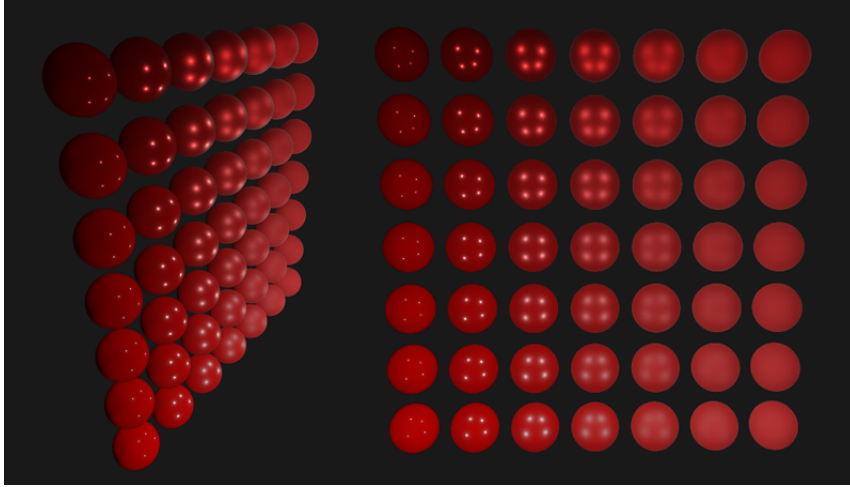
such as Ray Tracing, attempt to simulate in detail the behavior of rays and their interaction with materials, for the sake of a highly realistic picture. However, this method is obviously very resource intensive. Therefore, other ways have been devised to achieve similar results when rendering materials without accurately simulating the behavior of light rays.

To date, the materials of objects allocate a certain number of properties. If they are properly set up and rendered, you can accurately represent almost any material from the real world. To the properties of the material include — albedo, roughness, metalness, opacity, emission, IOR. In this thesis we will pay attention to some of them.

Albedo Albedo is responsible for the percentage of light that is reflected from an object. In the real world, light is partially absorbed and partially reflected when it collides with an object. And light of different spectra is most often absorbed/reflected differently. Thus we see different colors. In various programs, the Albedo parameter is usually responsible for the color of the object.

Roughness Roughness is a parameter that determines how strongly an object reflects light at an angle equal to the angle of incidence of light. In reality, the surfaces of objects have a large number of grooves and irregularities that cause the incident light to be reflected not completely at the right angle. An object with maximum roughness will reflect light in all directions in the same amount. On the contrary, an object with minimal roughness will have a near mirror-like surface because most rays will be reflected at an angle equal to their angle of incidence. When implementing material mapping, roughness is usually divided into two parameters — diffuse and specular. Diffuse defines the amount of diffused light, specular — the amount of light reflected at the angle of incidence.

Metalness Metalness, as the name implies, defines how strongly a material reflects light, as metals do in the real world. People have long noticed that metals reflect light differently than non-metals. This is partly why metalness is now often defined as a separate parameter of materials. Like roughness, metallicity is mostly responsible for the amount of light mirrored. A material with high metallicity and low roughness will literally have the properties of a mirror.



■ **Figure 1.11** Combinations of roughness and metalness. From bottom to top the metallic value is increasing, roughness is increasing left to right. [10]

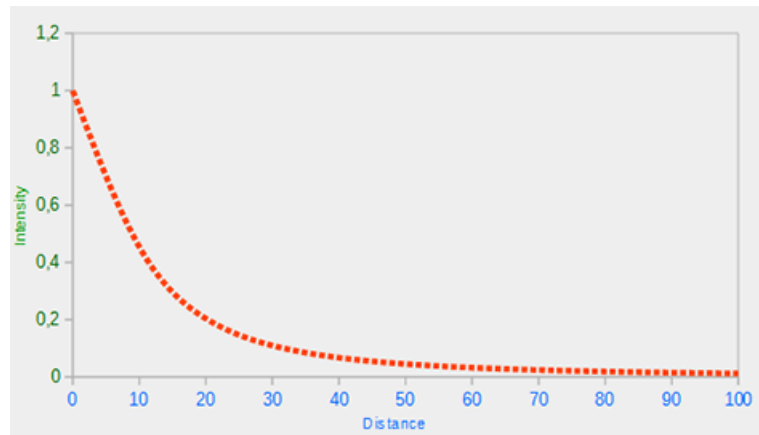
Light casters Light sources are usually categorized into three types — “directional light”, “point light” and “spot light”. In many programs with tools for creating 3D graphics is found still “area light”, but in this thesis we will not touch it. They differ only in the area of the illuminated region and the angle of incidence of light on the object.

Directional Directional light can be imagined as an infinitely distant light source. The level of illumination of objects illuminated by directional light does not depend on their location in space, and the angle of incidence of light on them is always the same. It is usually used to simulate illumination from the sun. And although the sun is not infinitely distant from the Earth, it is far enough away that on a small area of land we perceive the brightness and angle of incidence of its rays as the same at any point in space. The main parameters of such a light source are its directivity, brightness and color. These parameters are sufficient to calculate the measure of illumination of an object. [11]

Point Point light is light that emanates from a point in space and propagates in all directions. It usually has a limited range of coverage. A typical analog of point light in the real world is a regular light bulb. The parameters of a point light are its location in space, range, brightness, and color. Also additional parameters can be three terms — constant, linear and quadratic terms. These terms are used to calculate the attenuation of light as the object moves away from its source. The attenuation is calculated using the following formula:

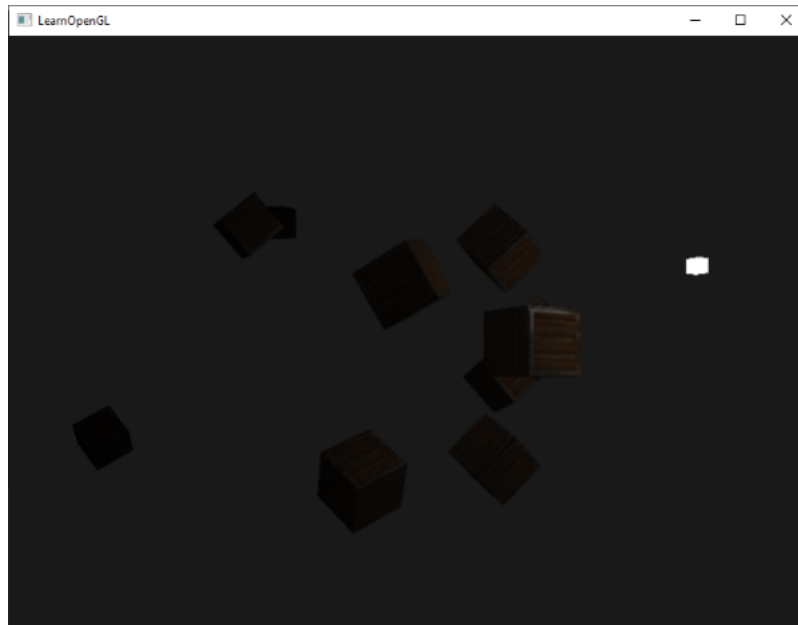
$$F_{att} = \frac{1.0}{K_c + K_l \cdot d + K_q \cdot d^2},$$

where d is the distance to the light source, and K_c , K_l and K_q are constant, linear and quadratic terms, respectively. [11] Depending on the values of these parameters, the illuminance level from distance will vary approximately as follows:



■ **Figure 1.12** Quadratic attenuation [11]

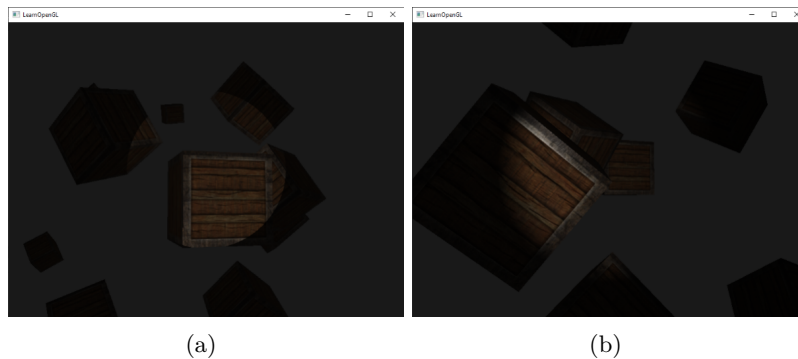
As can be seen, the light intensity decreases with distance from the source at a quadratic rate. This calculation gives a more plausible result than a sharp drop in brightness from 1 to 0 or a linear dependence of brightness with distance.



■ **Figure 1.13** Point light example [11]

However, not all programs allow you to adjust these coefficients. For example, in Unity 3D they are constant $K_c = 1$, $K_l = 0$ and $K_q = 25$. [12]

Spot Spot light is essentially a spot light, but with a limited illumination angle. The analog in the real world is a flashlight. It is not hard to guess that in general the parameters of such a light source will be the same as those of a spot light. The difference will be that the spot light also requires the direction and angle of the cone of light. The angle of the cone of light is also divided into outer and inner. This is to create a smooth, realistic transition between lit and unlit areas.

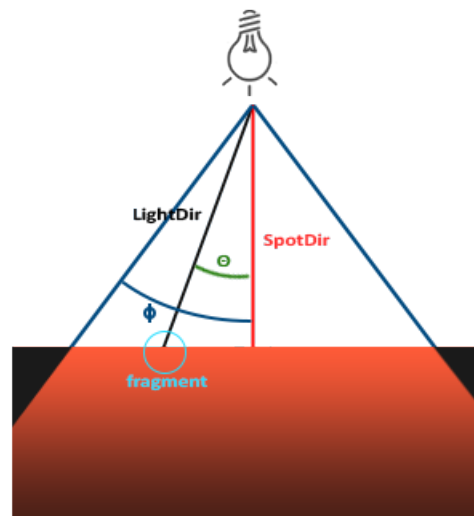


■ **Figure 1.14** Hard (a) and smooth (b) transitions of spot light [11]

The coefficient of this transition is calculated using the following formula:

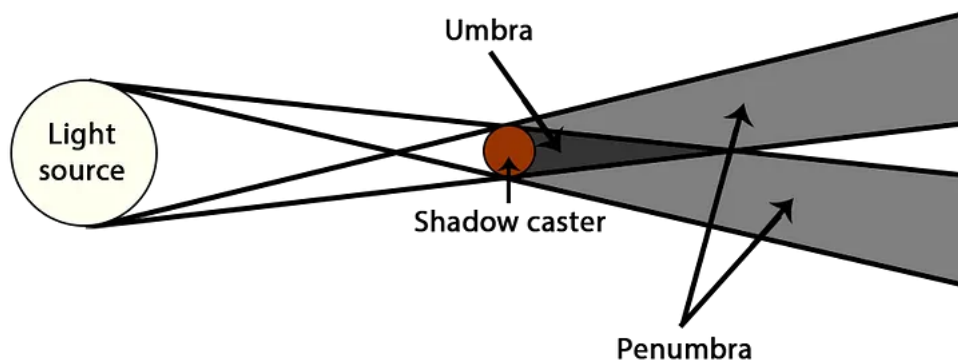
$$I = \frac{\theta - \gamma}{\epsilon}.$$

Here θ is the cosine of the angle between the vector of the illuminated point and the direction vector of the light cone, γ is the cosine of the angle of the outer cone and ϵ — the difference of the cosines of the angles of the inner and outer cones. [11]



■ **Figure 1.15** Spot light cones [11]

Soft shadows Soft shadows are an effect that is difficult to achieve in a rasterization shader. Soft shadows are characterized by a soft transition from the shadow to the unshaded part, through the penumbra. In the real world, soft shadows mainly occur when the light source is large enough, relative to the distance to the object, and the plane on which the shadow falls is far enough from the object. Given that most of the light in our world is reflected light, almost all shadows are soft.



■ **Figure 1.16** The principle of soft shadows [13]

Implementation of soft shadows using Ray Marching has its own peculiarities, which will be described below.

Ambient Occlusion Ambient Occlusion is essentially a technique for shading narrow spaces. Most objects in the world are illuminated not through direct light rays, but through reflected ones. When rendering graphics, indirect lighting is often simulated by adding a constant measure of ambient light to each object in the scene, whether they are directly lit or not. This does not take into account that areas of the scene obstructed by other objects should be darker. Ambient Occlusion partially solves this problem. This technique shades those areas that are potentially difficult for light rays to reach. For example, in one of the first versions of Ambient Occlusion called SSAO (Screen space ambient occlusion), the coefficient is calculated in a similar way: for a point on the screen, its position in space is taken, and then several spatial points around the original point are taken. Those points that are inside the geometry of the objects will contribute to the shading coefficient.[14]



■ **Figure 1.17** Ambient Occlusion example [14]

We will talk about the peculiarities of implementing Ambient Occlusion in Ray Marching later.

1.3 Game Engines

1.3.1 Game engine specifications

A game engine is a framework designed to create games. They usually include various libraries and a level editor. There are many game engines and each one has its own features. When choosing an engine you should pay attention to such criteria as:

Scale of the project Different engines are adapted to create projects of different scales. Some have complex and extensive tools and are more suitable

for AAA game development. Others are easier to learn and use, and are convenient for creating small 2D projects developed by small groups of people. Choosing the wrong engine according to this criterion can lead to difficulties and delays in development in the future.

Target platform The target platform, such as PC, consoles or mobile devices, of the future game also affects the choice of engine. Different engines support creation of projects for different platforms. Some are cross-platform and support a large number of platforms, others do not, and support only one. Therefore, prematurely selecting the target platform or platforms is also important when choosing a game engine.

Project Budget Budget constraints also affect the choice of an engine. Some are open source and free, while others may be paid or have various more complex monetization systems. Unity 3D, for example, is free for users whose annual game revenue does not exceed \$100,000.

Skillset Each game engine has its own set of technologies used, in particular the programming language. Skillset of the developer or developers, and the prospect and complexity of the engine technologies for further learning is also important when choosing an engine.

Graphics Different engines provide different capabilities when rendering graphics. With some of them, it is easier to achieve the desired result than with others. For example, Unreal Engine is famous for its photorealistic rendering.

Community A large and active community can be very helpful when learning a new engine, especially for single developers. The official documentation does not always cover all the many questions that a developer who has not experienced the engine and its technologies before may have. In such moments, answers and tutorials from the community are very important, which can significantly speed up development.

Long-term support When developing large, long-running projects, it is important to also take into account the development of the game engine. Regular updates and changes to the engine can affect the development of the game in the long run.

[15]

1.3.2 Engine selection

So, having familiarized ourselves with the criteria for selecting game engines, we can choose which criteria are important to us, and based on them make a choice of a suitable game engine.

- In the case of our project, it is important for the target engine to have good support for 3D rendering and writing custom shaders.
- The scale of our program is not very large, so it should be convenient to develop small projects on the engine we choose.
- At the same time, since our project is not quite a standard project, it is very important to have a large community, with the help of which we can quickly find solutions and answers to various questions.
- Our project is not commercial, so it is desirable that the target engine was free or conditionally free.
- And also important is the presence of personal experience in working with the future selected game engine.

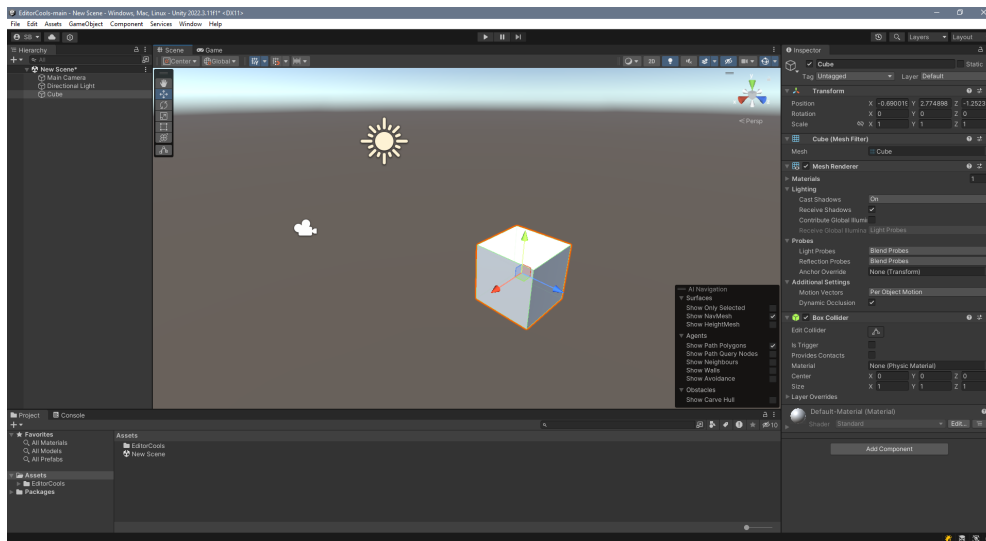
Taking into account all the above criteria one of the best choices will be Unity 3D engine. In the case of our project it is free, it is considered one of the easiest to learn, has good support for 3D rendering and writing your own shaders, it is relatively easy to implement small projects, has one of the largest community and the author of this thesis has experience working with it. All this makes Unity a good candidate for developing a Ray Marching editor.

1.3.3 Unity Fundamentals

In this section we will talk about the structure and basic concepts of the Unity 3D engine.

1.3.3.1 Interface

Below you can see a screenshot of the Unity workspace. Based on it we will describe all the basic concepts of the engine.



■ **Figure 1.18** Unity 3D 2022.3.11f1 interface.

In the center is the scene window. On it, the developer creates game levels by placing models, lighting, interface icons and other objects in the space.

On the left side is the hierarchy window, which currently contains several objects. The objects in the hierarchy window are the objects that are represented on the scene. These objects can be placed one under another, creating a hierarchical structure. Each child element in the hierarchy inherits the rotation, scale and position in space from its parent, adding them to its own. This is very convenient because it allows you to group objects and in many cases treat them as a single object.

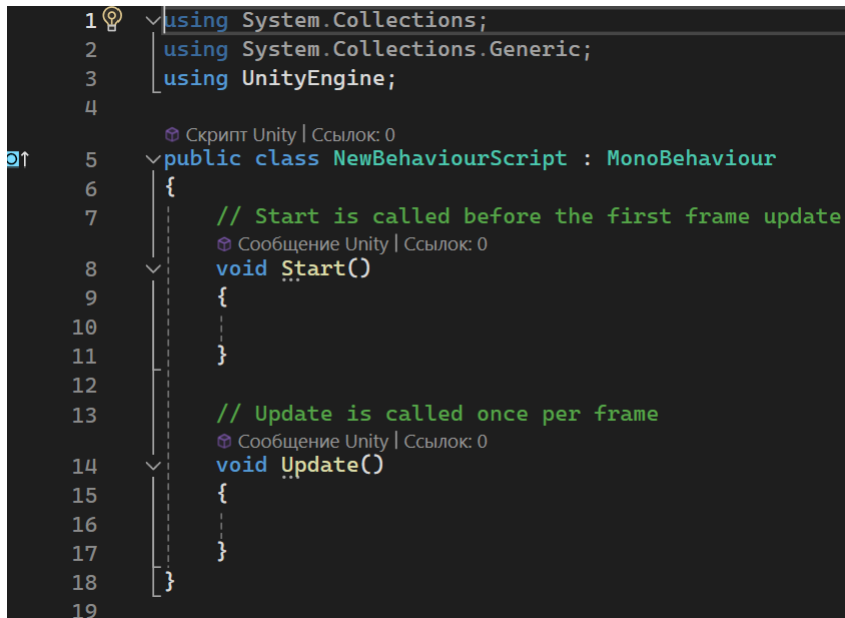
On the right side is the inspector window, which can be used to view and modify object properties. In this example, the cube has such components as Transform, for defining position, rotation and scale in space, Mesh Renderer, for displaying the cube on the scene, Box Collider, for physical interaction and cube material. Unity has a large number of ready-made components. By writing your own scripts you can create your own components for objects, giving them the necessary properties and specifying their behavior.

At the bottom is the project window, which displays all the files involved in the project. Here you can add images, models, scripts, scenes and any other files needed for the project. Also to the right of the project window label you can see the console window label. The console is used for debugging the program — displaying errors and messages for the developer.

Finally, at the top center, you can see the “Play” and “Pause” buttons, for playing and testing the game.

1.3.3.2 Programming part

Unity uses the C# programming language. After creating and opening a script through Unity, you will see the following template.



```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class NewBehaviourScript : MonoBehaviour
6 {
7     // Start is called before the first frame update
8     void Start()
9     {
10
11     }
12
13     // Update is called once per frame
14     void Update()
15     {
16
17     }
18 }
19
```

■ **Figure 1.19** Unity’s template of C# script, screenshot of Visual Studio 2022.

As you can see, by default, Unity adds in its library and creates a class that is named as a script and inherits from the `MonoBehaviour` class. `MonoBehaviour` is the base class of the Unity `GameObject`, which has a large number of useful methods. Two of these methods are `Start` and `Update`.

The `Start` method is called when a scene is started with the object that owns the script. If there is only one scene, this happens when the game starts. The `Start` method is called once each time a scene is opened. This is usually where the developer sets the initial configuration of the object, allocates containers, and other things to prepare the object.

The `Update` method is called at each frame of the game. This is where the main behavior of the object is defined — what the object “does”.

Of course, in addition to these and other pre-defined methods, an object can have other custom methods and attributes, both private and public.

Other things Here’s what else you should know about Unity objects:

- For a script to work (call the `Update` method), it must be assigned as a component to at least one object in the scene.
- Scripts can access public attributes and methods of other scripts.
- Public attributes can be configured manually in the Inspector window (which will be described later in this thesis).

Existing solutions

Ray Marching is not as popular as other rendering techniques, but there are relatively many tools available for its use. In this chapter we will look at some of them, and at the end we will summarize and decide which features may be unique for our project.

2.1 Ray Marching Tools

2.1.1 Raymarcher — Game toolkit for Unity

Raymarcher is a powerful tool from Matej Vanco that allows you to create and customize scenes with Ray Marching in a very flexible way. Here’s a list of key features the creators write about:

- “One-click setup”
- “Simple and user-friendly interface”
- “Collection of SDF primitives, fractals, and volumes.”
- “Collection of well-known SDF modifiers and boolean operations”.
- “Built-in standard material library”
- “Built-in rendering filters”
- “Complete toolkit for simple voxel manipulation”
- “Mesh-to-3D volume converter tools”
- “Cross-platform support, including VR, mobile & WebGL”
- “Compatibility with all Unity render pipelines (Built-in, URP, HDRP)”
- “Complete video documentation series”

- “Real-time support and access to unlimited updates”
- “Original example content”

[16]

This tool implements most of the features of Ray Marching technique. Of all existing similar projects, this one is one of the most advanced to date. Also, as the developer writes, the project is compatible with the three newest versions of Unity 2023.2.12f1, 2022.3.20f1, 2021.3.35f1 and with all Unity pipelines. However, the project is not open source, and one of the drawbacks, if it can be considered a drawback, is its high price. You can familiarize yourself with this tool here: [16].



■ **Figure 2.1** Screenshot from Fractal Sailor game, which was created with Raymarcher tool. [17]

2.1.2 FERM

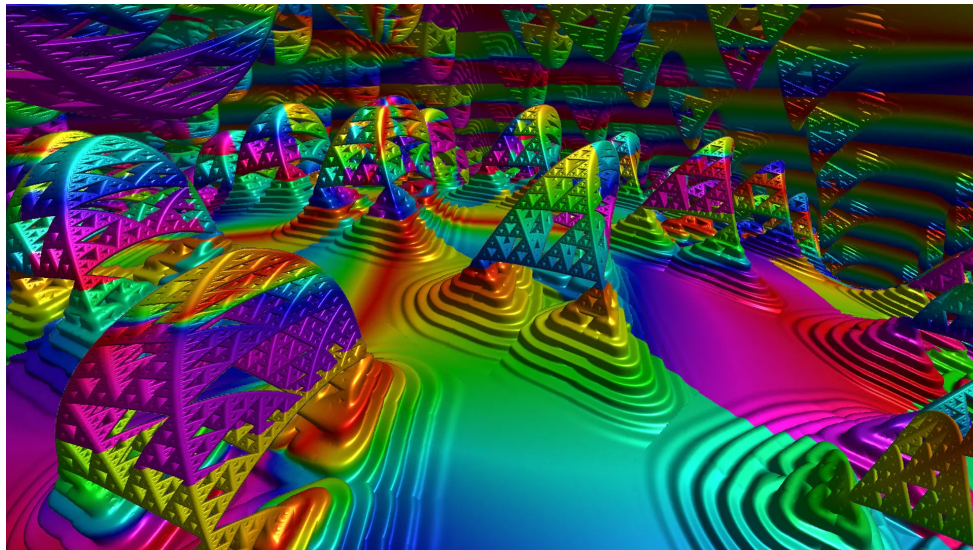
Fast Easy RayMarching is another paid large Ray Marching tool for Unity, created by Ward Dehairs. This project also provides a large number of features, about which the developer writes:

- “Real time rendering for all features, down to the biggest, baddest, infinitely repeating, shape shifting monster you could ever dream up.”
- “Skybox mode: fill the sky with mesmerizing fractals. Ideal for creating a dreamy or alien atmosphere.”
- “Animation. make complex shapes fold, rotate, move and otherwise animate, without breaking your head over complex math!”

- “Shape mixing, smooth unions that blend shapes like liquids, distorted mixes to create an otherworldly impression and more.”
- “Infinite rendering distance. Repeating fractals that run on to the horizon or fill the entire sky. Infinite planes, tunnels, cones and spirals!”
- “Unity lighting support, because who wants to reinvent the wheel? We support both the standard shader option and unlit behavior!”
- “Loads of primitive shapes in both 2D & 3D, such as spheres, boxes, planes, pyramids, arches, infinite tunnels, circles, spirals, polygons... you name it, we got it!”
- “Fractals again, both in 2D & 3D: Mandelbulb, Mandelbrot, Sierpinski, Julia and Koch snowflake, just to name a few. Not to mention, the option to create your own fractals with the powerful recursion component.”
- “Texture support. No more tedious UV unwrapping! Get it packaged automatically in a variety of shapes.”
- “Highly customizable with tons of options to optimize settings and performance.”

[18]

A distinctive feature of the project is the strong emphasis on fractal generation.

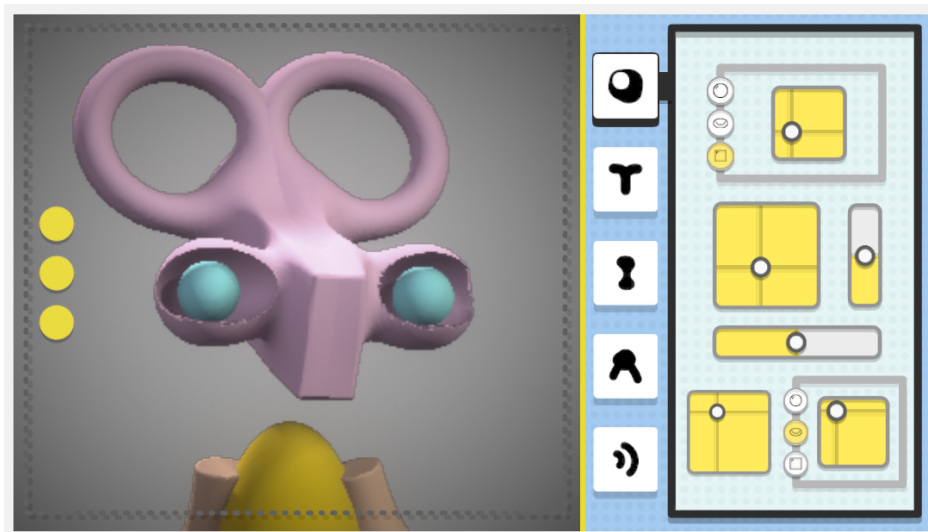


■ **Figure 2.2** Fractal rendered with FERM tool. [17]

2.1.3 Raymarching Toolkit for Unity

Raymarching Toolkit is another good Ray Marching tool for Unity. This addon was created by Kevin Watters and Fernando Ramallo in 2018, but for unknown reasons the link to download it was removed from itch.io, the only site it was distributed on, and it is currently unavailable. However, you can still read about the features of the tool on the developers' website. [19]

Judging by the description and photo and video materials from the developers' site, the addon had all the main features of Ray Marching technology. In the project were implemented such features as smooth Boolean operations (smooth union and smooth subtraction), various modifiers, including displacement, bend, twist and others, domain repetition, mirroring, rendering fractals, soft shadows and ambient occlusion and others. It probably used to be one of the most powerful Ray Marching tools as well, but it is unfortunately not available now. As of today, what remains of the project is a sample application made with the Raymarching Toolkit called Character Creator, which is available on the developers' website. [20]



■ **Figure 2.3** Screenshot of created in Character Creator character [20]

2.1.4 Other projects

There are several other projects on the Internet that perform the function of Ray Marching scene editor. Although they are not as big as the previous two, we will pay some attention to them.

Raymarching-Engine-Unity This project, judging by the v002 version listed on the project's github, is at an early stage of development. However, it

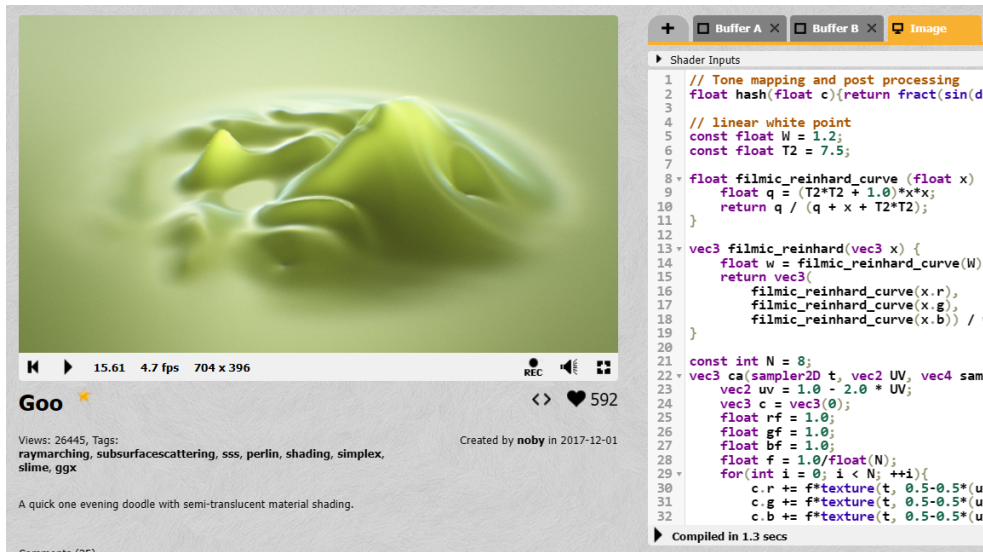
already supports over 28 primitives (including fractals, n-dimensional objects, volumetric clouds) and set operations (Union, Subtract, Intersect). A distinctive feature of this project is the detailed customization of lighting and shadows, as well as the creation of custom shapes. [21]

Unity Ray Marching This tool supports simple creation of several types of shapes and Boolean operations between them. However, this project has one interesting feature — the use of AABB trees in rendering. [22] In short, AABB trees are a way of organizing 3D objects in space, which allows you to calculate distances at each iteration only for objects that are most likely to collide with the ray. This technique allows to optimize calculations significantly. Probably this or other similar techniques were also used in previous major projects, but in any case it was not mentioned.

2.2 Shadertoy

I would like to mention separately the Shadertoy site. Although this site is not a tool for Unity and Ray Marching technology is not its central priority, it has historically been very important to the evolution of Ray Marching technology. This resource was created by Pol Jeremias and Inigo Quilez. Inigo Quilez has been mentioned before as one of the main popularizers of Ray Marching. On his YouTube channel you can find many tutorials on creating scenes with Ray Marching, and on his blog — many code examples and explanations of many Ray Marching and not only Ray Marching techniques. [6]

The Shadertoy site allows its users to write and render any shaders they want, as well as share them and freely view other people's shaders and shader code. Many shaders that can be found there apply Ray Marching technology.



■ **Figure 2.4** Example of one of Shadertoy user’s shader. The code on right side is available to edit. [23]

2.3 Conclusion

In this chapter, we looked at several tools for creating scenes using Ray Marching. In general, we can say that these tools are able to satisfy most of the users’ wishes. However, the main tools are not freely available. Also, there is one very obvious feature that was not found in any of the projects described above. It is about the hierarchy of applying Boolean operations to objects. It is quite obvious that a user may want to apply several subtraction/union/intersection operations one after another in a certain order. However, for some reason this has not been implemented in any of the projects. For this reason it was decided to add this feature to the project and make it the main feature of this work.

Part II

Practical part

Project development

3.1 Project Features

The goal of the project is to provide the user with as many tools as possible to create a graphical scene. Among them we can highlight:

1. Compatibility with Unity's built-in rasterization shader: one of the main conditions is that objects rendered with Ray Marching are compatible with Unity's regular polygonal objects. Objects rendered using both methods must overlap correctly, based on the depth buffer.
2. Primitive templates: creating complex scenes/objects with Ray Marching consists of combining various simple shapes such as sphere, cube, torus, etc. It is therefore important to provide the user with a wide choice of primitives.
3. Simple transformations: transformations such as object rotation and scaling when working with 3D objects have already become common for any game engine or 3D editor. Therefore, it is also important that they work for objects rendered with Ray Marching.
4. Boolean functions for combining objects: In the community of people who create shaders with Ray Marching techniques, Boolean functions for combining objects such as minimum, maximum, smooth minimum and smooth maximum are an integral part of creating a scene. Accordingly, these functions should also be added. It should be added that it is also important to implement combining not only single objects, but also groups of objects. For example, in theory a user may want to create an intersection of two primitives and then subtract this intersection from the third primitive. As mentioned in the conclusion of the previous chapter, the transformation hierarchy will be the main distinguishing feature of this project.

5. **Materials:** Materials are an important part of 3D graphics in general. Therefore, it is important to create an interface for creating custom materials and the best possible model of their processing and rendering, so that the final picture would look as good as possible.
6. **Light and shadow:** as well as material, light strongly affects the quality of the final image. Therefore, quality rendering of light and shadows is very important. Here we can emphasize the processing of directional, point and spot light sources, as well as rendering soft shadows and ambient occlusion.
7. **Object repetition and mirroring:** in the Ray marching technique object repetition as well as mirroring is one of the cheapest effects in terms of computational resources. Due to the cost of using a large number of different objects in a scene, object repetition is one of the most important tools.
8. **Other object deformations:** Ray Marching has a large scope for deforming the geometry of objects. Deformations such as displacement, twist and blend, which were demonstrated in Figure 3.7, should also be added.
9. **User Interface:** Finally, the user GUI is also an important part of the project. The GUI is intended to facilitate the user's interaction with the tool, and therefore should be intuitive and not cause difficulties or misunderstandings when interacting with it.

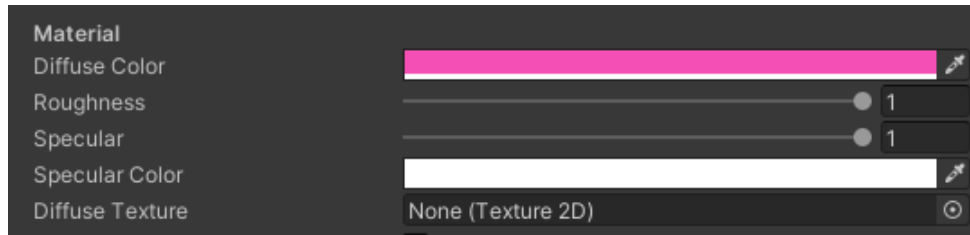
3.2 Methods

In general, the program can be divided into three parts: the interface, the primary input processing and the shader.

3.2.1 Interface

The interface part, as the name implies, is responsible for the graphical interface. The interface is what the user sees, and it is the only thing with which the user can use the program. A good interface should be intuitive and require minimal additional instructions.

The most logical and convenient way to display the interface is to use Unity's internal Inspector. The Unity Inspector is designed so that any global variables declared in the code will be displayed on the Unity Editor stage. These variables will also be available for manual editing. Therefore, if we want to create an object with custom parameters, we will only need to declare these parameters as global variables in the code, after which the user will be able to interact with them from the editor scene and see the result in real time.



■ **Figure 3.1** Example of custom interface created with Unity’s Inspector.

In addition, if we have a need for a more advanced interface that will contain elements not supported by Unity by default, we can resort to free addons developed by other people who have implemented the tools we need for us. Since the central theme of this work is not the interface, I think we can go for it.

3.2.2 Primary input processing

The primary input processing part is responsible for processing the data received from the user and then sending the data to the shader. The data received by the shader should be provided in the most “convenient” form for the shader. Anything that can be computed on the CPU and is not specific to individual pixels on the screen should be computed on the CPU. As mentioned here 1.2.1, each GPU core is much weaker than the CPU cores. Therefore, it is very resource intensive and pointless to perform identical computations for each pixel. Consequently, any such situations must be determined and accounted for in advance.

This part will be written entirely in the C# programming language — Unity’s main programming language. The details will be described in the following sections.

3.2.3 Shader

Finally, the shader part performs the main graphic calculations and produces the final image. This part is the largest and most important part of all. The shader will receive the scene data sent by the processor, such as data about the shapes in the scene, their various modifiers, the hierarchy of Boolean operators, materials, textures, lighting and shadows. The output of the shader will be just one parameter — the color of a pixel. Our main task in this part will be to write optimized code that correctly and efficiently renders the final image according to the given parameters.

In Unity you can create shaders of several types. All of them are mostly templates for creating your own shaders, which are more or less adapted to certain tasks. In this project we will use the `ompute` shader. A `ompute` shader is a type of shader that is not really targeted at any particular task. This makes

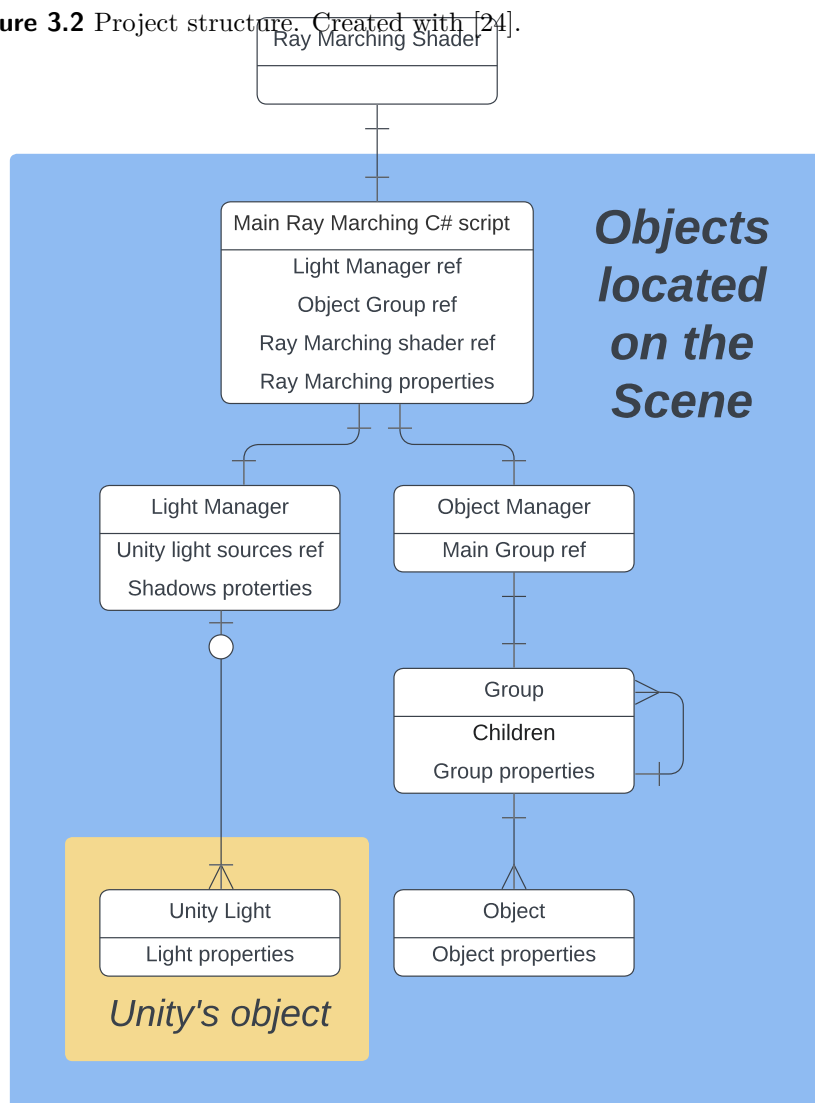
it very flexible, and in our not-so-standard project, it fits perfectly. Also, one of the features of compute shaders in Unity are compute buffers. Compute buffers allow you to send entire arrays of data structures to the shader. For example, if we were using a standard shader, we would have to send data through uniforms, and we could only send data as arrays of numbers. Moreover, the number of bytes of data that can be sent via uniforms is very limited and is not suitable for our purposes.

As for the software aspects of Ray Marching technology, many of its features have already been described by other people. For example, Inigo Quilez in his blog has disassembled and even posted the code of many features that we can use when implementing the project. [6]

3.3 Design

The following diagram shows the structure of the project:

■ **Figure 3.2** Project structure. Created with [24].



Most of the objects are in the “Objects located on the Scene” section. As mentioned here 1.3.3.2, in order for scripts to function (namely, for their Update method to be called), the script must be assigned to an object on the scene as a component. Therefore, even objects that have no physical representation, such as the Light Manager from the diagram, must be on the scene. In such cases it is common to use Unity’s Empty GameObject. Ray Marching shader is the only object that is not on the scene. This is because the shader is not a C# class inherited by MonoBehaviour, but a separate program written in the HLSL shader language.

Let’s analyze the individual elements in the diagram from bottom to top.

At the very bottom are “Object” and “Unity Light”. “Unity Light” is highlighted and described as Unity’s object because it is the only object that will not be written by us. We will use Unity’s built-in lighting to light the objects. This is appropriate, as we would like our procedurally rendered objects to be lit in the same way as regular polygonal models that may be on the scene. An “Object” is essentially a 3D shape rendered by a shader. It has an attribute “Object properties” which is responsible for setting the properties of the shape. This includes customizing the shape of the object, its position in space, material and other possible properties. For the most part, “Object” is just an interface holder that stores data about the shape that is processed by other scripts.

Next comes the “Group” object. This object is used to apply a particular Boolean operation to a group of shapes. As you can see on the diagram, a group can be linked to several “Objects” and several other “Groups”. These relationships are indicated by the “Children” attribute. These imply a hierarchical structure that can be used for various Boolean transformations. You can think of it as a tree whose leaves represent shapes (“Object”) and its other nodes represent groups (“Group”). There is always a group at the root of the tree. Theoretically, a leaf of the tree could also be a group, but that doesn’t make sense. In Unity, this hierarchy will be represented as a hierarchy of objects in the “Hierarchy” window. The “Group properties” attribute denotes various possible customizable group parameters, among which there are at least Boolean operations as at Figure 1.5, which can be applied to child objects.

Above the “Unity Light” entity is the “Light Manager”. This object will store references to the light sources that will take part in lighting the scene, as well as having several configurable parameters, such as soft shadows and ambient occlusion settings. Also here will be “wrapping” the lights in a convenient form for shading in the shader.

To the right is a similar entity — “Object Manager”. Except for shadow settings, the Object Manager will do the same thing as the Light Manager, but for groups and shapes. The input parameter will be a single reference to

the main group, which as descendants will have all other groups and objects present in the scene.

“Main Ray Marching C# script” stores references to “Object Manager”, “Light Manager” and also the main shader. This script does the final sending of all the necessary data to the shader, accepts the result from the shader and displays it on the screen. It will also have parameters to customize the Ray Marching algorithm (maximum number of approximation steps, maximum distance, etc.).

Finally, the “Ray Marching Shader” will render and return the final image based on the received data.

3.4 Implementation

Object

Implementations can be started with a script of shapes. This is the hardest part, from an interface point of view. The object interface should include position, rotation and scale settings, a drop-down list of all available shapes to choose from, settings for each shape, and material settings. Also, the object repetition setting can be added here.

Unity’s Transform component will help us with the first one. The Transform component has attributes Position, Rotation and Scale. This component is present in every Unity object by default. It also allows you to move, rotate or scale the object using the graphical interface on the Scene window.

A drop-down list of available shapes could be implemented using a global Enum variable. However, experimentally it was found out that when the scene is restarted, i.e. every restart of Unity including, the value of the Enum variable is reset. This makes it very difficult to use the tool. Therefore, an alternative way was found. We could write our own custom inspector, but instead we will use other people’s work. We will use the “EditorCools” toolkit [25]. It allows us to create a drop-down list using strings.

Different geometric shapes have different parameters. For example, a sphere would only have a radius, a cone would have a radius and height. We could replace some of these with parameters from the Transform component. So for example instead of having a separate variable that would store 3 radii for an ellipsoid, we could use the 3 parts of the Scale attribute. However, complex shapes may require additional parameters. We would also like these parameters to be displayed in the interface only if the user has selected the corresponding shape from the drop-down list. For this purpose, we will have to use an additional set of tools called “MyBox” [26]. One of the Inspector attributes that this package adds is the ConditionalField attribute, which allows variables to be displayed in the Inspector only under certain conditions. In this way we can create different parameters that will be displayed only when a certain shape is selected.

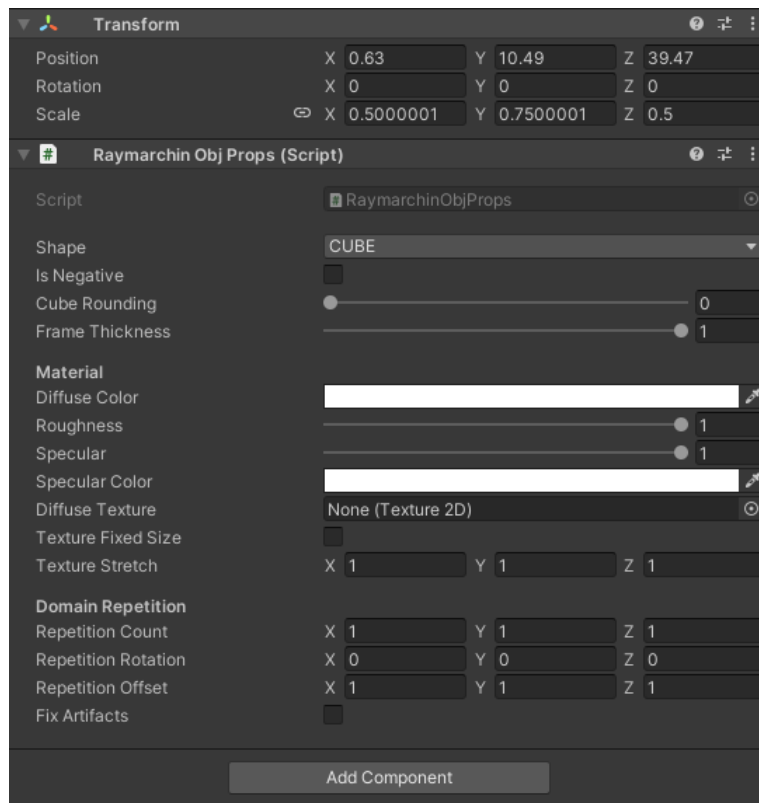
```

[Dropdown(nameof(GetShape))]
public string Shape;
...
[ConditionalField(nameof(Shape), false, "CUBE")]
[Range(0.0f, 1.0f)]
public float CubeRounding;

```

■ **Code listing 3.1** Example of "Dropdown" and "ConditionalField" attributes. The CubeRounding field appears in Inspector only if Shape variable's value is equal to "CUBE".

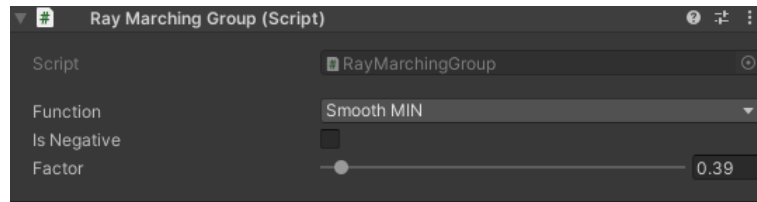
Otherwise, creating the interface for the other shape parameters is not particularly noteworthy, so it will be omitted in this section. The description of their work will be described later in the part about shader implementation.



■ **Figure 3.3** First version of shape's interface. Slider was created with "Range" attribute.

Group

The Group code is one of the smallest in the project. All it does is to store and customize pain operations for objects. As in the case of the Object, we will talk about the details of the implementation of Boolean operations in the part about the shader.

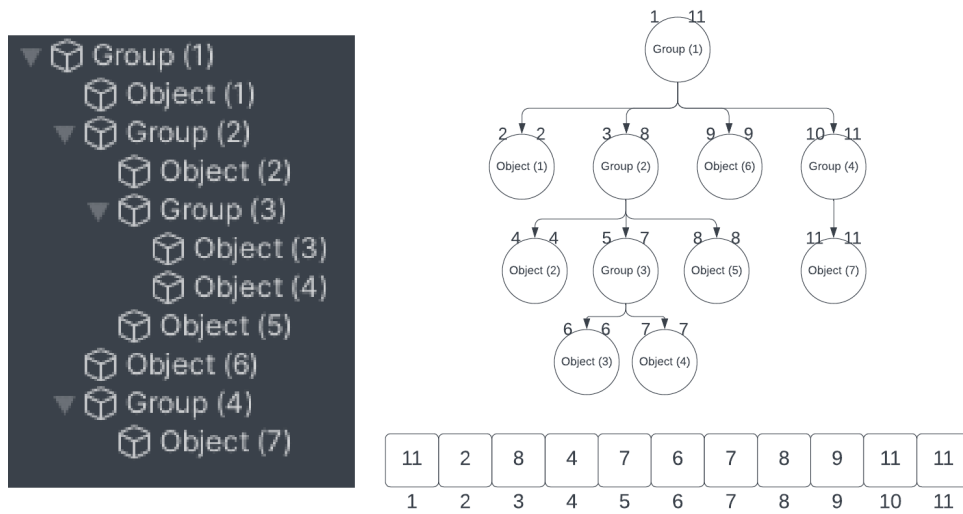


■ **Figure 3.4** Properties of Group.

Object Manager

Despite only one input parameter, the code of this element is quite extensive. Its main idea is to traverse a hierarchical tree consisting of Groups and Objects and somehow pack this tree into an array convenient for shader processing. But how to do it? The problem is that the HLSL language is very limited. Many tools, such as pointers, are not available in HLSL. This is due to the specifics of how shaders function. For this reason, passing a tree and traversing it is not an easy task.

Let's clarify the situation: we want to realize hierarchical application of Boolean operations to objects. To do this, we need to do it in a certain order, "from bottom to top". To do this, we need to pass the whole tree to the shader, preserving its structure. To begin with, we can separately pass an array with objects and an array with a tree. In the tree, instead of storing objects directly, we will store their indexes in the objects array. Thus, in each node of the tree we will store only one number instead of all object properties. The next question is how to pack the tree into an array? In fact, there are several ways to do it. Let's run the DFS algorithm along the tree starting from the root. Let's have a counter, which will be incremented by one at each first entry to a vertex. When leaving a vertex to its parent, we will write the number of the last visited vertex (the counter value) into it. Thus we will have an array of numbers, where the index will mean the number of the node, and the value — the number of the last node of the subtree of this node. All vertices with numbers from "index" to "value" will also lie in the subtree of the node with the number "index". This rule will work for any node in the tree. Thus, the root will be represented by the number "n" at index 1 in the array, where "n" is the number of nodes in the tree.



■ **Figure 3.5** Unity object's hierarchy and its representation in the form of a tree and its array. Created with [24].

As a result, we have a basis for further work. When we write a shader, we will come back to this tree and probably modify it.

We will compose this array every frame. We will also store a copy of the array from the previous frame and compare it with the new one, to detect the moment when objects in the scene were changed. We will do this so that we don't send the array to the shader every frame, but only when the data in it has been changed.

As mentioned above, we divided the object hierarchy into an array with the index tree, and an array with the shape data. The array with shape data will be a list whose unit is a custom structure containing all data about the shape.

```
public struct RayMarchingObject
{
    public int type;
    public int group;
    public float3 position;
    public float3 rotation;
    public float3 scale;
    public otherParameters otherParams; // 12
    public struct otherParameters{...};
};
```

■ **Code listing 3.2** Custom structure to store shapes data. `otherParameters` is another structure which simulates behaviour of float array of 12 elements. We can't use an array directly because of Unity. This "array" is for storing a specific to each shape data.

We will send this array using the compute buffer mentioned above. Its peculiarity is that if we create identical structures in the shader and in the C# script, we can directly send arrays of structure data from the C# script to the shader. This is pretty convenient.

Light Manager

The functionality of the Light Manager script will be similar to the Object Manager, but much smaller. All it does is build an array of light source parameters referenced by the user, and also holds an interface for shadow settings. It is only worth pointing out that in order for the user to specify light sources, it is enough just to create a global variable-list of lights (List <Light> Lights). After that, in the Unity Inspector will automatically appear a window in which user can move any number of lights with drag-n-drop.

Main Ray Marching C# script

This is the final C# script that will send data to the shader. In addition, it will also receive the finished image from the shader and set it. Therefore, we will perform all actions not in the Update method as in other cases, but in the OnRenderImage method. This method is called after each completion of image rendering and takes 2 parameters — two RenderTexture — "source" and "destination". Source texture is the image that was rendered using Unity's built-in rasterization shader. Destination texture is the image we want to see on the screen. Our goal is to take the source texture, augment it with rendered Ray Marching objects, and copy the result into the destination texture.

Let's list the basic data we need to send to the shader. First of all, we should send some data about the camera. Namely its coordinates, forward vector, resolution of image and frustum corners. Frustum corners are 4 normalized vectors that are directed towards the corners of the camera's field of view pyramid. We need them to calculate the ray direction vector using them and the pixel coordinates on the screen. Fortunately, the unity camera has a special method for calculating frustum corners.

```
Vector3[] frustumCorners = new Vector3[4];
CurrentCamera.CalculateFrustumCorners(new Rect(0, 0, 1,
↪ 1),
CurrentCamera.farClipPlane,
↪ Camera.MonoOrStereoscopicEye.Mono, frustumCorners);

Matrix4x4 frustumCornersMat = new Matrix4x4();
frustumCornersMat.SetRow(0,
↪ Vector3.Normalize(CurrentCamera.
transform.TransformVector(frustumCorners[0])));
frustumCornersMat.SetRow(1,
↪ Vector3.Normalize(CurrentCamera.
transform.TransformVector(frustumCorners[1])));
frustumCornersMat.SetRow(2,
↪ Vector3.Normalize(CurrentCamera.
transform.TransformVector(frustumCorners[2])));
frustumCornersMat.SetRow(3,
↪ Vector3.Normalize(CurrentCamera.
transform.TransformVector(frustumCorners[3])));

rayMarchingShader.SetMatrix("_FrustumCornersES",
frustumCornersMat);
```

■ **Code listing 3.3** First frustum corners are calculated. Then they are converted to global coordinates with `CurrentCamera.transform.TransformVector()`, normalized, and set to 4×4 matrix as rows. At the end matrix with frustum corners is sent to the shader.

Finally, we send data about objects, materials and lights to the shader. We store lights, objects and materials in arrays of custom structures. Loading this data into a compute buffer and sending it to the shader, we can do in just a few lines of code:

```
computeBuffer = new  
    ↪ ComputeBuffer(ObjectManager.objBuff.Count,  
    RayMarchingObjectByteSize);  
computeBuffer.SetData(ObjectManager.objBuff.ToArray());  
rayMarchingShader.SetBuffer(rayMarchingShader.  
    FindKernel("CSMain"), "Objects", computeBuffer);
```

■ **Code listing 3.4** Compute buffer creating and setting.

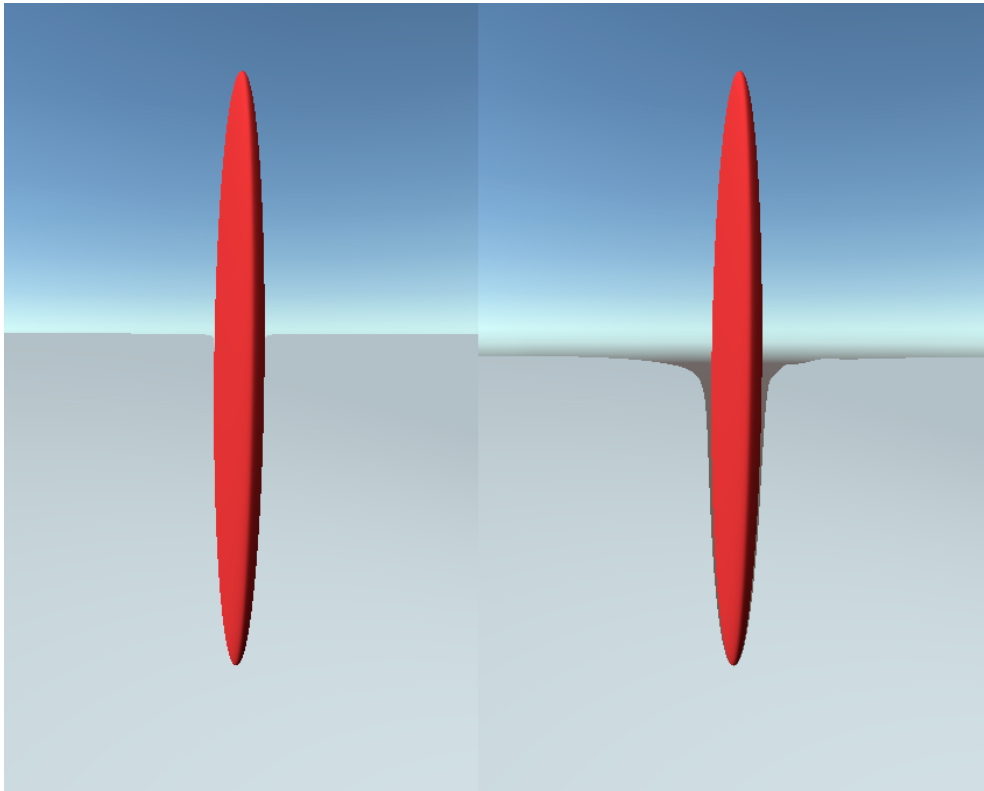
Shader

And here we come to writing the shader. In general, the shader algorithm can be divided into two parts. In the first part we look for the point of contact of the beam with the object, in the second part we illuminate it based on the distance to the object.

To find the point of intersection of the beam with the object, we must follow the following simple algorithm:

1. Start from the camera position.
2. Find the distance to the object (in all directions) from the current position using the SDF function of the object.
3. Move along the ray to this distance.
4. Repeat steps 2 and 3 n-number of times.
5. Done!

The search for the point of contact with the object is carried out in several (hundreds) iterations. From the number of iterations depends on the accuracy of determining the boundaries of objects and therefore the quality of the final picture, but also the amount of load on the system. Therefore, we will allow the user to configure this parameter independently through the interface.



■ **Figure 3.6** Two renders of an ellipsoid and a plane with 300 and 100 iterations. Equation of exact distance to an ellipsoid is too demanding, so we use its approximation. It causes artifacts when the number of iterations is small. [27]

Having found the distance, we can already visualize the object by substituting the depth as the final color. To apply lighting to the object, we must first find the normal of the object at the intersection point. Normal is a vector perpendicular to the object plane. To find it, we can use this function:

```

float3 getNormal(float3 p)
{
    float d = getDist(p).dist;
    float2 e = float2(0.001, 0);
    float3 n = d - float3(getDist(p - e.xyy).dist,
        getDist(p - e.yxy).dist, getDist(p - e.yyx).dist);
    return normalize(n);
}

```

■ **Code listing 3.5** getNormal function. getDist function returns distance from the point to an object.

As you can see, when calculating the normal, we indent a very small value in all 3 directions and look for distances. In this way we perform a direct differentiation:

$$\frac{df(p)}{dx} \approx \frac{f(p) - f(p - \{h, 0, 0\})}{h},$$

where h is as small as possible. You can read more about it here [28].

With the normal in place, we have everything we need to calculate the shading of an object. All we need to do is multiply the color of the object by the scalar product of the normal and light direction vectors (in fact, the cosine of the angle between them) [29]. In the case of directional light, we can take the light direction vector directly. Or, in the case of other types of light, we can calculate it using the coordinates of the illuminated point in space and the coordinates of the light source.

In the same part, we can realize attenuation of point light and spot light. We use the formulas that were given here 1.2.2. Fortunately, the Unity documentation contains code samples with the exact formula for calculating the attenuation. [12]

And finally we can realize glare from specular surfaces. Two of the object parameters we pass are *specular* and *roughness*. We will use the Blinn-Fong model to calculate the glare. The Blinn-Fong model is an improved version of the Fong model, which allows us to achieve better illumination results at sharp camera angles on the surface [30]. Its essence is that we will first calculate the mean vector between the gaze vector and the light incidence vector, and then find the scalar product of this mean vector with the surface normal. The result is a power of $\frac{1}{\text{specular}}$, where *specular* is a parameter we have given to the surface, whose value ranges from 0 to 1. We can use the *roughness* parameter to soften glare. The higher the *roughness* is, the dimmer the glare.

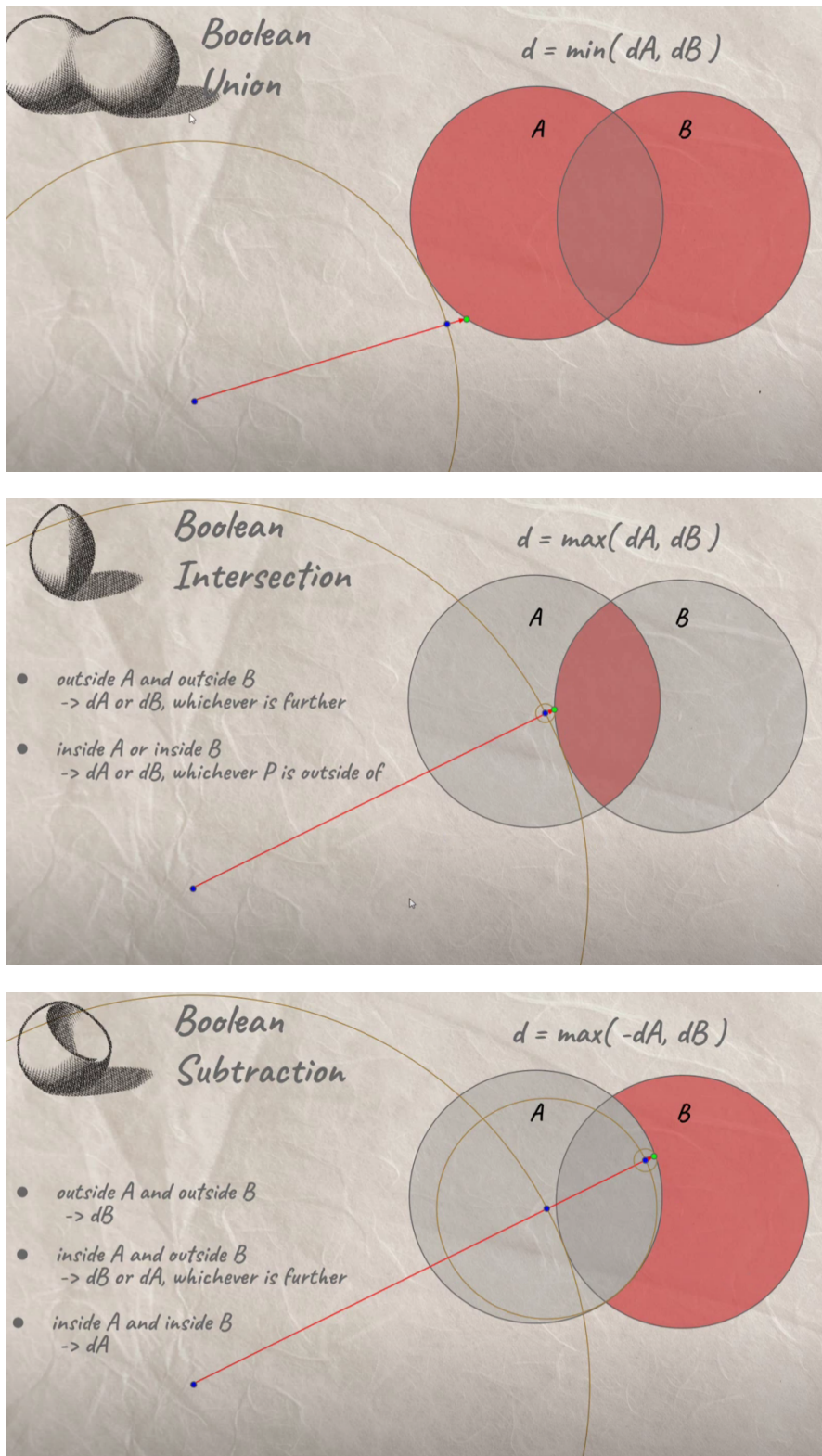
Let's get back to rendering shapes. How to render several shapes on the scene? The most obvious and most used way is to calculate distances for

each object at each iteration, using their SDFs, and take the minimum of all distances. This way we will never miss any of the objects and render an object that should be overlapped by another object. We can say that by using the minimum function we implement the union operation. [31] In addition to union, there are two more operations that would be useful to implement — intersection and subtraction. These functions are realized in a similar way.

For intersection of objects we need to take the maximum of their distances instead of the minimum. In this case, the ray will approach the farther object of the two, and when the ray enters the first (nearest) object, the distance to it becomes negative, the ray will simply take the distance to the second object. To give an example, if the objects do not intersect, the ray will go to infinity. [31]

To subtraction, we need to apply the maximum function in the same way, only one of the objects (the one we are subtracting), we need to invert — multiply its distance by -1. Inverting an object can be visualized as if the entire space around the object becomes “solid” and the object itself becomes empty. If you do this trick with one object in the scene and place the camera inside it, you can see its walls from the inside, as if the object were hollow. In this way, the operation of subtraction can be perceived as an operation of intersection with an inverted object. [31]

The following figures demonstrate the principle of all three operations:



■ **Figure 3.7** Illustrations of three SDF operations. These are screenshots from [31]

That's not all we can do with our objects. Basically, we could use any function with two parameters to get interesting results. In the Ray Marching community, smooth minimum and smooth maximum have become commonplace. The result of work of these functions has been illustrated here 1.6. These functions give an output value close to the minimum/maximum depending on the magnitude of their difference. [32]



■ **Figure 3.8** Example of smooth minimum function. Red and blue are initial functions, purple is the minimum of them, orange is the smooth minimum. Created in Desmos [33].

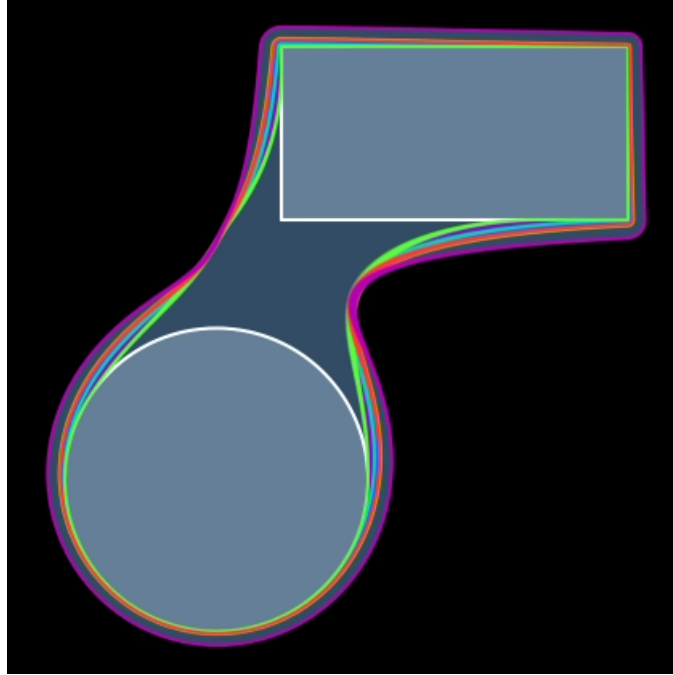
The smooth minimum function also takes a third parameter — coefficient k , which affects the distance at which the result of the smooth minimum will begin to appear. This coefficient k we will allow the user to change through the interface.

```
float smin( float a, float b, float k )
{
    k *= 1.0/(1.0-sqrt(0.5));
    float h = max( k-abs(a-b), 0.0 )/k;
    return min(a,b) - k*0.5*(1.0+h-sqrt(1.0-h*(h-2.0)));
}
```

■ **Code listing 3.6** Circular smooth minimum function code [32].

There are quite a large number of functions of smooth minima. Nevertheless, they are all quite similar, and the differences in their operation are not always visible to the naked eye. For our project I chose one of them —

circular smooth minimum 3.6. I chose this particular function, as it is clear from the demonstrations on this [32] resource that this function is the weakest in causing “volume thickening”.



■ **Figure 3.9** Different smooth minimum functions comparison in 2D space. Green is the circular smooth minimum [32].

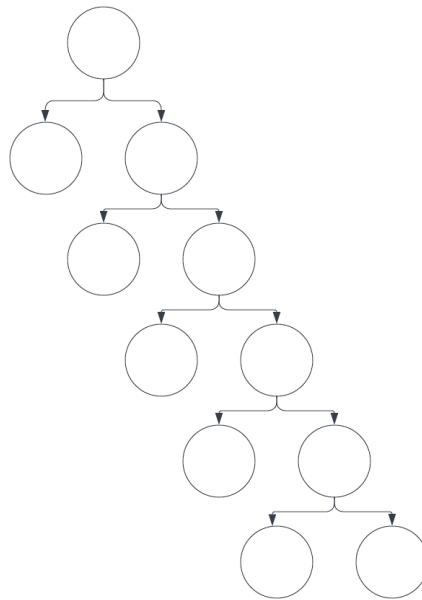
It is not hard to guess that there is an analogous maximum function for the maximum function. So we have four functions — minimum, maximum, smooth minimum and smooth maximum, the last two of which are parametric. That’s quite a lot. With one smooth minimum function, you can create many combinations using different coefficients. For this reason, support for hierarchical processing of combination functions is desirable. Here we return to our tree.

The tree information we pass by array would already suffice for a tree traversal. However, there is one nuance.

HLSL does not support recursion, which is not surprising for a shader. Therefore, we will have to iteratively traverse the tree. For such a traversal, we would need to additionally remember various information, for example, about the children passed. And we would have to store this information in arrays with a constant size. As mentioned here 1.2.1, using arrays is not an easy task for a shader, so we will use shared memory to create them. Shared memory is fast but is available to all threads, so we will divide it into several parts [9]. Let’s create one large two-dimensional array with sizes $t \times n$, where t is the number of threads in the block and n is the size of the one-dimensional

array we need. Thus, each thread will be able to access its memory area by its thread number. Further experiments showed that using shared memory instead of local memory significantly speeds up performance.

However, we still have a memory problem. The amount of shared memory is very limited. Unity generates an error when using more than 32 kilobytes of shared memory of the video card[34]. That's why we have to use it wisely. What exactly are we going to store in our arrays? While taking into account the transfer of all necessary information about the object tree, we must at least store intermediate results when applying combining functions during traversal. We cannot immediately combine objects that are in different groups; we must first count the values of their groups. For this reason, we must memorize the intermediate results. Look at the tree in the following illustration.



■ **Figure 3.10** "Bad" tree for naive tree traversal. Created with [24].

Let's imagine that we go around this tree and connect its elements. We can only connect objects that are in the same group (i.e. have a common parent). If an object is a group, we must first process its children. If we traverse our tree in a "naive" way, an unpleasant situation may occur. For example, if when going around the tree above we keep going down to the leftmost child, we will have to memorize the distances to each child until we get to the last bottom right child and can start "connecting them". So this will take up 5 memory cells. Now imagine that instead of going down to the leftmost child, we go down to the rightmost child. In such a situation, if we connect all available intermediate results at the first opportunity, we would only need 1 memory cell! Thus it becomes clear that the correct order of traversing the tree can

save us a lot of memory.

Now we face the question: which traversal algorithm will be the most optimal? First of all, let's say that we will perform this traversal not in the shader, but in the C# script when building the tree. Thus, we will arrange the subtrees in the order that will be optimal for a naive pass. To understand which algorithm will be optimal, let's think about which tree will require the most memory. We will quickly come to the conclusion that the most "heavy" tree will be a perfect binary tree. A perfect binary tree is a binary tree of height h that has $2^{h+1} - 1$ nodes. Such a tree will require exactly h memory cells, where h is the height of the tree, during its elimination (I will call the sequential connection of nodes elimination). So, my claim is that if we, when eliminating a tree, first go down to the subtree that has the largest perfect binary tree in itself as a subtree, then the total amount of memory we will require is h , where h is the height of the largest perfect binary subtree among all perfect binary subtrees in the given tree. There is no complete proof of this statement, but it has been tested on many cases. Thus, if we assume that this statement is true, then even allocating 10 memory locations will be enough to use at least 512 objects (the number of leaves of a perfect binary tree of height 10).

```

float getDist(float3 p)
{
    int queuePivot = 0;
    int lastProcessedI = 1;
    int currGroupSize = 0;
    int treeDeep = 0;
    int prevChildNode = 0;
    // Find leafe
    for (; lastProcessedI < groupsTreeSize;
        ↪ lastProcessedI++, treeDeep++)
    {
        if (lastProcessedI == groupsTree[lastProcessedI].x)
            break;
    }
    for (int i = lastProcessedI; i != 0;) //start in leafe
    {
        if (i != groupsTree[i].x)
        {
            if (prevChildNode > i + 1)
            {
                objProcQueue[threadId][queuePivot - 2] =
                ↪ applyFunc(objProcQueue[threadId]
                ↪ [queuePivot - 2],
                ↪ objProcQueue[threadId][queuePivot - 1],
                ↪ abs(groupsTree[i].w));
                queuePivot--;
            }
            if (groupsTree[i].x > lastProcessedI)
            {
                i = 1 + lastProcessedI + treeDeep;
                treeDeep++;
            }
            else
            {
                if (groupsTree[i].w < 0)
                {
                    objProcQueue[threadId][queuePivot - 1]
                    ↪ *= -1;
                }
                prevChildNode = i;
                i = groupsTree[i].z;
            }
        }
    }
}

```

```
    else
    {
        objProcQueue[threadId][queuePivot] =
            ↪ calcObjectDist(p, groupsTree[i].w);
        if (groupsTree[i].w < 0)
        {
            objProcQueue[threadId][queuePivot] *= -1.0;
        }
        queuePivot++;
        lastProcessedI = i;
        treeDeep = 0;
        prevChildNode = i;
        i = groupsTree[i].z;
    }
}
return objProcQueue[threadId][0];
}
```

■ **Code listing 3.7** Shader's function, which performs common tree traversal. Tree is constructed in an optimal way already. `objProcQueue` is array allocated in GPU's shared memory. `calcObjectDist` calculates distance from point to object, using its special SDF (depends on object). `applyFunc` applies combining function (min, max, `smooth_min`, `smooth_max`) to distances.

So, we decided that with our method we have enough 10 memory cells to traverse the tree. However, this applies only to distances, which we can store with a single float number. Nevertheless, we also have the materials of the objects. We have to mix the materials in a similar way. We can use a function from here to mix them, but that's not the point right now. If we have 10 cells to store intermediate values for calculating distances to objects, we must also have 10 cells for materials. However, unlike distances, we need more memory to store materials than we do to store numbers. Namely 11 times more, in our case (3 numbers for albedo color, 3 numbers for color from texture, 3 numbers for specular color, 2 numbers for specular settings). We could optimize this number by reducing the number of settings or combining albedo color and texture color, but the final value will still be large. And this number will be large because we allocate arrays for all threads at the same time. And if we calculate, $10 \text{ cells} \times 11 \text{ material numbers} \times 4 \text{ bytes} \times 64 \text{ threads} = 28,160 \text{ bytes}$ of memory! And even though this amount does not exceed the limit, experiments have shown that allocating such large arrays in shared memory greatly reduces performance. Therefore, it was decided to move away from memorizing intermediate values when calculating the final

material and store only coefficients of how much the material of each object affects the final result. Thus, we will need to store only 1 number, but for each material, which will not allow us to create a scene with a large number of materials (for example, several dozens).

```

MaterialMix getMaterial(float3 p, float3 n)
{
    int queuePivot = 0;
    int coefsPivot = 0;
    int lastProcessedI = 1;
    int treeDeep = 0;
    int prevChildNode = 0;
    float k = 0;
    int refI = 0;
    int j;
    // Find leafe
    for (; lastProcessedI < groupsTreeSize;
        ↪ lastProcessedI++, treeDeep++)
    {
        if (lastProcessedI == groupsTree[lastProcessedI].x)
            break;
    }
    for (int i = lastProcessedI; i != 0;) //start in leafe
    {
        if (i != groupsTree[i].x)
        {
            if (prevChildNode > i + 1)
            {
                k = getMixFactor(queuePivot - 2, queuePivot
                    ↪ - 1, abs(groupsTree[i].w));
                refI =
                    ↪ int(objMaterialCoefs[threadId][coefsPivot
                    ↪ - 1] / 10.0);
                for (j = coefsPivot - 1; ((j >= 0) &&
                    ↪ (int(objMaterialCoefs[threadId][j] /
                    ↪ 10.0) == refI)); j--)
                {
                    objMaterialCoefs[threadId][j] = 10 * i +
                        ↪ (objMaterialCoefs[threadId][j] % 10)
                        ↪ * k;
                }
            }
        }
    }
}

```



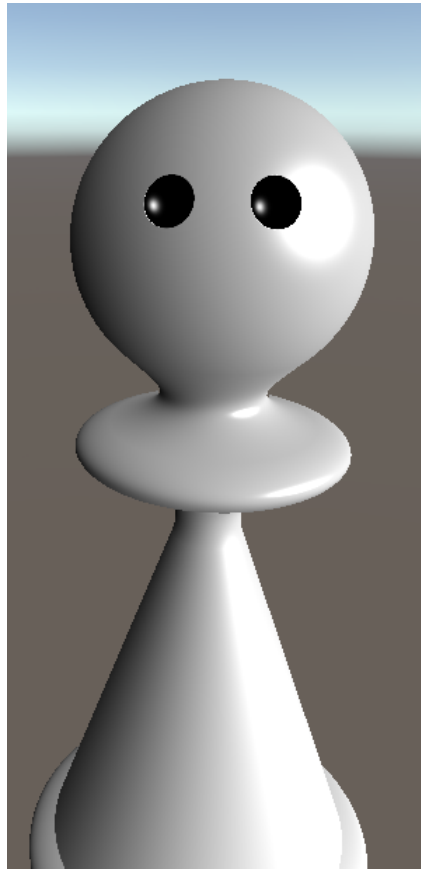
```

    for (; ((j >= 0) &&
        ↪ (int(objMaterialCoefs[threadId][j] / 10)
        ↪ >= i)); j--)
    {
        objMaterialCoefs[threadId][j] = 10 * i +
        ↪ (objMaterialCoefs[threadId][j] % 10)
        ↪ * (1 - k);
    }
    objProcQueue[threadId][queuePivot - 2] =
    ↪ applyFunc(objProcQueue[threadId][queuePivot
    ↪ - 2], objProcQueue[threadId][queuePivot
    ↪ - 1], abs(groupsTree[i].w));
    queuePivot--;
}
if (groupsTree[i].x > lastProcessedI)
{
    i = 1 + lastProcessedI + treeDeep;
    treeDeep++;
}
else
{
    if (groupsTree[i].w < 0)
    {
        objProcQueue[threadId][queuePivot - 1]
        ↪ *= -1;
    }
    prevChildNode = i;
    i = groupsTree[i].z;
}
}
else
{
    objProcQueue[threadId][queuePivot] =
    ↪ calcObjectDist(p, groupsTree[i].w);
    objMaterialCoefs[threadId][coefsPivot] = 1.0;
    if (i == groupsTree[i].z + 1)
        objMaterialCoefs[threadId][coefsPivot] +=
        ↪ (groupsTree[i].z * 10.0);
}
}

```

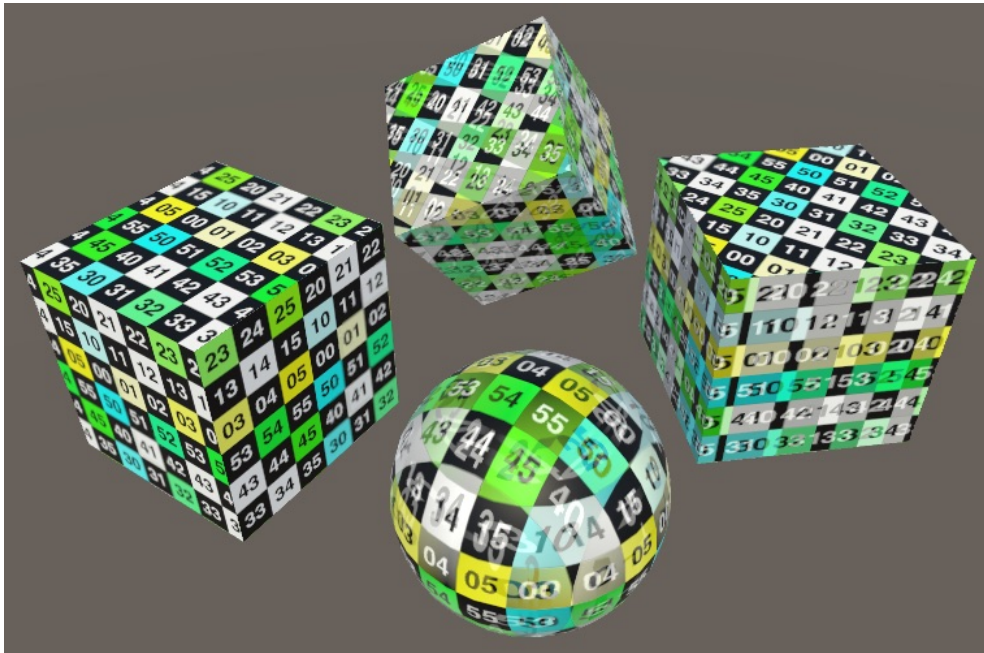
```
        if (groupsTree[i].w < 0)
        {
            objProcQueue[threadId][queuePivot] *= -1.0;
        }
        queuePivot++;
        coefsPivot++;
        lastProcessedI = i;
        treeDeep = 0;
        prevChildNode = i;
        i = groupsTree[i].z;
    }
}
MaterialMix new_m;
for (j = 0; j < coefsPivot; j++)
{
    new_m.color += Materials[j].color *
        ↪ (objMaterialCoefs[threadId][j] % 10.0);
    new_m.roughness += Materials[j].roughness *
        ↪ (objMaterialCoefs[threadId][j] % 10.0);
    new_m.specular += Materials[j].specular *
        ↪ (objMaterialCoefs[threadId][j] % 10.0);
    new_m.specularColor += Materials[j].specularColor *
        ↪ (objMaterialCoefs[threadId][j] % 10.0);
    new_m.texColor += getTexColor(j, n, p) *
        ↪ (objMaterialCoefs[threadId][j] % 10.0);
}
return new_m;
}
```

■ **Code listing 3.8** Modified tree traversal for materials' coefficients calculation.



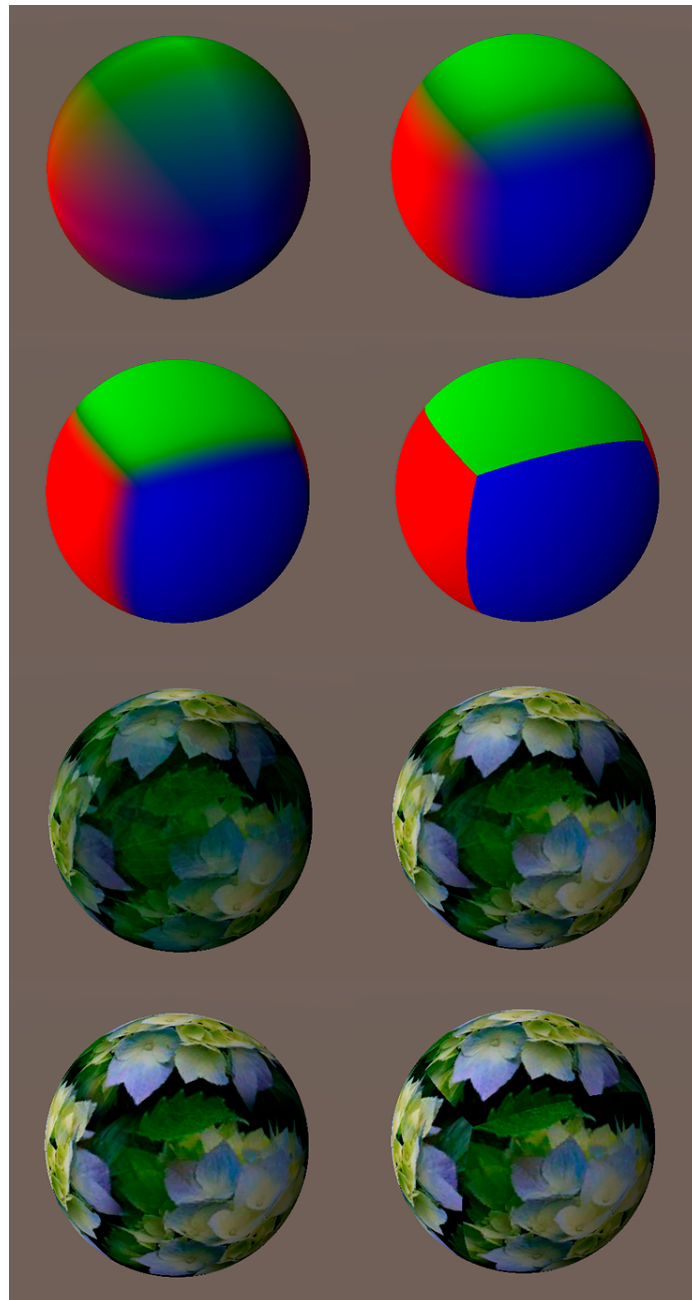
■ **Figure 3.11** A pawn, created with hierarchical shapes combinations.

Back to the material. There's another interesting thing I'd like to talk about, and that's texturing. In normal 3D graphics, when we texture an object, we use special UV coordinates that are defined for each vertex of a polygonal object. But how do you apply a texture to a procedurally generated object? This is where triplanar mapping can help us. Triplanar mapping is a method of texture mapping, when the texture “envelops” the object from all three spatial axes [35].



■ **Figure 3.12** Example of using triplanar mapping [35].

To do this, we first need to calculate the texture pixel coordinates for the three planes — XY , XZ , YZ . We can do this by taking the coordinates of the textured point modulo 1 and normalizing the result. After that we need the normal, which we already know. With the normal, we can understand how each texture contributes to each side. In this way, texture blending will occur on sloped surfaces like a sphere. We can reduce the effect of blending by taking the normal to a higher degree and normalizing. In this way we will increase the influence of a particular side, and reduce the seam between the joints of the textures.



■ **Figure 3.13** Different powers of norm comparison. 1, 4, 10 and infinite powers are demonstrated.

In addition, in order to display the texture correctly, we need to apply to the point where we are looking for the texture value and to the normal all the same spatial operations (translation, rotation, scale) that we applied to the object before.

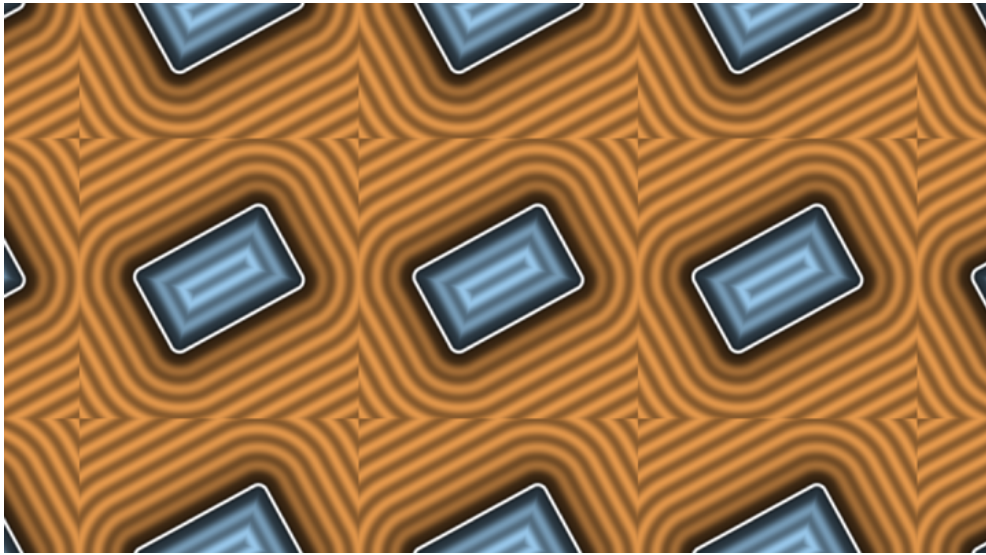
```
float3 getTexColor(int obj, float3 n, float3 p)
{
    n = rotate(n, Objects[obj].rotation);
    n = abs(oneNormNormalize(pow(n, 10)));
    float3 texUV = rotate(p - Objects[obj].position,
        ↪ Objects[obj].rotation);
    if (!Materials[obj].texFixed)
        texUV = ((texUV % (Objects[obj].scale *
            ↪ Materials[obj].texStretch)) / (2.0 *
            ↪ Objects[obj].scale * Materials[obj].texStretch)
            ↪ + 0.5);
    else
        texUV = ((texUV % Materials[obj].texStretch) / (2.0
            ↪ * Materials[obj].texStretch) + 0.5);
    texUV *= TEXTURE_SIZE;
    float3 texXY = textures[float3(texUV.xy, obj)].xyz;
    float3 texXZ = textures[float3(texUV.xz, obj)].xyz;
    float3 texYZ = textures[float3(texUV.yz, obj)].xyz;

    return float3(n.z * texXY + n.y * texXZ + n.x * texYZ);
}
```

■ **Code listing 3.9** Function, which returns texture color at specific point.

Another feature I would like to tell you about is domain repetition (object repetition). This function allows you to repeat an object along any axis (or all of them at once) an unlimited number of times with almost no loss of performance. This is achieved by applying the module function to a ray, which makes the ray pass through a limited space, encountering the same object time after time [36].

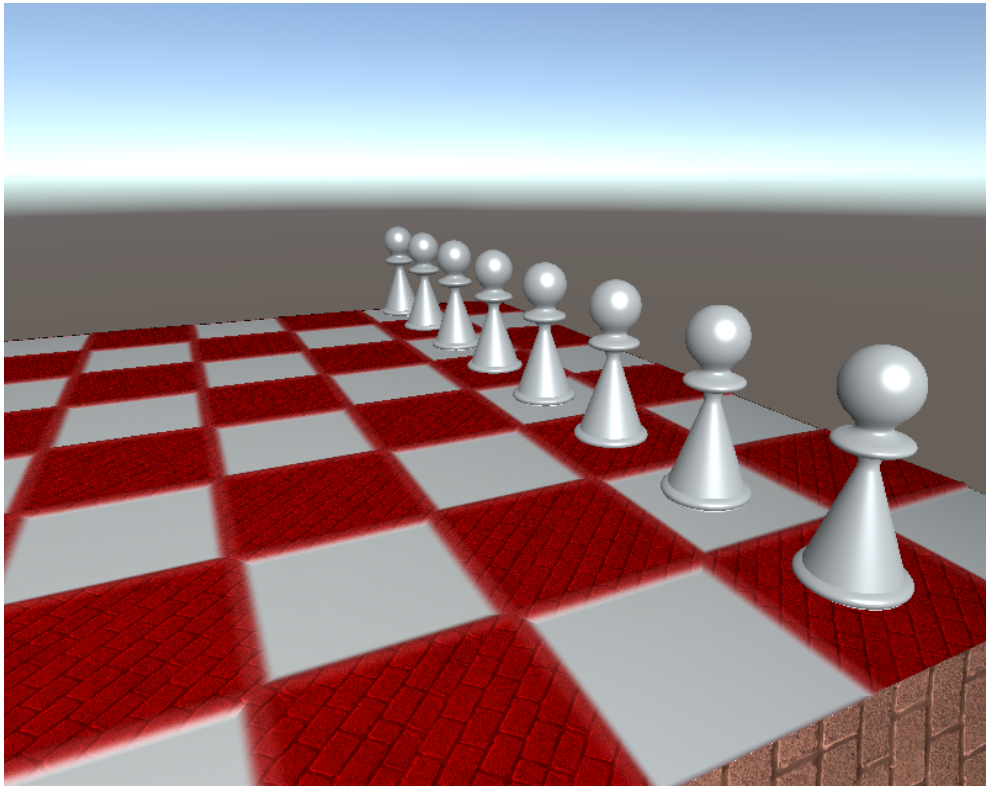
However, not everything is so simple. The whole point is that an object that passes through the bounded space time after time sees only one instance of the object. But in theory, it could happen that the closest object is the instance of the object that is kind of farther away in the next iterations. Take a look at the following figure:



■ **Figure 3.14** Example of non-symmetric object repetition in 2D space. Lines shows space with the same distance to the closest object (“distance lines”). [36]

As you can see, the “distance lines” suffer discontinuities when moving from the area of one instance to another. These discontinuities seem insignificant, but in three-dimensional space they will cause serious artifacts. These inaccuracies appear in those situations when the object ceases to be symmetrical with respect to three spatial axes. To correct this situation we need to additionally check distances to the nearest neighboring instances along all spatial axes. That is, in 3D we need to additionally check 7 neighbors. So, the performance reduction is 8 times. This is not much, considering that in this way we can generate an unlimited number of objects [36].

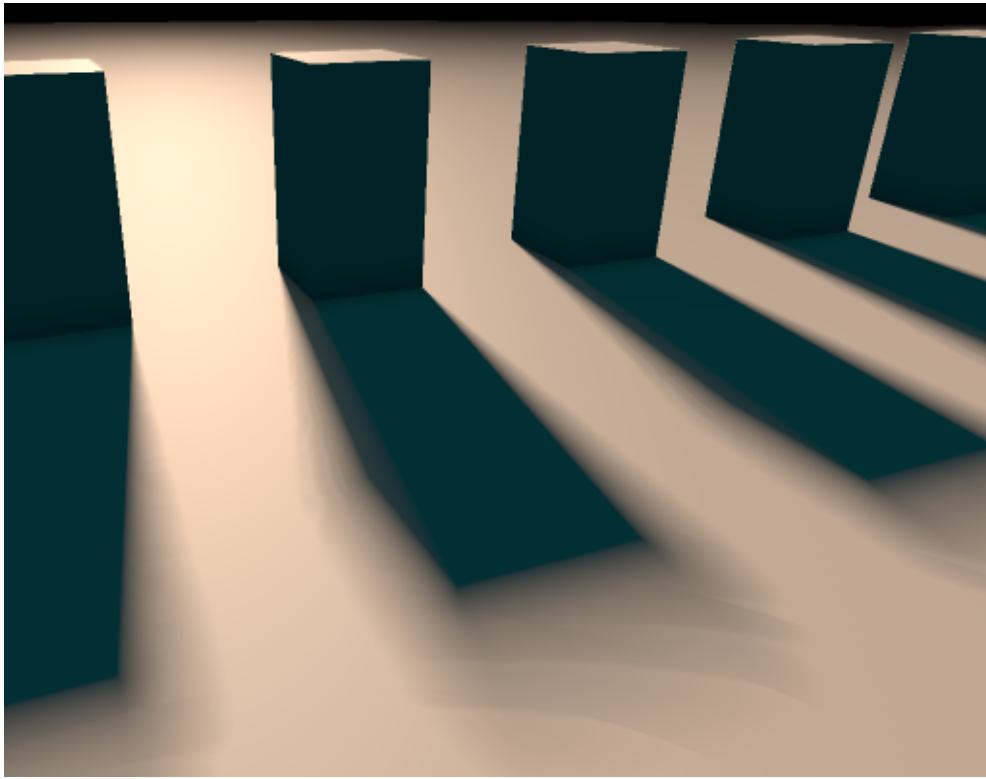
Since this problem may arise only for asymmetric ones, we can leave it up to the user to enable neighbor checking.



■ **Figure 3.15** Example of implemented domain repetition feature. Checkerboard and pawns are created with domain repetition.

Now let's talk about shadows. Soft shadows are quite easy to realize with RayMarching. The basic idea is as follows: we will move from the point where we are looking for a shadow towards the light source in an iterative way, just like when we are looking for an intersection point with an object. At the same time we will memorize the minimum distance to the nearest object. Thus, if our ray does not reach the light source, we draw a hard shadow. If it did, but the minimum distance is small enough — we draw a soft shadow. And finally, if the minimum distance is relatively large, the point is fully illuminated, there is no shadow [37].

However, this approach is not completely optimal. In some situations it may generate artifacts. This happens with angular objects, when the beam goes over the corner of the object and does not capture the minimum distance correctly.



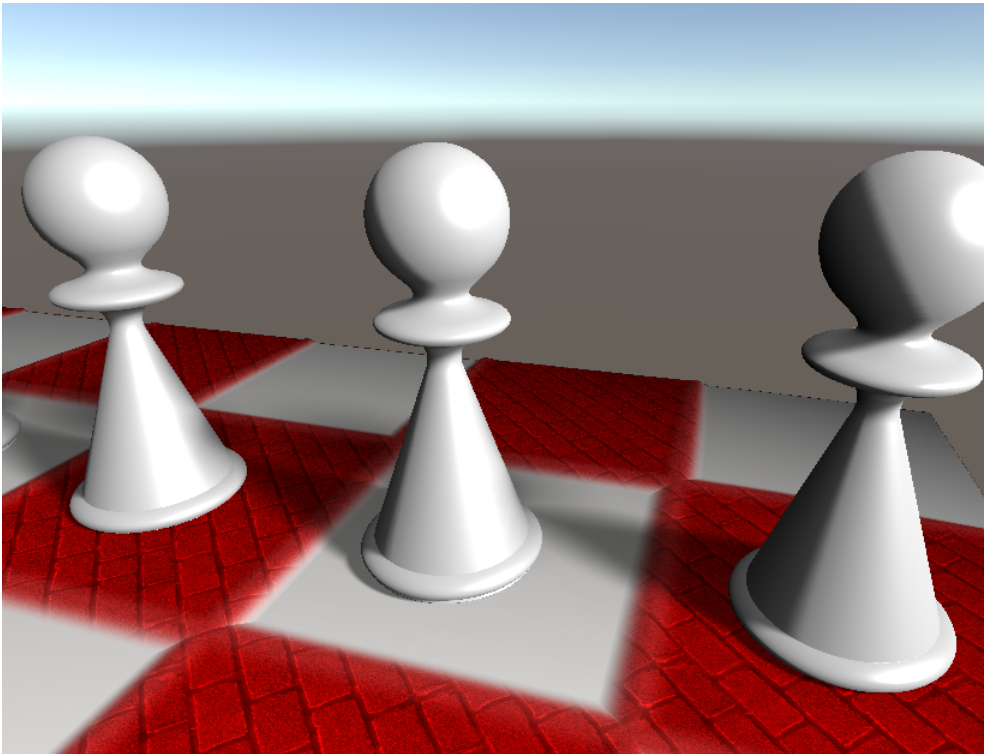
■ **Figure 3.16** Artifacts on shadow [37].

There are two improved methods. One is that we will limit the minimum and maximum step of the beam, and thereby allow it to go inside the geometry of the object. We will interrupt the loop only if the beam goes really deep inside the object and the shadow is really hard. This method has shown better results than the second one, so we will use it.

```
float softShadow(float3 p, float3 dir)
{
    float res = 1.0;
    float mint = _ShadowsMinDistance;
    float maxt = _ShadowsMaxDistance;
    float k = _ShadowsSoftness;
    float mins = _ShadowsMinStep;
    float maxs = _ShadowsMaxStep;
    float iter = _ShadowsIterations;

    for (int i = 0; i < iter && mint < maxt; i++)
    {
        float h = getDist(p + mint * dir);
        res = min(res, h / (k * mint));
        mint += clamp(h, mins, maxs);
        if (res < -1.0 || mint > maxt)
            break;
    }
    res = max(res, -1.0);
    return 0.25 * (1.0 + res) * (1.0 + res) * (2.0 - res);
}
```

■ **Code listing 3.10** Soft shadows function. Taken and modified from [37]



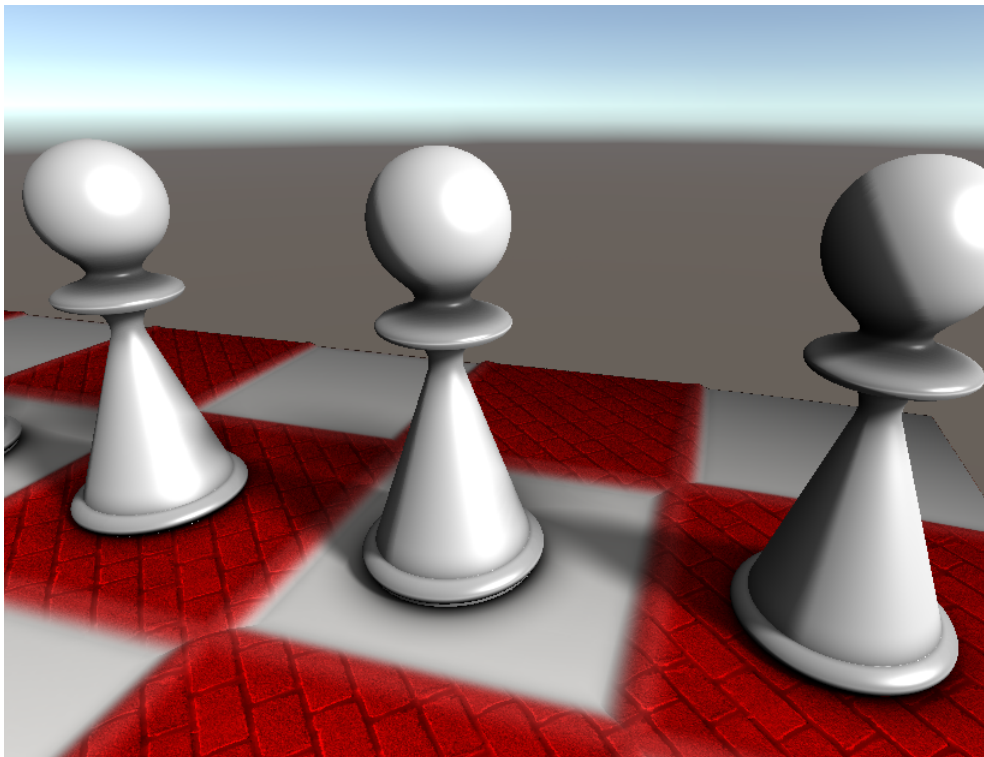
■ **Figure 3.17** Implemented soft shadows.

The implementation of the ambient occlusion effect is quite simple. All we will do is to walk a constant amount along the normal, from the point where we are looking for shading, and measure the distance to the nearest object. If this distance is less than the length of the traveled path, then we add their quotient to the total shading coefficient.

This method allows to achieve good results with small iterations.

```
float ambientOcclusion(float3 p, float3 n)
{
    float step = _AmbientOcclusionStep;
    float k = _AmbientOcclusionIntensity;
    float iter = _AmbientOcclusionIterations;
    float ao = 0.0;
    float dist;
    for (int i = 1; i < iter; i++)
    {
        dist = step * i;
        ao += max(0.0, (dist - getDist(p + n * dist)) /
            ↪ dist);
    }
    return (1 - ao * k);
}
```

■ **Code listing 3.11** Ambient occlusion function. Taken and modified from [38]



■ **Figure 3.18** Implemented ambient occlusion.

3.5 Testing

Testing was done for the most part manually. Each of the implemented features was tested both separately and in combination with the others. Each feature works properly. The overall picture quality is satisfactory. No critical bugs were found during the whole time of work. We have noticed frame per second drops in frames with a large number of objects, as well as particularly strong drops when increasing iterations for soft shadows.

As for the interface, it is worth noting some imperfections.

The coordinates of an object of the group type do not depend on the location of child objects, so it may be problematic to manipulate groups in space.

The slider for adjusting the smooth minimum and maximum factor has limits from 0 to 10. Since the smooth minimum function is directly dependent on the scale of the objects, it can be problematic to adjust the smoothing factor using the slider when the scale of the objects is small.

Using a hierarchy makes it difficult to fine-tune the location of objects. In Unity, the coordinates of objects in the Transform component are displayed in local space. Because of this, it can be difficult to evenly position objects that are in different groups relative to each other.

The functions of smooth minimum and smooth maximum do not always produce the desired result, for example, thickening the objects at their junction. This may require more sophisticated tuning of the hierarchy of shape combinations to achieve the desired result.

At the moment all textures sent to the shader are compressed to 512×512 . This was done to simplify sending textures to the shader using a texture array. Because of this, the textures may look blurry when stretched.

Texture mapping using smooth minimum and maximum functions may produce not quite the expected result at shape junctions. It is especially noticeable in dynamics.

In general, the final program performs its tasks with performance acceptable for modern computers and can be useful for creating certain unusual scenes.

Conclusion

The goal of this thesis was to create a convenient tool for the Unity engine to create objects and scenes using Ray Marching technology.

At the beginning of this thesis we analyzed existing rendering technologies, described the main differences of Ray Marching technology and identified its main advantages and key features.

Then we described the necessary technologies for our project. We found out what shaders are and how graphics processors are organized on a basic level. We also analyzed what graphical visual effects developers emphasize today.

After that, we identified the differences of modern game engines; we decided what factors are important for us in our project. Based on this, we found out why the Unity engine is the most suitable for us. Also at the end we talked about the basics of Unity, its design and interface.

Then we analyzed what solutions to our problem exist at the moment. We noted their strengths, noted interesting solutions of some of them and talked about what they lack. On the basis of this we defined the main key feature and difference of our program — the hierarchy of object association.

In the practical part we once again repeated the main features that we should implement, as well as made a general design of the program and defined its structure.

Next, we proceeded to the implementation, during which we discussed in detail the work of each of the features we implemented and demonstrated their work in figures.

Finally, in the testing part we noted all possible difficulties and non-idealities concerning the program performance, interface usability and the quality of the final picture.

In general, we can say that the tool fully copes with the tasks set in the beginning. At the same time, it is also worth noting the great potential for the development of the project. There is still a huge space for adding new features regarding Ray Marching, as well as for improving the already implemented

ones and eliminating the problems described in the testing part.

Bibliography

1. KAERTNER, Stefan. What is 3D rendering. *What is 3D Rendering? / Understanding the 3D Visualization Process* [online]. 2023 [visited on 2024-04-24]. Available from: <https://www.realspace3d.com/resources/what-is-3d-rendering/>.
2. ANGELIA, Tifany. Rendering techniques in Computer Graphics. *Medium* [online]. 2020 [visited on 2024-04-24]. Available from: <https://medium.com/@tifangel/rendering-techniques-in-computer-graphics-504e6134fea4>.
3. CAULFIELD, Brian. What's the difference between Ray Tracing, rasterization? *NVIDIA Blog* [online]. 2020 [visited on 2024-04-24]. Available from: <https://blogs.nvidia.com/blog/whats-difference-between-ray-tracing-rasterization/>.
4. *IMDb* [online]. 1992 [visited on 2024-04-24]. Available from: <https://www.imdb.com/title/tt0304947/>.
5. WONG, Jamie. Ray Marching and signed distance functions. *Jamie Wong* [online]. 2016 [visited on 2024-04-24]. Available from: <https://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/>.
6. QUILEZ, Inigo. Distance functions. *Inigo Quilez* [online]. 2024 [visited on 2024-04-24]. Available from: <https://iquilezles.org/articles/distfunctions/>.
7. QUILEZ, Inigo. Raymarching distance fields. *Inigo Quilez* [online]. 2008 [visited on 2024-04-24]. Available from: <https://iquilezles.org/articles/raymarchingdf/>.
8. DALRYMPLE, Robert. *GPU memory* [online]. 2014. [visited on 2024-05-15]. Available from: <https://www.ce.jhu.edu/dalrymple/classes/602/Class13.pdf>.

9. MILAKOV, Maxim. *Fast dynamic indexing of private arrays in CUDA* [online]. 2022. [visited on 2024-05-15]. Available from: <https://developer.nvidia.com/blog/fast-dynamic-indexing-private-arrays-cuda/#entry-content-comments>.
10. DE VRIES, Joey. Lighting. *LearnOpenGL* [online]. 2018 [visited on 2024-05-06]. Available from: <https://learnopengl.com/PBR/Lighting>.
11. DE VRIES, Joey. Light casters. *LearnOpenGL* [online]. 2015 [visited on 2024-05-06]. Available from: <https://learnopengl.com/Lighting/Light-casters>.
12. TECHNOLOGIES, Unity. *LightProbes.coefficients* [online]. 2018. [visited on 2024-05-13]. Available from: <https://docs.unity3d.com/400/Documentation/ScriptReference/LightProbes-coefficients.html>.
13. WESTER, Alexander. Ray-tracing soft shadows in real-time. *Medium* [online]. 2020 [visited on 2024-05-06]. Available from: <https://medium.com/@alexander.wester/ray-tracing-soft-shadows-in-real-time-a53b836d123b>.
14. DE VRIES, Joey. SSAO. *LearnOpenGL* [online]. 2015 [visited on 2024-05-06]. Available from: <https://learnopengl.com/Advanced-Lighting/SSAO>.
15. GAME-ACE. *Game engine comparison* [online]. 2023. [visited on 2024-05-10]. Available from: <https://game-ace.com/blog/game-engine-comparison/>.
16. VANCO, Matej. *Raymarcher: Game toolkits* [online]. 2024. [visited on 2024-05-08]. Available from: <https://assetstore.unity.com/packages/tools/game-toolkits/raymarcher-168069#description>.
17. VANCO, Matej. *Fractal sailor - gameplay trailer* [online]. YouTube, 2024 [visited on 2024-05-08]. Available from: https://www.youtube.com/watch?v=Lk1RyY_L96c.
18. DEHAIRS, Ward. *FERM* [online]. 2021. [visited on 2024-05-08]. Available from: <https://assetstore.unity.com/packages/vfx/shaders/fullscreen-camera-effects/ferm-140678>.
19. WATTERS, Kevin; RAMALLO, Fernando. *Raymarching Toolkit for Unity* [online]. 2018. [visited on 2024-05-08]. Available from: <https://kev.town/raymarching-toolkit/>.
20. WATTERS, Kevin; RAMALLO, Fernando. *Character creator* [online]. 2018. [visited on 2024-05-08]. Available from: <https://kev.town/raymarching-toolkit/examples/character-creator/>.
21. ANIKETRAJNISH. *A raymarching engine for Unity* [online]. 2021. [visited on 2024-05-08]. Available from: <https://github.com/aniketrajnish/Raymarching-Engine-Unity>.

22. THEALLENCHOU. *Unity-ray-marching: Ray marching sandbox* [online]. 2020. [visited on 2024-05-08]. Available from: <https://github.com/TheAllenChou/unity-ray-marching>.
23. NOBY. *Shadertoy* [online]. 2017. [visited on 2024-05-08]. Available from: <https://www.shadertoy.com/view/111BDM>.
24. INC., Lucid Software [online]. 2010. [visited on 2024-05-15]. Available from: <https://www.lucidchart.com/>.
25. DATSFAIN. *Unity editor tools* [online]. 2022. [visited on 2024-05-11]. Available from: <https://github.com/datsfain/EditorCools>.
26. DEADCOWS. *MyBox is a set of attributes, tools and extensions for Unity* [online]. 2018. [visited on 2024-05-11]. Available from: <https://github.com/Deadcows/MyBox>.
27. QUILEZ, Inigo. *Ellipsoid SDF* [online]. 2008. [visited on 2024-05-13]. Available from: <https://iquilezles.org/articles/ellipsoids/>.
28. QUILEZ, Inigo. *Normals for an SDF* [online]. 2015. [visited on 2024-05-12]. Available from: <https://iquilezles.org/articles/normalsSDF/>.
29. DE VRIES, Joey. Basic Lighting. *LearnOpenGL* [online]. 2015 [visited on 2024-05-13]. Available from: <https://learnopengl.com/Lighting/Basic-Lighting>.
30. DE VRIES, Joey. Advanced Lighting. *LearnOpenGL* [online]. 2015 [visited on 2024-05-13]. Available from: <https://learnopengl.com/Advanced-Lighting/Advanced-Lighting>.
31. CODE, The Art of. *Raymarching: Basic Operators* [online]. 2019. [visited on 2024-05-13]. Available from: <https://www.youtube.com/watch?v=AfKGMUDWfuE>.
32. QUILEZ, Inigo. *Smooth minimum* [online]. 2013. [visited on 2024-05-13]. Available from: <https://iquilezles.org/articles/smin/>.
33. STUDIO, Desmos. *Let's learn together*. [online]. 2020. [visited on 2024-05-13]. Available from: <https://www.desmos.com/>.
34. TECHNOLOGIES, Unity. *Scripting API* [online]. 2022. [visited on 2024-05-15]. Available from: <https://docs.unity3d.com/ScriptReference/index.html>.
35. FLICK, Jasper. *Triplanar mapping* [online]. Catlike Coding, 2018 [visited on 2024-05-16]. Available from: <https://catlikecoding.com/unity/tutorials/advanced-rendering/triplanar-mapping/>.
36. QUILEZ, Inigo. *Domain repetition* [online]. 2008. [visited on 2024-05-16]. Available from: <https://iquilezles.org/articles/sdfrepetition/>.
37. QUILEZ, Inigo. *Soft shadows in raymarched SDFs* [online]. 2010. [visited on 2024-05-16]. Available from: <https://iquilezles.org/articles/rmshadows/>.

38. PEERPLAY. *Raymarching Shader - Unity CG/C# tutorial _chapter[8] = "ambient occlusion"; //peerplay* [online]. YouTube, 2019 [visited on 2024-05-16]. Available from: <https://www.youtube.com/watch?v=22PZF7fWLqI>.