VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
INSTITUTE OF COMPUTER SCIENCE

Course work

Raymarching rendering implementation in the Unity game engine

(Spindulio žygiavimo atvaizdavimo technologijos implementacija Unity žaidimų variklyje)

By: 4th course 3 group student
Liutauras Gaidamavičius (signature)

Supervisor:
lect. Irus Grinis (signature)

Vilnius
2021

Turinys

Introduction

Background

In the field of real-time computer graphics, rasterization has been the primary method of rendering surfaces for more than two decades. Even though creating a realistic and physically correct image is a challenge when rasterization, and the requirement to polygonize surfaces into triangles creates challenges when rendering more complex surfaces as well as loses detail, it's performance has been the main reason it is a staple method in the field of computer graphics.

However with GPU cycle speed exponentially increasing in the past years, methods that were only being used in offline rendering started to be used in real-time applications too. These methods try to simulate actual light rays coming off a light source, bouncing off of objects and landing inside the observers "eye". One of these methods, ray marching, uses signed distance functions of implicit surfaces. This allows nearly any shape to be easily rendered by simply defining its distance function. Also, 3D geometry data is always available in the rendering pipeline which simplifies many rendering features like shadows or ambient occlusion.

However, currently no mainstream real-time rendering engine supports raymarching as the main way to render scenes, and almost all still use rasterization. In the past few years there has been a rise of realtime ray tracing solutions, but raymarching isn't yet looked at as a mainstream rendering technology.

Purpose

The main goal of this project is to implement raymarching as the main rendering method in the Unity game engine. Unity was chosen due to it being one of the two most popular real-time 3D rendering engines and being heavily customizable. Since many users of Unity are artists, game designers, etc. and do not have graphics programming knowledge, the implementation should be done in a way where it is easy to use by user's with relatively low experience.

Another goal is to use this as an opportunity to implement distance field transformations that allow to transform primitives into more complex shapes, while focusing on keeping the features easily editable and customizable.

1. Raymarching

1.1. Surface Representation

While in rasterization surfaces are represented by collections of triangles, raymarching uses implicit functions, that define whether a specific point is inside, outside, or directly on the surface being rendered. The specific type of function being used is signed distance functions. These functions, when passed the coordinates of a point in space, return the shortest distance between that point and some surface[Won16]. An example of such a function for a sphere surface would look like this:

$$f(x,y,z) = \sqrt{x^2 + y^2 + z^2} - r$$

The function must return positive and negative values as a way to indicate whether the point being sampled is inside or outside the surface.

1.2. Sphere Tracing

The specific raymarching algorithm used in this project is Sphere Tracing. This technique is first and foremost based on defining light rays that bounce from lit surfaces towards the observer. However, since most of the light bouncing of a surface never reaches the observer, we can define the the rays ”backwards”, i.e. going from the observer towards the lit surface. We will define the ray equation as:

$$r(t) = r_o + tr_d \tag{1}$$

Here r_o depicts the point in space in which the ray originates, r_d is the direction vector of the ray and t is the distance travelled along the ray. The result, $r(t)$ is the point in space after the ray has travelled the distance of t.

Instead of trying to find the ray-surface intersection, which is difficult to compute for most non-basic primitives, we will step forward the ray, sampling our signed distance function at every step, until the distance function returns a small enough value where we can definitively say that the sampled point is on the surface, thus finding the intersection.

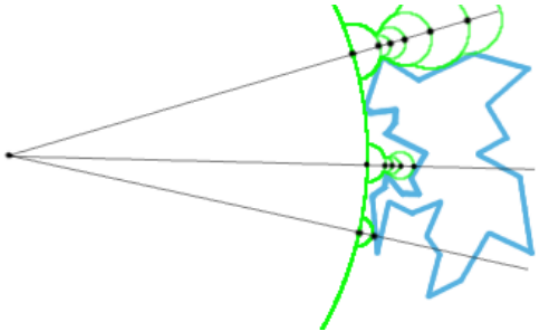


Figure 1: Sphere tracing. Green circles represent unbounding spheres and black dots their centers[Tom12]

The step length of each iteration will also be determined using the signed distance function. We can use unbounding volumes to visualize this procedure. Unbounding volumes are defined as volumes that do not contain any part of the objects [J C89]. A sphere, which radius is the value returned by sampling the signed distance function from the current point of the ray will not contain any surface, therefore we can safely step the distance along the ray, repeating the sampling process.

$t \leftarrow 0$
$d \leftarrow 0$
while $k < maxK$ and $t < maxDist$ **do**
    $d \leftarrow sdf(r(k))$
    if $d < \epsilon$ **then return** t ▷ Intersection happens
    end if
    $t \leftarrow t + d$
end while
return 0 ▷ No intersection happens after $maxK$ steps

Compared to using a fixed step size, sphere tracing lets us step to the surface much quicker, thus reducing the computational power required to find the intersection point, as well make it very easy to know in which cases the ray will not hit any surface whatsoever.

2. Sphere Tracing-based Raymarching in Unity

Unity's rendering pipeline is a standard real-time rendering pipeline, fit for rasterization. However, by using raymarching, we skip many of the conceptual stages of a rasterization-based renderer, like geometry processing, model and view transformations, etc.

Instead we can choose to take a simpler approach and simply generate a texture of a raymarched implicit surface and simply display that. To populate the texture, using the GPU is still preferred, since raymarching is easily parallelized, but since we want to avoid using the rasterization-based graphics pipeline of most GPUs, we can instead write our raymarching program in a compute shader. Compute shaders are programs that run on the graphics card, outside of the normal rendering pipeline. [19a]

2.1. Texture Preparation

The created texture must fit the rendered window or screen, therefore it much be created based on the properties of Unity's `Camera` object. This object contains the pixel width and height size required to create our texture, as well as transformation matrices, required to generate rays for every pixel of the texture.

To create the texture we will be displaying, a `RenderTexture` class object must be used. To allow the GPU access to read from or write to a texture, it must be wrapped in a resource view. The default case of a Shader Resource View is not sufficient in this case, since it does not allow for the resource to be written to. To allow the Compute shader to write data into the created texture we must wrap it in an Unordered Access View, which is more costly in terms of performance, but allows simultaneous read and write. To set the `RenderTexture` object to be used in an Unordered Access View we simply enable the `RenderTexture.enableRandomWrite` flag.

2.2. Raymarching Data

The most effective way of making raymarching more user-friendly is to use Unity's in-built GameObject and Scene structure to generate all the data we need to start our raymarching algorithm. This way we will avoid hard-coding data into our raymarching code, allowing people with no graphics programming knowledge to use the renderer. GameObjects are the fundamental objects in Unity that usually represent characters, props and scenery. They do not accomplish much in themselves but they act as containers for Components, which implement the functionality [19b].

We can use GameObjects with in-built Camera and Light components to generate the viewport and lighting data we will need render the surfaces. For the surfaces themselves however, we define our own component, purely for the purpose of data storage, to contain the position, scale, and surface type of objects. The position and scale will be used by the signed distance functions to compute the distance form a sampling point to the surface, while the surface type will specify which signed distance function to use with this function.

2.3. Rendering Distance Fields

After collecting all the required data from the Scene and moving it to the Compute shader, we employ the sphere tracing technique to first generate the distance fields of specific signed distance function surfaces. In the distance field, the value of each pixel will be the result of the sphere tracing algorithm for the ray corresponding to that pixel, normalized to the range of [0; 1]. This allows us to see the general shape of the surfaces being rendered.

Since the surfaces are connected to the GameObjects in the scene, they can easily be moved, scaled, and their signed distance function changed. The results of these changes are instantly visible, since the ComputeShader is always running and being provided with the updated data from the Scene.

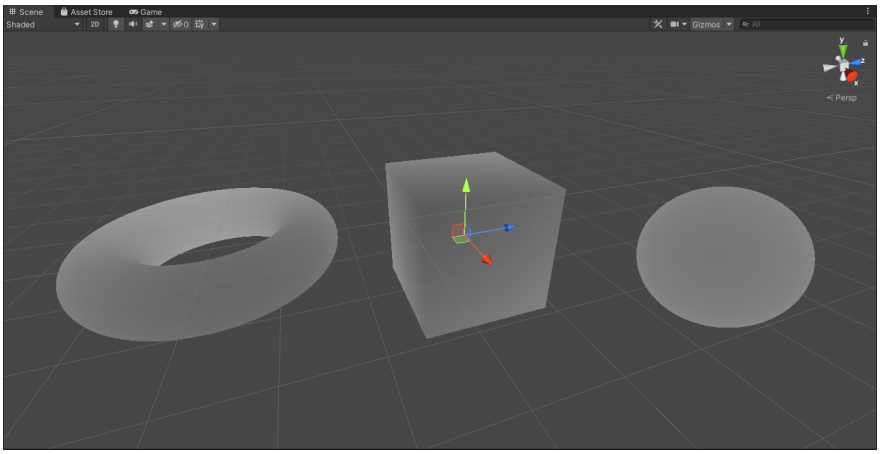


Figure 2: Distance field of a torus, box and a sphere being rendered in the Unity scene view. Position handles are visible that can be used to move the object in real time.

2.4. Phong Illumination Model

To convert the distance fields to a more realistic image of the surfaces we will be using the Phong illumination model, based on [Pho75]. To use the Phong Illumination Model, we first have to calculate the surface normals at every surface point of our image. The surface normal at a surface point is equivalent to the gradient for the surface distance function at that point, since the distance increases most rapidly when moving directly away from the surface.

To compute the gradient we will compute the rate of change of the function along the x-axis, then along the y-axis and finally along the z-axis[Pru17]. This can be done by calculating the difference quotient:

$$\vec{n} = \begin{bmatrix} f(x+\delta, y, z) - f(x-\delta, y, z) \\ f(x, y+\delta, z) - f(x, y-\delta, z) \\ f(x, y, z+\delta) - f(x, y, z-\delta) \end{bmatrix} \tag{2}$$

As stated in [Pho75], the illumination at each surface point will be composed of ambient reflection, diffuse reflection and specular reflection. Per-scene properties, like ambient color, positions

of lights, color of light are retrieved from existing Unity properties and reused in the Compute shader for raymarching. Multiple lights work by calculating the Phong contribution of each light per pixel and then adding the different lights together.

2.4.1. Shadows

To calculate shadows we simply create new raymarching rays with their origin being just above the surface of an object (calculated by moving the intersection point along the surface normal), and direction being towards each of the lights. If another surface is hit with this ray, that surface point is not affected by the respective light.

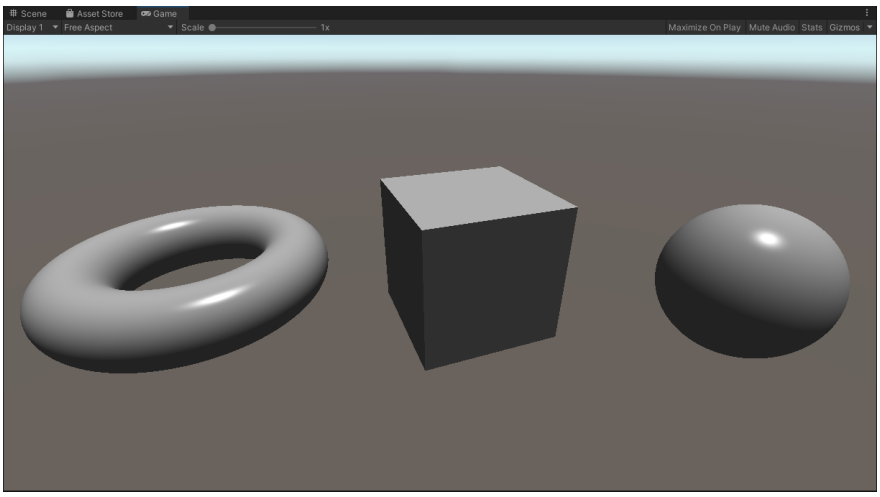


Figure 3: Surfaces rendered by a raymarcher with Phong illumination model and a single directional light in Unity.

2.5. Unique Surface Materials

Up until now the entire scene was rendered using a single Phong illumination-based material. However, it is highly required to have multiple materials for different objects to increase the potential complexity of a scene. To achieve multiple materials we first define additional properties for our Surface component. We add the per-object properties of the Phong illumination model to our Surface component: the diffuse color and the smoothness factor.

This additional data is used alongside the position and surface type data in the Compute shader. Instead of simply keeping track of the distance to the surface in our sphere tracing algorithm we can also check which object was closest to the simple point and keep track of that objects properties. When the ray-surface intersection point is finally found, we use that specific object's properties to run our illumination model, resulting in surfaces of different colors and smoothnessess.

These additional properties, like before, are exposed to the user in the Unity Inspector of our custom-made Surface component, allowing them to freely change the color and/or smoothness of an object and see the results in real-time, greatly increasing usability.

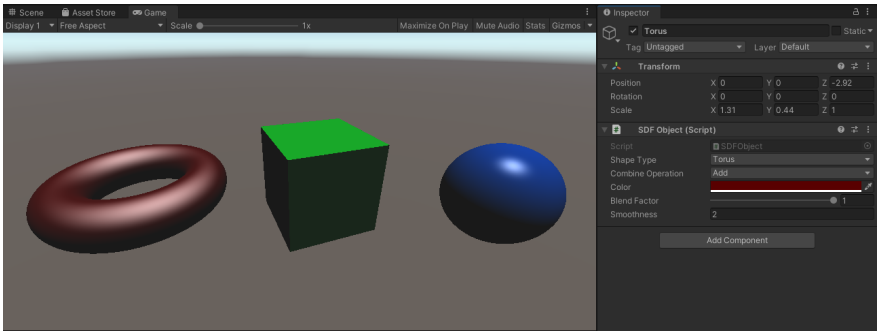


Figure 4: Surfaces using unique materials. Also visible is the Inspector for the Surface component, used to easily set various properties of the surface.

2.6. Constructive Solid Geometry

To model more complex shapes and surfaces we use the Constructive Solid Geometry procedural modeling method. In CSG, simple primitives are combined by means of regularized Boolean set operators that are included directly in the representation [FvDF+96]. We are able to define intersection, union and difference of the signed distance function by combining the through very simple and cheaply computable functions. Simply finding the minimum or a maximum of two distance is enough to fully implement Constructive Solid Geometry. The boolean operators can be expressed as such:

$$intersect(d_1, d_2) = max(d_1, d_2)$$

$$union(d_1, d_2) = min(d_1, d_2)$$

$$difference(d_1, d_2) = max(d_1, -d_2)$$

3. Raymarcher Improvements

3.1. Smooth Union

The main disadvantage of Constructive Solid Geometry is that the edges between two combined surfaces are very sharp, which can cause difficulties when trying to recreate more organic shapes and trying to unify smooth objects. We can use a smooth minimum function, provided in [Qui13b] to improve our union operator and allow for smooth connections between multiple surfaces.

The form of a smooth minimum function is

$$smin(a,b,k) = \begin{cases} a, & \text{if } a-b \geq k \\ b, & \text{if } a-b \leq k \\ f(a,b,k), & \text{if } a-b \in (-k,k) \end{cases}$$

where $f(a,b,k)$ is a smooth interpolator and k defines the interpolation range. The polynomial smooth interpolator derived in [Qui13b] takes the form of

$$f(a,b,k) = a(1-h) + hb - kh(1-h)$$

We can use the newly defined smooth minimum function to expand our Constructive Solid Geometry boolean operation set with smooth operations such as:

$$smoothUnion(d_1, d_2) = smin(d_1, d_2, k)$$

Focusing on the ease of use aspect of this project, we expose the choice of CSG operation to the Surface component so the boolean operations can be edited in real time, which allows for a real-time instant feedback modelling experience. The smooth interpolation range property k is also exposed, as seen in Figure 4.

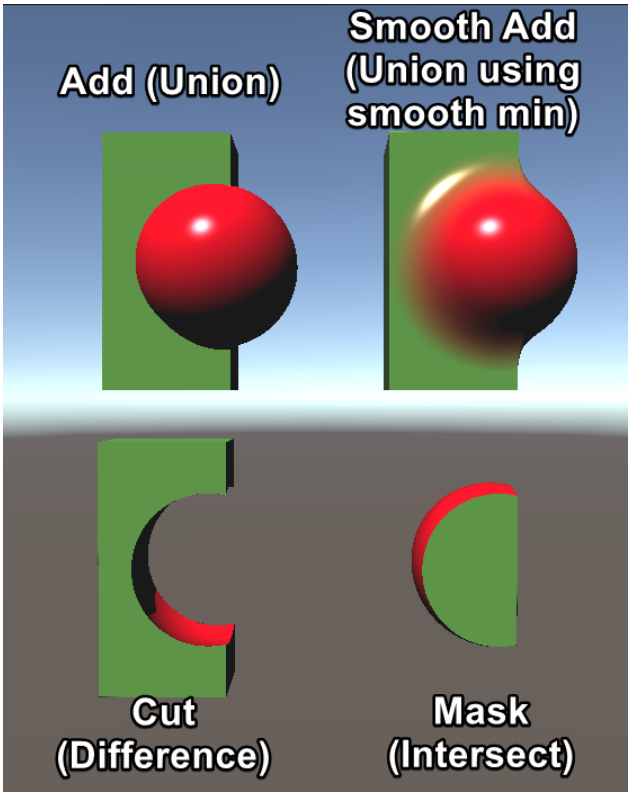


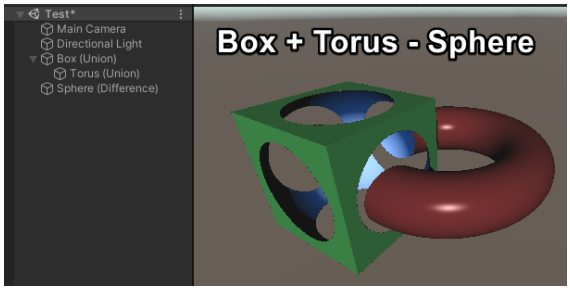
Figure 5: The different CSG combination operators between and SDF of a box and a sphere, as well as an example of a smooth union operator using $k = 1$.

3.2. Child-Parent system for surfaces

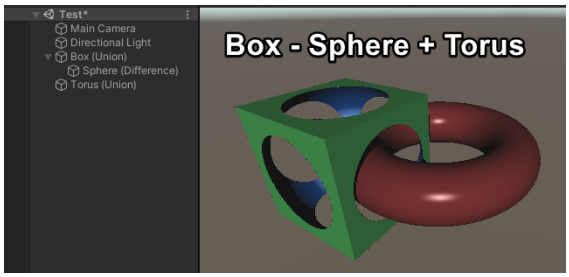
An issue that quickly arises when trying to create more complex shapes using multiple primitives and different CSG operators is that the order of the CSG operations is important. The boolean operations used in CSG are not associative, and the order of execution is based on the order of surface objects in the scene, which is neither intuitive nor easily changeable. The core of the issue is that the whole scene is being managed as a simple object, thereofe it is hard to perform localized operations to a single object, and not the whole scene.

To alleviate this issue we use the fact that GameObjects can be parented to one another in Unity. We use this to define relations between objects that make CSG operation execution order more intuitive. For any parent object, it has all the children objects merged with it first (using their selected combination operations), before merging with other parent objects.

Since the Compute shader running the raymarching algorithm has no access to the scene data, we have to setup our data in a way where it is possible to distinguish parent objects and their children objects. This is done by reordering the surface object buffer that is being sent to the Compute buffer in a way where the child objects follow straight after the parent objects, as well as adding a new property to the Surface component providing the amount of children this surface object has. With this information the Compute shader can first merge the child surface objects, before merging the parent objects.



(a) The union between the box and the torus is merged first.



(b) The difference between the box and the sphere is merged first.

Figure 6: An example of two different results achieved using the same surface objects by reordering the CSG sequence using the Parent-Child system.

3.3. Distance Field Transformations

Once the primitives are rendered, we can apply various euclidean and non-euclidean transformations to the distance field, which will change the shape of our surface. Using these various effects we can create even more complex shapes, while retaining all of the strength of raymarching and not losing too much performance.

3.3.1. Primitive Alterations

3.3.1.1. Elongation

We can use elongation to construct diferent shapes from the primitives we have. By elongating a signed distance field, we split the surface in two, move them apart and then connect them. This technique can be used to f.e. create a capsule shape, by elongating a sphere. We transform the sampling point $\overrightarrow{p}$ by doing the following:

$$\overrightarrow{q} = \overrightarrow{p} - clamp(\overrightarrow{p}, -\overrightarrow{h}, \overrightarrow{h})$$

where $\overrightarrow{h}$ is the vector by which we separate the split primitives. Then $\overrightarrow{q}$ is used to sample the signed distance function.

3.3.1.2. Rounding

Rounding is a very simple transformation, we simply subtract some value from the result of the raymarched distance:

$$rounded(\overrightarrow{p}) = sdf(\overrightarrow{p}) - r$$

3.3.1.3. Onion

To hollow out a shape without performing more expensive boolean operations we can simply take the absolute value of the signed distance field result to imitate a hollow shape and the subtract

some value to give it thickness:

$$onion(\overrightarrow{p}) = |sdf(\overrightarrow{p})| - r \tag{3}$$

Figure 7: Examples of primitive alterations: elongated torus, rounded box, hollow box.

3.3.2. Distortions

Deformations and distortions allow to enhance the shape of primitives or even fuse different primitives together. The operations usually distort the distance field and make it non euclidean anymore, so one must be careful when raymarching them. [Qui13a] To avoid artifacts when using deformations or distortions we must reduce the step size of our raymarching function. Since in sphere tracing the step size is not constant, we simply reduce it by some factor, depending on the strength of the deformation effect.

3.3.2.1. Displacement

To add displacement to the calculated surface we can choose some displacement pattern function, f.e.

$$\overrightarrow{p} = (x, y, z)$$

$$d = sin(x) * sin(y) * sin(z)$$

and simply add the displacement pattern to the calculated distance:

$$displaced(\overrightarrow{p}) = sdf(\overrightarrow{p}) + d$$

3.3.2.2. Twist

We can create a twist effect for a surface by rotating the sampling point on the Y axis based on the Y position of the point.

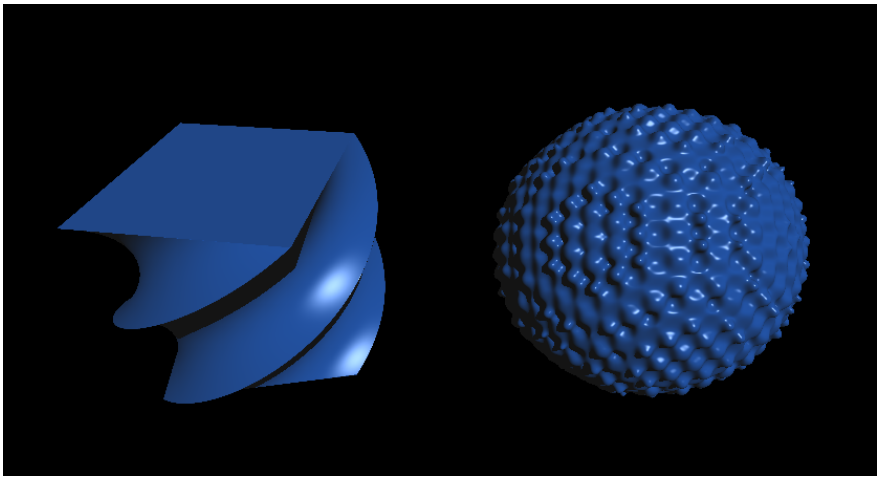


Figure 8: Examples of distortions: twisted box, sphere with a displacement pattern.

3.3.3. Infinite Repetition

By using the modulo operation we can repeat a bounding box of the distance field infinetely many times, which in turn allows us to render infinitely many primitives without increasing the memory usage of our application and still only evaluating a single signed distance function[Qui13a]. The modulo operation can be applied with a custom repetition period like so:

$$\overrightarrow{q} = \overrightarrow{p} * \frac{\overrightarrow{c}}{2} mod \overrightarrow{c} - \frac{\overrightarrow{c}}{2} \tag{4}$$

$\overrightarrow{c}$ is the repetition period which can be different in each coordinate direction.

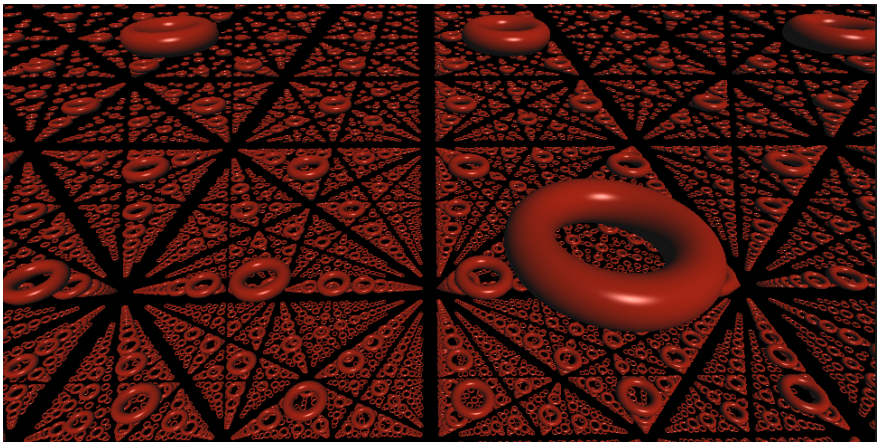


Figure 9: ”Infinitely” repeated torus by using a modulo operation on the distance field.

3.3.4. Transformations Interface

The type of transformation used by the surface, as well as parameters affecting that transformation are also exposed in the Surface Component. Transformations can be easily edited, with real-time results, allowing for easy experimentation.

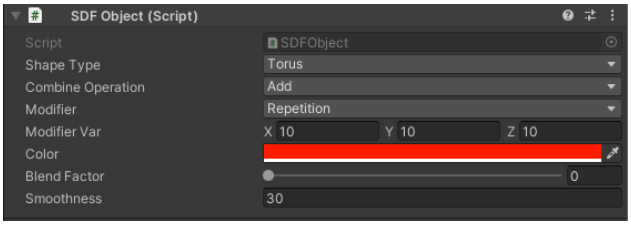


Figure 10: The Surface MonoBehavior component.

Conclusions

The focus of the project was to implement a raymarching renderer as a replacement of the default rasterized in Unity, explore whether the concept works within the Unity ecosystem, as well as use this as a chance to create a more approachable raymarching renderer, that can be easily used by artists and other users with little to no graphics programming knowledge. Another goal was to explore more in-depth raymarching techinques, while still providing good usability.

The renderer was successfully implemented in the Unity game engine. Low-level graphics APIs of Unity worked well in this case. Also a lot of the scene setup and rendering concepts from the default rendering engine we're reused even with a wildly different rendering technique: parent-child hierarchy, light and camera GameObjects.

The advantages of using Unity as a container for a raymarching renderer are clear. Well developed scene editing system allows to edit, move, scale the surfaces in real-time and see the results instantly. Custom components we're also implemented with the goal of being heavily customizable and editable, therefore all features share the advantage of being easy to use and experiment with.

While the implementation is somewhat bare-bones, the potential is obvious, and with more development efforts raymarching could become a choice as the main renderer used in an Unity project of some capacity, and if not, the availabilty of geometry data in every rendering step can be a useful addition in a more widely used real-time graphics pipeline, namely for computing shadows or reflections. The possibility of rendering any surface with a defined distance function also allows for the rendering of surfaces that could be very hard to polygonize, especially with the addition of using various distance field transformations.

Bibliography

[19a] *Unity Manual - Compute shaders*. 2019.

[19b] *Unity Manual - GameObjects*. 2019.

[FvDF+96] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley Professional, Boston, USA, 1996. section 12.7 Constructive Solid Geometry, p. 557.

[J C89] L. H. Kauffman J. C. Hart D. J. Sandin. Ray tracing deterministic 3-d fractals. *Computer Graphics*, 23, 1989.

[Pho75] B. T. Phong. Illumination for computer generated pictures. *Graphics and Image Processing*, 1975.

[Pru17] J. Prunier. Rendering implicit surfaces and distance fields: sphere tracing. `https://www.scratchapixel.com/lessons/advanced-rendering/rendering-distance-fields/basic-sphere-tracer`, 2017.

[Qui13a] I. Quilez. Distance functions. `https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm`, 2013.

[Qui13b] I. Quilez. Smooth minimum. `https://www.iquilezles.org/www/articles/smin/smin.htm`, 2013.

[Tom12] L. J. Tomczak. *GPU Ray Marching of Distance Fields*. MA thesis, Technical University of Denmark, 2012.

[Won16] J. Wong. Ray marching and signed distance functions. `http://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/`, 2016.