ANKARA UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

BLM 4061 PROJECT PROPOSAL

Rendering Singed Distance Fields Using Raymarching

Alp Ertunga Elgün

19290238

Doc. Dr. Gazi Erkan Bostancı

October 2022

ABSTRACT

Raymarching SDFs (Signed Distance Fields, or Functions sometimes) is slowly getting popular, because despite its elegance and simplicity, it is a powerful way to render 3D models, both procedural and not. Graphics processing units are evolving their computational capabilities faster than their memory bandwidth which means purely mathematical Signed Distance Functions started to be competitive against regular polygonal meshes.

This project started as a hobby project to just learn more about the Signed Distance Fields but slowly changed its focus to finding an ansver to the question “How can we modify the distance fields without recompiling all of the shader ?”

You will read and learn about SDFs, its drawbacks and strengths and how did we find an answer to that interesting question.

You can find the source code of the project in its github repository: https://github.com/Sayuris1/solarpunk_raymarcher

You need Rust to compile the source code. We suggest that you follow the instructions on this website to download and install Rust: https://www.rust-lang.org/tools/install

And you need a GPU that supports OpenGL Core 4.1 to run the renderer.

TABLE OF CONTENTS

1. LITERATURE REVIEW

1.1 What are SDF and Raymarching

1.1.1 What is a Signed Distance Function

Signed Distance Functions explicitly define what is the inside and outside of an object. Consider the function $F: \mathrm{R3} \rightarrow \mathrm{R}$ this function takes a 3D point (x, y, z) in input and returns a single value [1]. This value tells us explicitly if we are outside or inside the volume: Positive distances indicate the point is outside the geometry, a 0 distance indicates the point is on the geometry surface, and a negative distance indicates that the point is inside the geometry. By combining those distances for different geometries we get a field on which the rays can march. We call that field a "Signed Distance Field". [2]

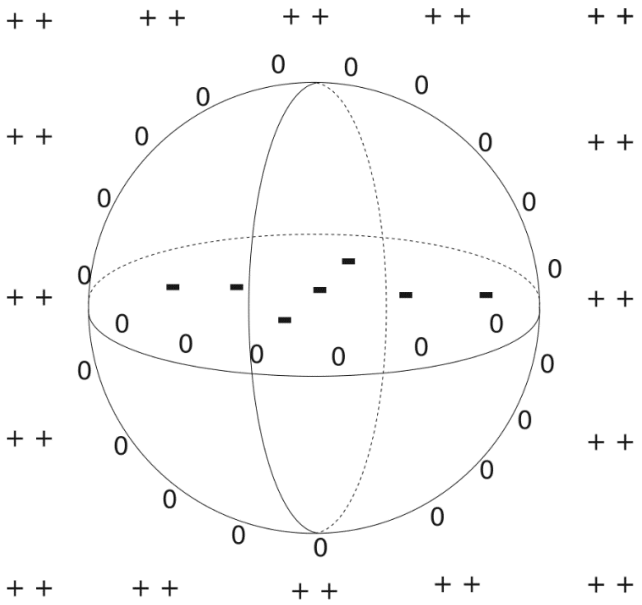

Image 1.1: A distance field [3]

With SDFs, we can store models using smaller memory, draw volumetric representations of real objects easily and use interpolation based anti-aliasing without the need for multi-sampling. It is also easy to create new shapes at runtime and there is no resolution associated with the functions. [4]

1.1.2 What is Raymarching

Raymarching (also known as sphere tracing) is a technique used to render geometry that is defined by signed distance functions (SDFs) [5-6]. Unlike other rasterization based renderers, a raymarching renderer does not need vertices and normals, it uses SDFs which calculate the distance from a point in space to the geometry.

At a high level, the renderer will shoot out a bunch of imaginary rays from a virtual camera that looks at the world. Each of these rays will "march" along the direction of the ray, and at each step, evaluate the SDF. This will return the distance to the closest point on the scene geometry. Then if it takes this distance as the radius of a sphere centered around the current position of the ray, it can safely move forward along our ray by that amount without overshooting or missing any objects. [7]

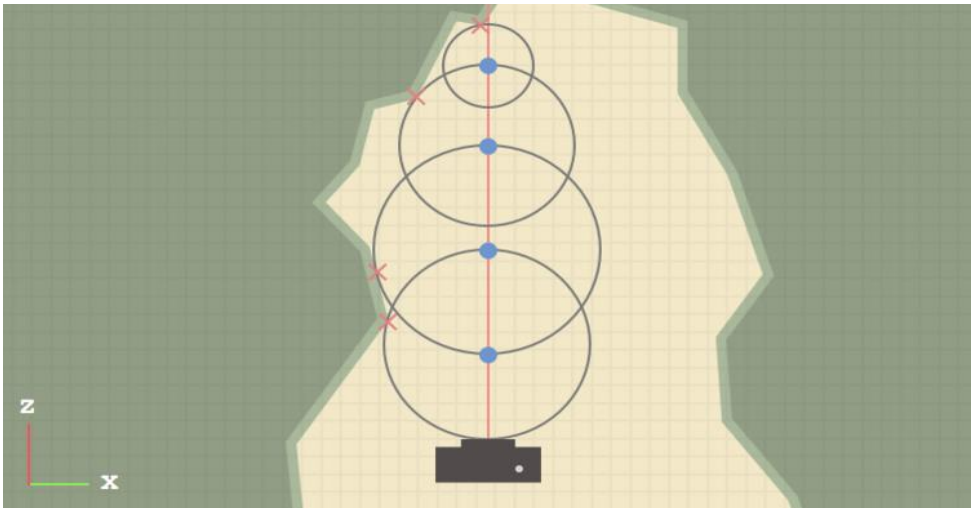

Image 1.2: Raymarching [2].

Raymarching can "emulate" ray tracing. It is possible to achieve the same level of photorealism as ray tracing which has become the standard for high-fidelity offline rendering. And it is possible to do this faster than ray tracing [8].

Current Graphics Processing Units (GPUs) are made specifically for rasterization based renderers, which makes raymarching renderers hard to integrate and raymarching pipelines immature. But GPUs are evolving their ALU capabilities faster than their memory bandwidth. This means purely mathematical SDFs started to be competitive against polygonal renderers [4].

1.2 Examples

1.2.1 Shadertoy and Demoscene

There are lots of examples of this kind of rending in the https://www.shadertoy.com/ website made by demoscene developers and hobbyist graphics programmers. My favorite one is “Selfie Girl” made by I. Inigo Quilez. You can see it here:

Image 1.3: https://www.shadertoy.com/view/WsSBzh

1.2.2 Pre-Compiled WebGL Based Raymarching Renderers

There are some other WebGL based renderers. I’ll list them below.

Adrian Baker: https://github.com/radian628/raymarching-engine
Varun Un: https://github.com/varun-un/Raymarcher
Daniel Estaban: https://github.com/danielesteban/three-raymarcher

All of those renderers work in a pre-compiled scene. If we want to add another geometry to the scene dynamically, we need to re-compile the shader code. This makes those renderers not suitable to use in an actual production environment.

1.2.3 Voxel Volume-Based Renderers

Sebastian Aaltonen, in his game Claybook, stores signed distance fields in a big volume texture (3D texture) with MIP maps, then divides this big texture into little predefined voxels. When scene geometry changes, he only calculates the new distance values of the affected voxel to save lots of resources. Then he raymarches that volume texture to draw the scene. [9]

Another game that uses volume texture to store SDF is Dreams. It supports user generated scenes in a dream like world. This renderer tries to make the voxels as big as possible to save resources. It starts with a big 4x4x4 voxel scene. Then iteratively refining the voxels by splitting them into little 4x4x4 voxels if that voxel has any geometry. Then instead of using regular raymarcher, they use point-clusters to achieve more dreamlike scenes instead of photorealistic ones that a normal raymarcher produces. [10]

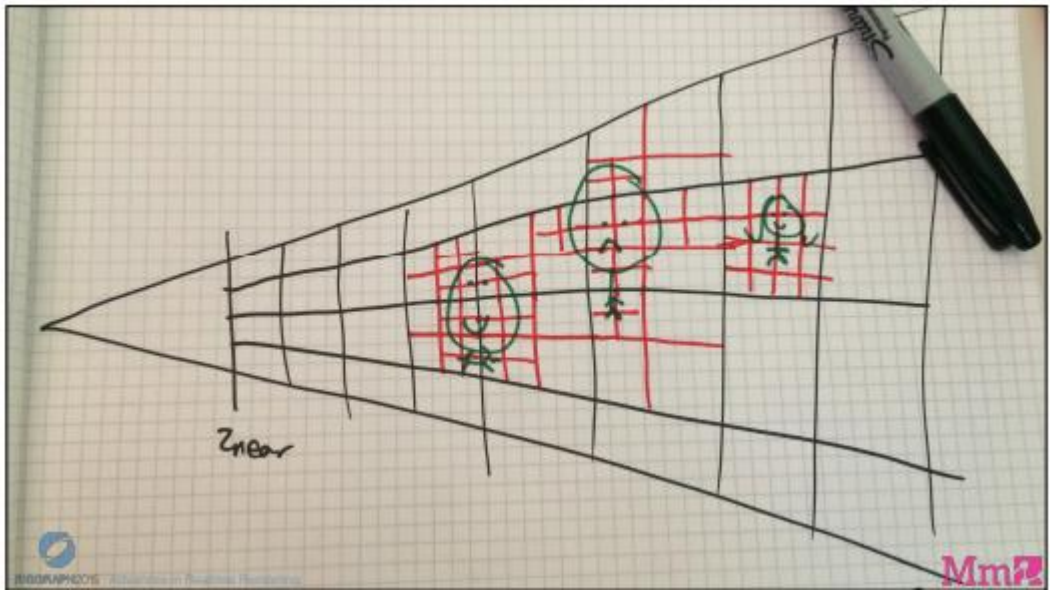

Image 1.4: Dreams renderer refining the voxels. [10]

Even though those renderers make it easy to dynamically add geometry to the scene, they introduce new resolution and memory problems. And with the new GPUs, calculating the signed distance field instead of storing it in a big volume texture could be faster.

1.2.4 Other Examples

Sieben Corgile tried to calculate the signed distance field from a stack of distance functions so he can add new geometry dynamically but failed to make the stack machine perform well in the GPU. Then he tried to inject new geometry directly into SPIR-V byte-code. This technique works well but only one injection point per shader is currently expected. We need to be able to define multiple functions to generate beautiful scenes [4].

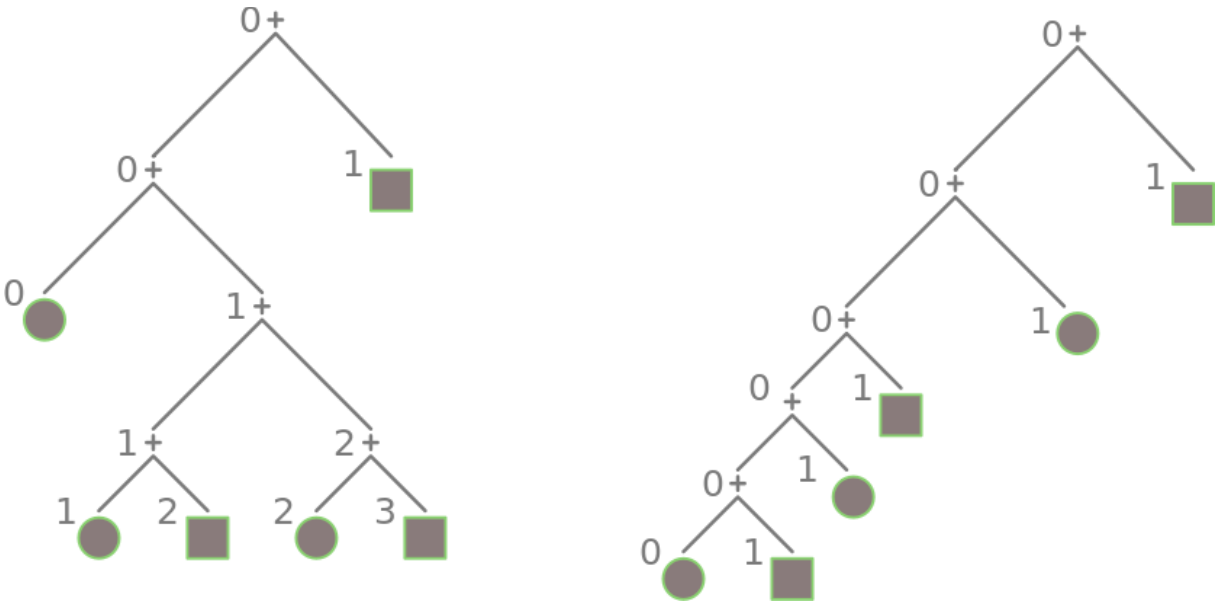

Image 1.5: Stack machine representation [4].

There is a CUDA based raymarcher that is made by Avelina and Ognjen. They use an explicit Frame Buffer Object (FBO) to render the results of the CUDA code. This way they avoid the need for using the rasterization hardware of the GPU to display the texture. Unfortunately, they don't talk a lot about the SDF creation process that we are interested in. And they don't have the source code or an executable available on the internet. But the raymarching technique they used in their renderer is interesting [8].

Lastly, we have Inigo Quilez who can be credited as the person who introduced raymarching to the computer graphics community. His website contains dozens of articles on raymarching which are considered the de facto reference for the construction of raymarching shaders [17].

2. DISCUSSION

2.1 Why SDFs

As you can see from shadertoy demoscenes. It's possible to make mesmerizing scenes that morph and change using SDFs and math. And from the Dreams example, you can see that it's easier for inexperienced users to create beautiful sculptures by adding and carving. Which is easy to do with SDFs, unlike polygon based geometry.

And with the raytracing algorithms, we can render those geometries in a photorealistic, anti-aliased, globally illuminated way. We can do everything a ray tracer does but faster. [8]

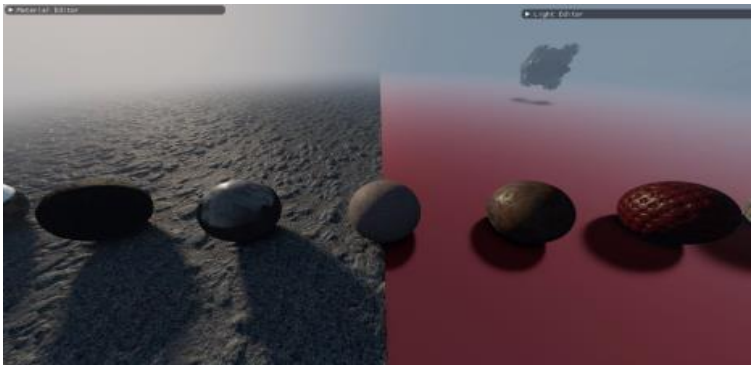

Image 2.1: A globally illuminated raymarched scene. [8]

2.2 What Can We Make

What I want to do is make a renderer that can dynamically add signed distance functions, combine them and raymarch the resulting signed distance field. This way the user can sculpt the shape and immediately see the wanted material applied to it. Make it metallic, claylike or even transparent.

We can calculate the signed distance field in a separate render target then march and draw the resulting image to the screen. OpenGL is one of the well known rendering backhands and it can handle different render targets, 3D textures and shader programs.

Because of the complexity of the project, We had decided to divide the project into two halves. We focused on modeling and sculpting this semester. We will focus on photorealistic rendering in the next semester.

2.3 Choices

2.3.1 Libraries

For window management and input handling, SDL2 is one of the best and most widely used tools. And for the main coding language Rust was a good choice. It has wrappers for both OpenGL and SDL2. And the way it abstracts away inappropriate manual memory management helped in our use case.

And lastly, to draw the raymarch results we can use an FBO or a full screen triangle. In some devices, FBO is faster. But in some other devices, a full screen triangle is faster. [11]

2.3.2 Using a 3D Texture

As you can see from the examples above. We could either store the signed distance field in a 3D texture (volume texture) and then update it when necessary. Then push it to the GPU to raymarch or we can calculate the signed distance field in the GPU each frame.

Storing the distance field in a texture makes the field resolution dependent. Bigger resolutions use more VRAM but would let the artists draw more detailed models.

Not storing it will mean the renderer need to calculate all the distance field again each frame. More details will require more calculations.

We had decided to use 3D textures because one of the main goals of this project was to let the users sculpt any shape they wanted. If we didn't use them users would be limited on how many details they can add, this would limit their creativity.

3. IMPLEMENTATION

3.1 Using Rust

All of the code written in Rust has memory safety guarantees enforced at compile time. However, computer hardware is inherently unsafe. So Rust has a second language hidden inside it that doesn't enforce these memory safety guarantees: it's called unsafe Rust and works just like regular Rust, but lets us call an unsafe function or method. [12]

Using safe code makes sure that there won't be any segmentation faults. So we need to wrap carefully written unsafe code in modules before using them.

3.2 Initializing SDL and OpenGL

Simple DirectMedia Layer (SDL) is a cross-platform development library designed to provide low level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL and Direct3D [13]. In this project, we used SDL to get inputs and issue draw calls to GPU.

SDL2 doesn't support Rust natively but Rust provides Foreign Function Interface (FFI) to C libraries. Low level C libraries are unsafe. But most of them have safe to use wrappers that we can use without any worries. Rust-SDL2 create is one of those wrappers. It uses an MIT license and is easy to use. We used it in our project to call SDL2 functions safely.

After initializing an SDL window and telling SDL to create OpenGL context. We need to load OpenGL functions. This proces is diffirent for each GPU driver. But GL-SD Rust create abstracts away all those different processes. We used that create to load OpenGL function pointers.

3.3 Shader and Shader Program Wrappers

In modern OpenGL, shaders are little programs that rest on the GPU. These programs are run for each specific section of the graphics pipeline. In a basic sense, shaders are nothing more than programs transforming inputs into outputs. Shaders are also very isolated programs in that they're not allowed to communicate with each other; the only communication they have is via their inputs and outputs. [14]

Shader compilation, error handling and linking are all unsafe low level C function calls. So we carefully wrote those unsafe parts and wrap them in a struct before using them in our main program.

3.3.1 Resource Cleanup

It's important to clean any leftover variables and memory allocations that we created in unsafe code before dropping the wrapper structs. Rust makes sure there are no memory leaks in its safe code but the programmer needs to take full responsibility for unsafe codes. [12]

In the early stages of the development, before we wrapped the shader implementation in a struct, there were lots of memory leaks and segmentation faults. Because they are runtime bugs, they are hard to find and fix. The way Rust forces the programmer to write safe code prevents all of this [12]. And if using unsafe code is a must like in our case, those unsafe code needs to be very carefully wrapped.

Rust calls a "Drop" function before a variable goes out of the scope. We freed all of the unused but allocated memory here. And deleted every variable no more needed.

3.3.2 Error Handling

We need to define our SDFs in the shader and raymarch those SDFs in the shader. So we need to see a proper error message if the shader fails to compile.

If there is any error in the shader, instead of returning the error as a string OpenGL writes that returned error to a buffer. So we need to get the length of the error message and allocate memory of the correct size.

We used the 'Vec' variable type to create the buffer. A contiguous growable array type, short for 'vector'. Filled it with spaces and then gave it to OpenGL to write the error. But C 'string' and Rust 'string' are completely different. We used 'CString' create to convert this 'Vec' to C 'string' using the same allocation. Then give it to OpenGL.

There is a special enum called 'Result' in Rust. This type is used for returning and propagating errors. It is an enum with the variants, $Ok(T)$, representing success and containing a value, and $Err(E)$, resenting error and containing an error value. [12]

The function $unwrap(self) \rightarrow T$ will give you the embedded T if there is one. If instead there is not a T but an E or $None$ then it will 'panic'. Rust handles critical errors in code with the 'panic' macro. When this macro executes, the whole program unwinds, that is, the program clears the stack and then quits. Due to the way a program quits after panicking, 'panic' is commonly used for unrecoverable errors. [12]

3.4 3D Textures and Frame Buffer Objects

3.4.1 3D Textures

3D textures or 'Volume textures' are a series of normal textures arranged as slices, they can be read using point or tri-linear sampling. They can be sampled using 3 coordinates, UVW. [15]

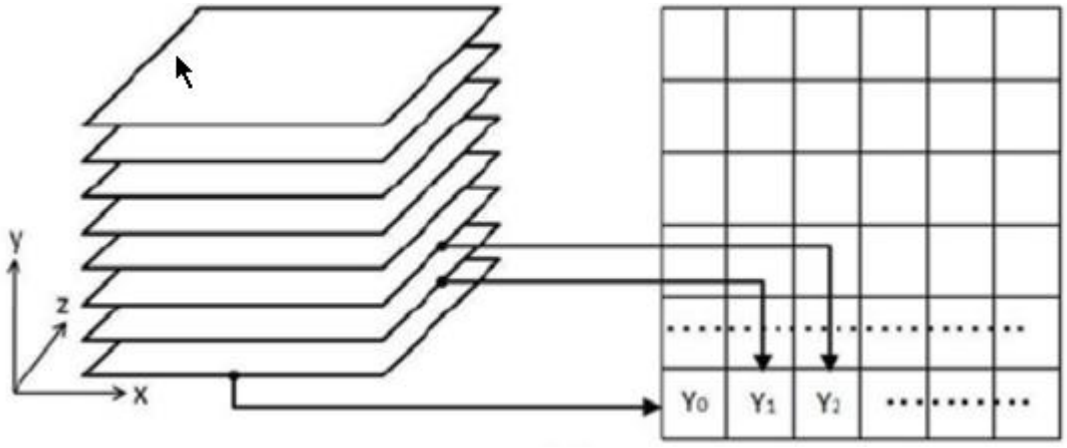

Image 3.1: 3D textures are just big 2D textures. [15]

3.4.2 Frame Buffer Objects

A Framebuffer is a collection of buffers that can be used as the destination for rendering. The Default Framebuffer represents what you see on some part of your screen. Framebuffer Objects (FBOs) are OpenGL Objects, which allow for the creation of user-defined Framebuffers. With them, one can render to non-Default Framebuffer locations and thus render without disturbing the main screen. [15]

We need to store the distance field in a 3D texture. To create and modify the distance field we need to be able to render to a 3D texture without disturbing the main screen. We are attaching a 3D texture as the main color buffer of an FBO and set this FBO as the render target. After we modified the distance field and rendered the new distance target to the 3D texture, we set the default FBO as the render target and raymarch the modified distance field by sampling from the 3D texture.

3.5 Initializing Distance Field

3.5.1 Full-Screen Triangle

To render the distance field to the 3D texture, we needed to run a full screen pixel shader. To render a full screen pixel shader pass, we needed to fill the screen with geometry. There are two ways to achieve this: Rendering with 2 triangles that fit the viewport perfectly or 1 massive triangle that spills over the sides of the viewport.

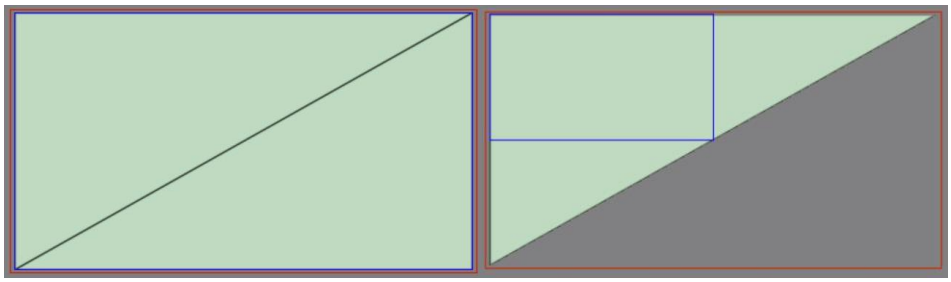

Image 3.2: A full screen quad on left. A big triangle in right. [11]

Chris Wallis made some profiling to test which approach is faster. He found out that using 1 triangle is faster. He found out the reason for that. In fixed function rasterization, things always render as batches of 2x2 pixel quads. Regardless of whether your triangle is only the size of a single pixel, the GPU will render a full 2x2 quad even if 3 of those pixels don't exist (the other 3 pixels are called "helper pixels"). So the diagonal edge between the triangles causes overlapping quads that result in the pixel shader getting called twice for all pixels along this edge. [11]

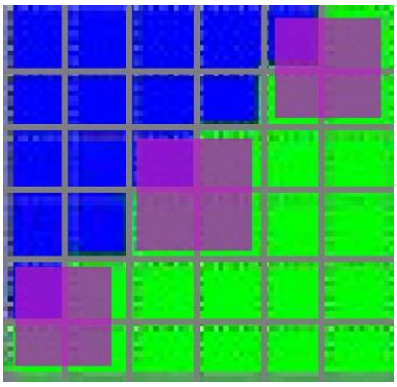

Image 3.3: All of these purple quads are essentially getting run twice. [11]

3.5.2 Texture Filtering

Texture filtering or texture smoothing is the method used to determine the texture color for a texture mapped pixel, using the colors of nearby texels (pixels of the texture) [15]. We used 2 different filtering methods in our project.

Nearest-neighbor interpolation simply uses the color of the texel closest to the pixel center for the pixel color. We used nearest filtering to write distance field to the 3D texture without losing any precision on the distance data.

Linear interpolation is performed using interpolation first in one direction, and then again in the other direction. Although each step is linear in the sampled values and in the position, the interpolation as a whole is not linear but rather quadratic in the sample location [15]. 3D textures let us interpolate between three different directions instead of regular two directions. This helped us to render smooth images at the end of the raymarching.

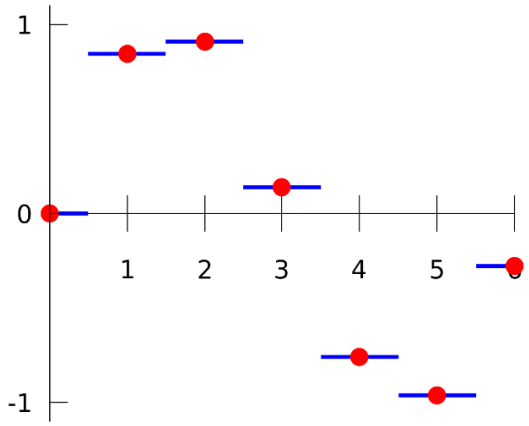

Image 3.4: Nearest interpolation. Blue lines are possible interpolation values. Red points are the real dataset. [16]

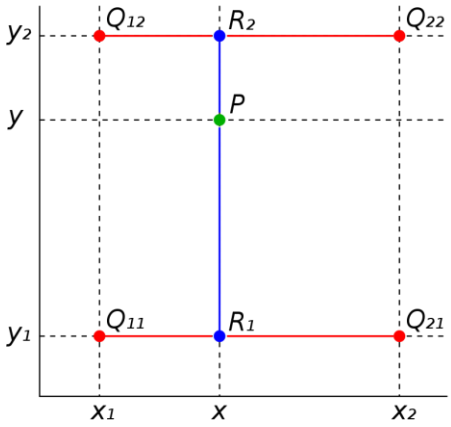

Image 3.5: Linear interpolation. The four red dots show the data points and the green dot is the point at which we want to interpolate. [16]

3.5.3 Signed Distance Function of a Sphere

To fill the distance field, we needed to find the distance between the current point and the closest surface of the geometry. Our scene contains only one big sphere at the start. So we needed to find the signed distance function of the sphere.

A signed distance function $F: \mathrm{R3} \rightarrow \mathrm{R}$ takes a 3D point (x, y, z) input and returns a single value, the distance. If we name the 3D input point as p, the 3D center point of the sphere as c and the radius of the sphere as r. The signed distance function of a sphere would be $F = \big||p - c|\big| - r$. [7]

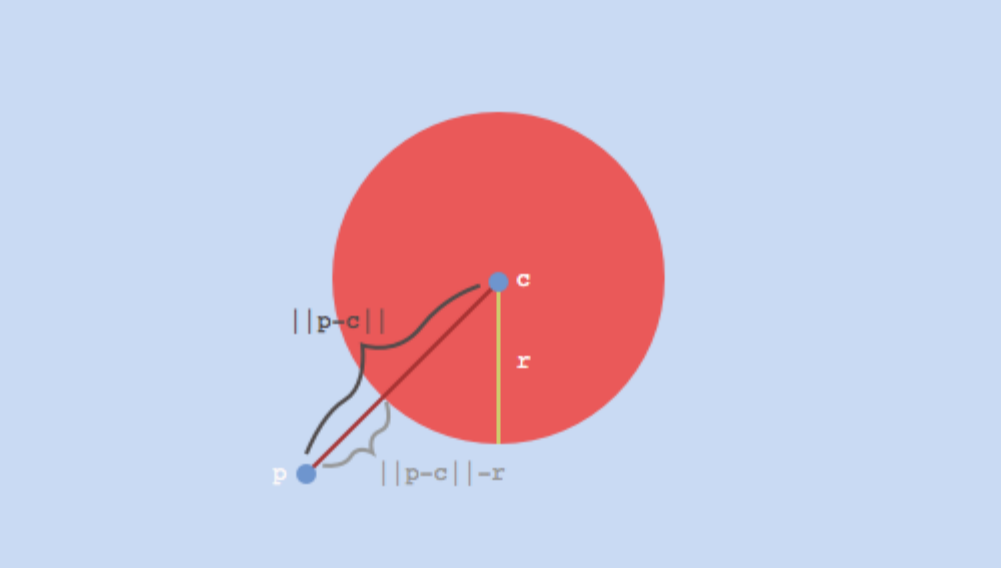

Image 3.6: Visual representation of the signed distance function. [7]

We filled our 3D texture by calling that function for each texel of the texture which looked like image 3.7.

Image 3.7: Rendering of a distance field of a sphere.

3.6 Main Loop

A renderer needs to update the default frame buffer as long as the program is running. It needs to clean all of the screen and draw all of the geometry again. And it needs to be able to repeat this process at least 30 times per second to produce smooth translations.

That's why our program has a main loop. It reads user input, updates variables according to that input, cleans the screen and draws it again.

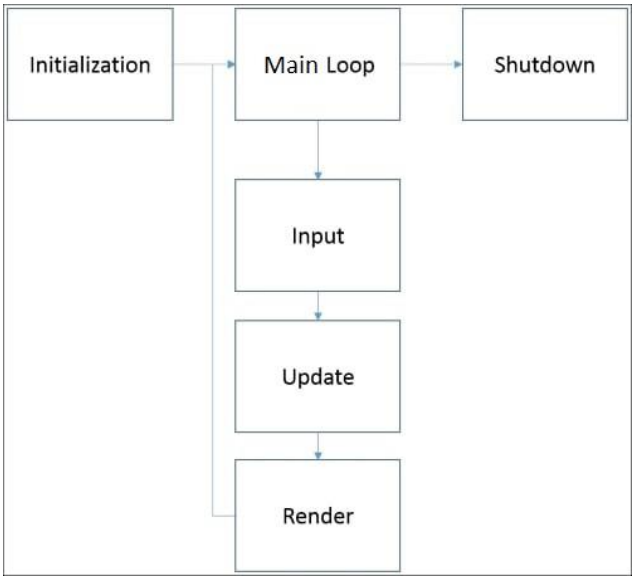

Image 3.8: Program Diagram

3.6.1 Input Handling

SDL records input from devices (like the keyboard, mouse, or controller) as events, storing them in the "event queue". For every update of the main loop, we pull each event off the event queue to process the input. [13]

For each button the user can press to interact with the program, we created a struct. That struct stores pieces of information about the current status of the button. When the user presses that button, we update the status of that button and then update variables that are related to that button.

3.6.2 Updating the Camera

The camera in our renderer defines the properties of the rays that the renderer marches on. All of the rays have a uniform start position, a 3D point (x, y, z). And they all have a unique direction they are going, a normalized 3D vector (u, v, w).

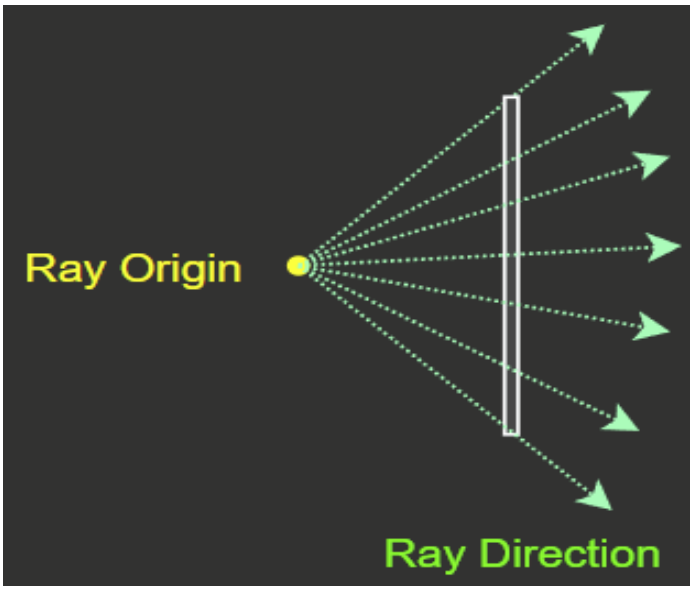

Image 3.9: Yellow dot is ray origin, green dotted lines are rays. [18]

If the user sends an event that is going to change the camera position, we just update the ray origin variable in our raymarch shader. Because it's a uniform, sending new data from the CPU to GPU doesn't slow down the rendering process.

But if the user sends an event that is going to rotate the camera, sending new ray directions from the CPU to GPU breaks the parallelism of the rendering. GPU needs to stop all of the rendering, update all of the unique ray directions and start rendering again. Which causes slowdowns. [15]

So we created a uniform variable called 'camera_rotation'. Updated it with input and send this new uniform to GPU. Then for each ray, we inilitized the ray direction as normal and then rotated them all one by one using a basic rotation matrix.

Rotating all of the rays each frame could feel like a lot. But because the GPUs are designed to run parallel programs faster, rotating each ray is actually a lot faster than sending new data.

3.7 Updating the Distance Field

When the user sculpts the surface of the geometry, all we do is modify the 3D texture that holds the distance information. This changes the scene geometry and gives the user a satisfying sculpting experience.

To update the field, we needed to write to the same 3D texture we just read at the same frame. This caused lots of race conditions to occur. So we duplicated the 3D texture we created at the initialization, used one to read from and the other one to write to. And at the end of the frame, we swapped those two textures.

To make sure we are always up to date with the new modifications, we made sure the renderer raymarches on the last written texture.

3.7.1 Operators

To sculpt the geometry, the users need to be able to add or reduce shapes from the geometry. To add a shape to a distance field, all we needed to was find the signed distance function of the shape and then find the “union” between our existing distance field and the new values the distance function will produce.

If A, B are the distance fields and d is a function that returns the values inside a distance field. Union of A and B is just $\min(dA, dB)$. [17]

To reduce a distance field, we need to subtract the new values from the existing distance field. Again if A, B are the distance fields and d is a function that returns the values inside a distance field. Subtraction of A from B is just $\max(dB, -dA)$. [17]

3.7.2 Finding the Location to Update

The shape we used to sculpt the geometry has a 3D middle point (x, y, z). X and Y values of that middle point is just the position of the mouse cursor on our scene. Those values are uniform so we are just updating those values every frame the user moves the cursor.

The z value of that middle point is harder to find. We need to send an extra ray from the cursor X, Y point to the camera ray direction and use the intersection point of that ray as our Z position. Our first idea was to march that ray once in the GPU and then send the Z value to the CPU to use. But a GPU can't render and send data to the CPU at the same time. Sending data to the CPU broke the parallelism and caused slowdowns. [15]

So we decide to send an extra mouse ray for each ray we used but that doubled the ray count and caused slow downs as well. The solution was calculating the Z value per vertex and sending the value to the rays to use. Modern GPUs can do calculations in vertex shader and send the values to fragment shader (runs once for each ray in our renderer) without any cost [15]. This reduced the extra ray count to just three.

4. RENDERING

The main algorithm that we used in our program was raymarching which we explained in chapter 1.1.2. The only difference, at each step instead of evaluating the SDF, we sampled it from a 3D texture that we stored the distance field.

4.1 Finding the Surface Normals and Diffuse Lighting

But just using raymarching produced the result you can see in image 3.7. We needed to use a simple lighting algorithm to produce better results. We didn't need a complex lighting algorithm, which is out of the scope of the project that we build this semester. And the main goal of a sculpturing program is not to render photorealistic scenes, we focused on more frames.

For every kind of lighting algorithm, we need to find the normals of the geometry. Finding the Surface normals of a distance field is very easy, all we need to do is find the partial derivatives of a point for each direction (x, y, z). We can write it mathematically like this $\nabla f(p) = \left\{\frac{df(p)}{dx}, \frac{df(p)}{dy}, \frac{df(p)}{dz}\right\}$. Those partial derivatives can be computed with small differences, as we know from the definition of derivative. For example, $\frac{df(p)}{dx} \approx \frac{f(p+\{h,0,0\})-f(p)}{h}$. where h is as small as possible. This is called forward differentiation, since it takes the point under consideration $p = (x, y, z)$ and evaluates f at $p' = (x + h, y, z)$, a point in the positive x direction. Backwards and central differences are also possible, which take the following form: $\frac{df(p)}{dx} \approx \frac{f(p+\{h,0,0\})-f(p-\{h,0,0\})}{2h}$. By repeating that formula for each direction and then normalizing the resulting vector, we can find the surface normal of that point. Note that the division by $2h$ can be simplified away since the normalization will rescale the normal anyways to unit length. [17]

Then by using the normals we just found, we can implement really basic diffuse lighting. Which looks like image 4.1 on the next page.

4.2 Fixing Surface Normals and Floating Point Errors

In image 4.1, you can see the sphere looks like it's made of little cubes. This is because we used nearest-neighbor interpolation that we explained in chapter 3.5.2.

To solve this issue, we added another sampler that used tri-linear interpolation. But we activated that sampler just before raymarching and swapped it back to the nearest-neighbor before modifying the actual distance field. That is because linear interpolation changes the real values of the distance field which introduces lots of unwanted artifacts to the scene.

Using tri-linear interpolation as is caused lots of little artifacts as you can see in image 4.2. To produce smooth surfaces surface normals, we needed every precision we gen get from the floats. To fix this issue instead of using exact numbers in our shader code, we used numbers that the GPU can calculate with full precision.

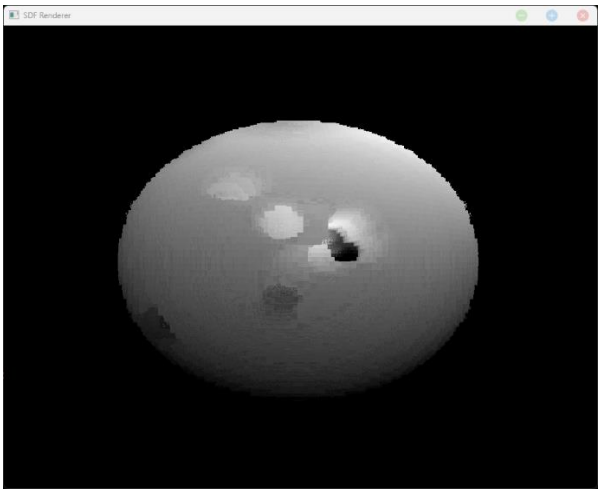

Image 4.1: Raymarching using nearest-neighbor interpolation.

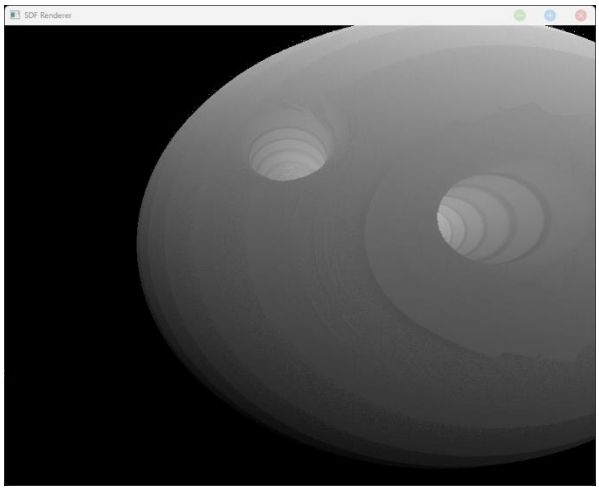

Image 4.2: Floating point errors with tri-linear interpolation.

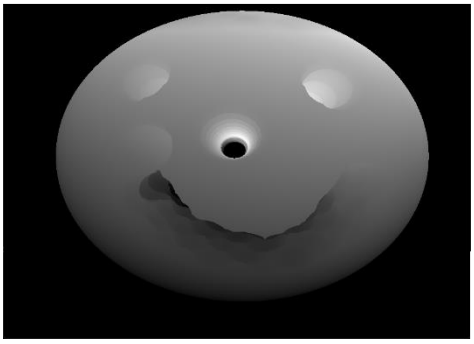

Image 4.3: To the left. Smooth sphere we sculpted at the end.

References

1. Osher, S., Fedkiw, R. (2003). Signed Distance Functions. In: Level Set Methods and Dynamic Implicit Surfaces. Applied Mathematical Sciences, vol 153. Springer, New York, NY. https://doi.org/10.1007/0-387-22746-6_2

2. Jason Cole: Article. https://jasmcole.com/2019/10/03/signed-distance-fields/

3. Rodolphe Vaillant: Article: http://rodolphe-vaillant.fr/entry/86/implicit-surface-aka-signed-distance-field-definition

4. Sieben Corgie Article. https://siebencorgie.rs/posts/nako/

5. Hart, J.C.; Sandin, D.J.; Kauffman, L.H. 1989. Ray tracing deterministic 3D fractals. In Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques, Boston, MA, USA, 31 July–4 August 1989; pp. 289–296.

6. Hart, J.C. 1996. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. Vis. Comput. 12, 527–545.

7. Michael Walczyk Article: https://michaelwalczyk.com/blog-ray-marching.html

8. Hadji-Kyriacou, A.; Arandjelovi´c, O. 2021. Raymarching Distance Fields with CUDA. Electronics 2021, 10, 2730.

9. Sebastian Aaltonen. 2018. Advanced Graphics Techniques Tutorial: GPU-Based Clay Simulation and Ray-Tracing Tech in 'Claybook'. GDC 2018

10. Alex Evans. 2015. Learning from Failure: a Survey of Promising, Unconventional and Mostly Abandoned Renderers for ‘Dreams PS4’, a Geometrically Dense, Painterly UGC Game. SIGGRAPH 2015

11. : Chris Wallis, Article: https://wallisc.github.io/rendering/2021/04/18/Fullscreen-Pass.html

12. 2018. The Rust Programming Language. https://www.rust-lang.org/

13. 2020. SDL Website. https://www.libsdl.org/

14. Joey de Vries. Website. https://learnopengl.com/

15. John Kessenich, Graham Sellers, and Dave Shreiner. OpenGL The Red Book.

16. Wikipedia images.

17. I. Inigo Quilez: Article. https://iquilezles.org/articles/raymarchingdf/

18. Rito15. https://rito15.github.io/