Øivind Albrigtsen

A Web Based Scripting Environment for Creating and Interacting with Ray Marched 3D Graphics.

Master's thesis in Datateknologi
Supervisor: Michael Engel
July 2021

Øivind Albrigtsen

A Web Based Scripting Environment for Creating and Interacting with Ray Marched 3D Graphics.

Master's thesis in Datateknologi
Supervisor: Michael Engel
July 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

A Web Based Scripting Environment for Creating and Interacting with Ray Marched 3D Graphics.

Øivind Albrigtsen

Abstract

This thesis presents a problem with a branch of computer graphics called ray marching where there is a lack of accessibility and usability. This was an interesting problem for me personally to work on as I have been making 3D rendering engines as a hobby for several years, so the technical challenge of making a rendering engine where most of the engine is written in shader code was interesting. I have also been involved with the procedural art community for several years as well. Procedural art is an art style where computer programs and mathematics are used to make images and video. Ray marching is one of the techniques often used for creating art, so I have seen first hand the gap between people who are very experienced and are capable of using ray marching and those who can't.

The thesis then details the research project that was undertaken and the software solution developed to solve this problem. The result of the project is that a scripting language and corresponding scripting environment was made for the purpose of creating and interacting with ray marched graphics. The environment presents the user with a GUI for changing the properties of the graphics and a scripting language that lets the user build graphics step-by-step using simpler primitives. Some examples of graphics created using this scripting environment and the corresponding scripts are also included.

Sammendrag

Denne tesen presenterer et problem innen en gren av datagrafikk kalt ray marching der det er en mangel av brukervenlig og tilgjengelig programvare. Dette var et interessant problem å jobbe med for meg pesonlig siden jeg har laget 3D grafikkmotorer som en hobby i flere år, så den tekniske utfordringen ved å lage en grafikkmotor der mesteparten av motoren er skrevet i GLSL var interessant. Jeg har også vært involvert i *procedural art* miljøet i flere år. *Procedural art* er en kunstform der dataprogrammer og matematikk er brukt for å lage bilder og videoer. Ray marching er ofte brukt som en teknikk i *procedural art*, så jeg har vitnet den store forskjellen mellom folk som har nok erfaring og har evner til å bruke ray marching og de som ikke kan.

Denne tesen går så over detaljene til forskningsprosjektet og programvareløsningen utviklet for å løse problement. Resultatet av prosjektet er at et scripting språk og tilhørende scripting miljø var laget for den hensikt å skape og interagere med ray marched grafikk. Miljøet viser brukeren en GUI for å endre egenskapene til grafikken og presenterer et scriptespråk som lar brukeren bygge grafikk steg for steg ved bruk av enklere former. Noen eksempler på grafikk laget ved å bruke denne programvaren og korresponderende script er også inkludert.

Contents

Abstract . . . **iii**
Sammendrag . . . **v**
Contents . . . **vii**
Figures . . . **ix**
Tables . . . **xi**
Code Listings . . . **xiii**
Preface . . . **xv**
Acronyms . . . **xvii**
Glossary . . . **xix**
1 Introduction . . . **1**
1.1 Background . . . 1
1.1.1 Computer Graphics . . . 1
1.1.2 Triangle rasterization . . . 1
1.1.3 Ray based rendering . . . 3
1.2 Problem . . . 6
1.2.1 Existing software . . . 6
1.3 This thesis . . . 10
1.3.1 Widely available . . . 10
1.3.2 Easy to use . . . 10
2 Requirements . . . **13**
2.1 Methodology . . . 13
2.2 Functional requirements . . . 13
2.3 Non-functional requirements . . . 14
3 Language Design . . . **15**
3.1 Semantics and syntax . . . 15
3.1.1 Shape . . . 16
3.1.2 Material . . . 19
3.1.3 Data . . . 20
3.1.4 Other . . . 21
3.2 Javascript . . . 21
3.3 Usage . . . 22
3.3.1 Multiple primitives . . . 22
3.3.2 Primitives occupying the same space . . . 23
3.3.3 Primitives moving in a coordinated way . . . 24

3.3.4 Multiple textures from URLs 25
3.3.5 Loading a JavaScript library 26
4 Implementation 27
4.1 Programming languages 27
4.1.1 TypeScript 27
4.1.2 JavasScript 30
4.1.3 GLSL 30
4.2 Libraries 30
4.2.1 Dear ImGui js 30
4.3 Tools 32
4.3.1 Github 32
4.3.2 Chrome 33
5 Deployment 35
5.1 Clone 35
5.2 Install 35
5.3 Build 35
5.4 Run 36
6 Evaluation and Discussion 37
6.1 Evaluation 37
6.2 Discussion 38
7 Conclusion 41
Bibliography 43
A Appendix A 45
B Appendix B 57
C Appendix C 61

Figures

1.1 A set of polygons forming a mesh. 2
1.2 A set of pixels forming an image. 2
1.3 Diagrams of 4 steps of ray marching. 4
1.4 Height map colored as landscape. 5
1.5 Height map colored as landscape with height curves. 5
1.6 Height map with a filled height curve. 5
1.7 Filled height curve colored by its normal vectors. 5
1.8 Shadertoy shader. 7
1.9 Full torus. 9
1.10 Torus cut in half. 9
1.11 Full torus moved. 10

3.1 Multiple primitives in a scene. 23
3.2 Primitives intersecting each other. 24
3.3 Primitives moving coordinated. 24
3.4 Multiple textures. 25
3.5 Loading a library. 26

4.1 Project structure. 28
4.2 Language module structure. 29
4.3 GUI. 31
4.4 Script input 32
4.5 Objects window 32

Tables

3.1 Table describing how types in our language map to actual JavaScript types/objects. 16
3.2 Table describing how to use the built in functions. 22

Code Listings

1.1 Lines 200 to 223 of the shader code. 8
1.2 Code calling the torus function. 9
1.3 Bounding box cutting the torus in half. 9
3.1 Syntax for creating a sphere. 15
3.2 Shape syntax for creating a sphere with a position and radius. . . . 16
3.3 Setting a javascript function to update the position of a sphere. . . . 16
4.1 Code example creating a shader from a scene. 28
4.2 Code example creating some GUI elements in Dear ImGui. 31
5.1 NPM command to install dependencies 35
5.2 Running the build script with NPM from the command line 36
5.3 Running the start script with NPM from the command line 36
6.1 Code for creating 252 spheres. 37
6.2 Code for creating 40 spheres with a position property initialized to a random position. 38
6.3 Code for creating 20 spheres with a position property being updated. 38
A.1 Shadercode from shadertoy.com 45
B.1 Fragment shader generated by the compiler. 57
C.1 Script to generate a scene with multiple primitives. 61
C.2 Script to generate a scene with intersecting primitives. 61
C.3 Script to generate a scene where primitives move in a coordinated way using JavaScript scripting. 62
C.4 Script to create a planet using multiple textures fetched from the web. 63
C.5 Script to load a JavaScript library from the scripting environment. . 64

Preface

The main motivation for this thesis is the increase in popularity of ray based rendering that has taken place these last few years and the lack of accessibility hindering this. As ray based rendering techniques become more widely used there is a need for better tooling as currently only very knowledgeable and experienced people are able to work with this technology. Having tools that are easier to use can also in turn speed up the adoption of these techniques as more graphical artists become aware of the capabilities and the possible use cases.

Acronyms

API	Application programming interface. 10
GPU	Graphics processing unit. 1, 2, 10
GUI	Graphical user interface. 11, 30

Glossary

GLSL GLSL (OpenGL Shading Language) is a high-level shading language with a syntax based on the C programming language.. 11, 27, 28, 30, 35

mesh A set of vertices and edges combined to form a polyhedral object.. 1

Monte Carlo method Algorithms that use random sampling to obtain numerical results.. 3

NPM Node Package Manager. The package manager for the Node ecosystem.. 35

shader Computer program for programming the GPU's rendering pipeline.. 2, 3, 7

smooth Property of a function, where it has continuous derivatives.. 4

tsc The TypeScript compiler.. 35

WebGL WebGL is a JavaScript API for rendering interactive 3D graphics within any compatible web browser.. 10, 31, 33

Chapter 1

Introduction

In this section I will outline the necessary background to put the problem in context, the problem itself as I see it and how this thesis tries to solve it.

1.1 Background

Polygon rasterization has been the standard way of working with computer graphics for decades. Hardware is now designed specifically to speed up polygon rasterization as that is what is used in 3d animation and video games.
Ray based rendering is a technique that has been used since the 60s [1]. This way of rendering computer graphics never gained prominence even though it has certain technical advantages over traditional polygon rendering. The main reason for this is that creating high-fidelity 3d scenes is too difficult and rendering them is too compute intensive. Now Graphics processing unit (GPU)s are becoming fast enough that complex ray based scenes can be rendered in real time. And this development is speeding up as modern GPUs are adding acceleration for ray based rendering techniques[2].

1.1.1 Computer Graphics

Computer graphics refers to the methods for creating and manipulating digital imagery. In this paper the most relevant parts of computer graphics are those methods employed in real-time 3d interactive programs and pre-rendered 3d imagery such as video games and animated movies.

1.1.2 Triangle rasterization

The most common way of representing objects to be rendered are to build them up using vertices and edges which form triangles which combined forms a mesh, the surface of an object. These objects can then be rendered using a technique called triangle rasterization.

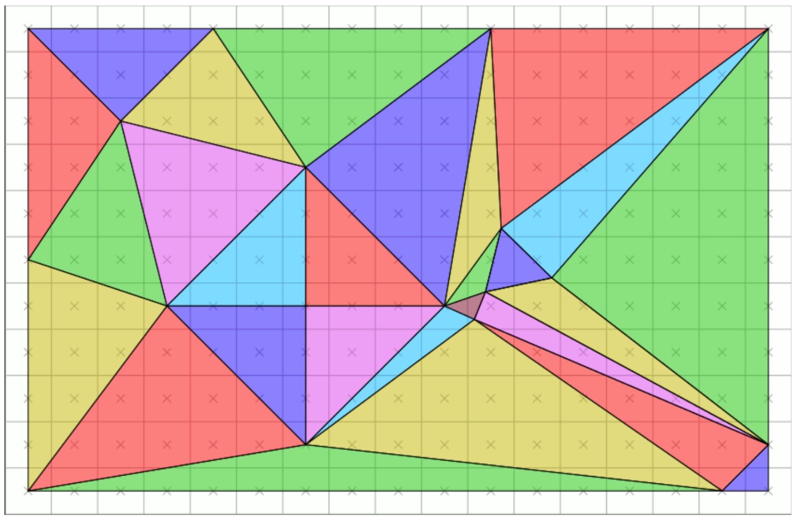


Figure 1.1: Polygons before they have been rasterized.

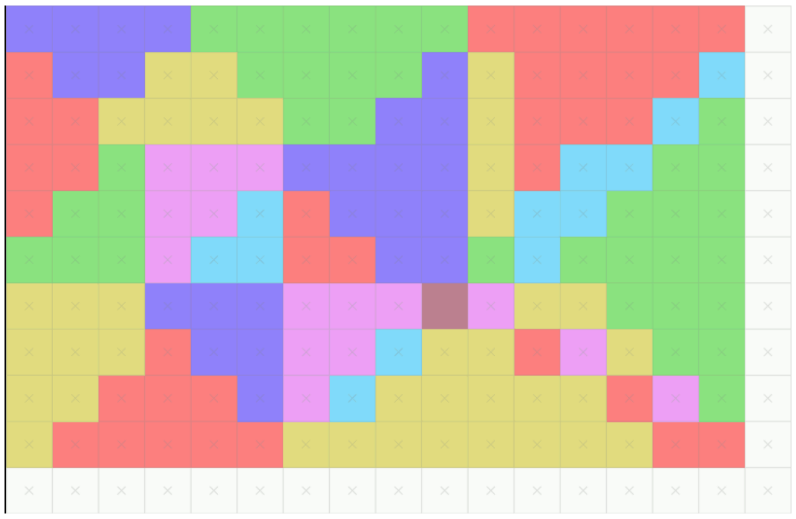


Figure 1.2: The polygons from 1.1 rasterized using the top-left rule.

Triangle rasterization involves iterating over all the triangles to be drawn, translating the vertices positions into screen-space positions and then iterating over all the pixels enclosed by the vertices and their edges. To ensure certain properties in the finished image certain rules have to be followed when determining if a pixel should be rasterized, a common one is the top-left rule [3]. Each pixel determined to be inside the polygon is then colored based on the output of some computation done on the GPU. This computation is described using shader code.

1.1.3 Ray based rendering

There are several related terms used in the graphics community for a set of techniques similar in some aspects. These terms are *raytracing*, *pathtracing* and *raymarching*. The core similarity between these techniques is that they use rays to mimic, to various degrees, the behavior of photons. This is then used to determine the color of each pixel in the rendered image. Another difference from the polygon rasterization mentioned above is that for these ray based techniques the entire program can be inside a shader.
The lines between these different techniques are blurry and the terms are sometimes used interchangeably.

Ray tracing and path tracing

In general ray tracing means the image is rendered by tracing the path light would take in reverse. Starting from the camera and into the scene, calculating how it would interact with the objects to gain its color. Going in reverse is not technically necessary, but it saves a lot of computation as most photons bouncing around in the scene will not hit the camera and will therefore not affect the image. These photons are excluded by starting from the camera and going in reverse.
Path tracing is a form of ray tracing where Monte Carlo methods are used as well to more accurately simulate lighting conditions at the points the ray intersects the scene.
To determine what object the ray will hit, a ray intersection formula is used.
As an example, a ray with direction $\overrightarrow{d}$ and origin $\overrightarrow{o}$ is defined as the parametric equation in 1.1 and a sphere with center $\overrightarrow{c}$ and radius r is defined as 1.2. Where $\overrightarrow{p}$ is a point on the sphere.

$$\overrightarrow{r}(t) = \overrightarrow{o} + t\,\overrightarrow{d}. \tag{1.1}$$

$$(\overrightarrow{p} - \overrightarrow{c})(\overrightarrow{p} - \overrightarrow{c}) = r^2. \tag{1.2}$$

The intersection between the ray and the sphere can then be determined by combining these equations, giving us equation 1.3, and solving for t.

$$(\overrightarrow{o} + t\,\overrightarrow{d} - \overrightarrow{c})(\overrightarrow{o} + t\,\overrightarrow{d} - \overrightarrow{c}) = r^2. \tag{1.3}$$

Solving for t leads to solving a polynomial of degree 2, meaning there will be either zero solutions meaning no intersection, one solution meaning the ray is tangent to a point on the sphere surface or two solutions meaning the ray intersects the sphere twice, once when entering and once more on exit [4]. Finding the visible point is trivial with one solution as we already have 1.1 for determining the ray position with a given t. When we have two solutions we need to determine which of them corresponds to the entry point, as that is the point on the sphere that we can actually see.

Ray marching

Ray marching is similar to ray tracing in that we are shooting rays out from the camera into our scene and determining how they interact with the objects. The main difference is that, as the name implies, the ray object intersections are not solved analytically, instead the ray is marched forward some distance until it has hit an object. This allows for a different way of describing objects, as we can now render any function f, where f is defined as in equation 1.4.

$$f : \mathbb{R}^3 \longrightarrow \mathbb{R}. \tag{1.4}$$

Of course there are properties f could have to make it more or less useful for modeling objects. Usually a function that is supposed to model some discrete object would be smooth so that it creates a surface without holes or jumps. The functions for some basic shapes are given in [5].
These functions map each point in space to a value. This value can be viewed as a density, like the density of a cloud varying as you move through it. The color of the pixel generating this ray would then be determined by adding up the densities at each step as we march through the cloud. This would give us a cloud that is translucent in some areas and opaque in others, depending on the thickness of cloud blocking our ray.
This could be made even more realistic by shooting more rays out at each point while we are marching through the cloud. We could then take into account the light transfer happening inside the cloud itself.
Fig 1.3 shows the four steps of the ray marching process.

- 1. Shooting the ray through the volume described by f.
- 2. Determining the values at each point the ray marches through.
- 3. Calculate each sampled points color contribution.
- 4. Combine the calculated values to get the final pixel color.

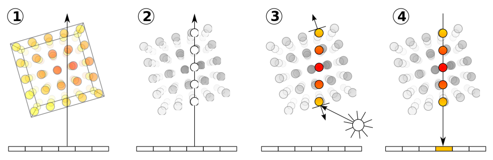


Figure 1.3: Four steps of coloring with ray marching. By "Florian Hofmann" licensed under CC BY-SA 3.0

A different way to use f is to think of a threshold for the density function which would then describe an implicit surface enclosing the area with lower density than the threshold.
The concept is easier to demonstrate by removing one dimension and looking at it as a familiar 2d height map, like the one shown in fig 1.4. The 2d height map is created by a function g, $g : \mathbb{R}^2 \longrightarrow \mathbb{R}$ and we can draw height curves on it by changing the color for height values close to the height value h we want a height

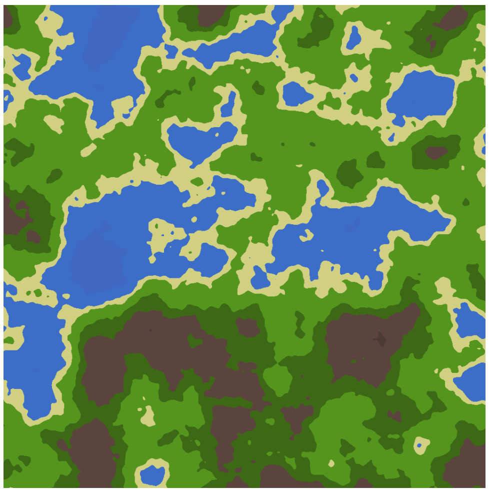


Figure 1.4: A standard 2d map of a landscape.

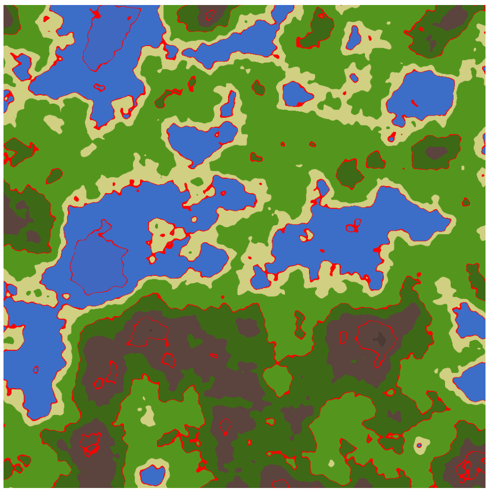


Figure 1.5: The 2d map with height curves.

curve for. This means finding the points where $g(x, y) \approx h$. In fig 1.5 these height curves are drawn in red for a few different values of h.

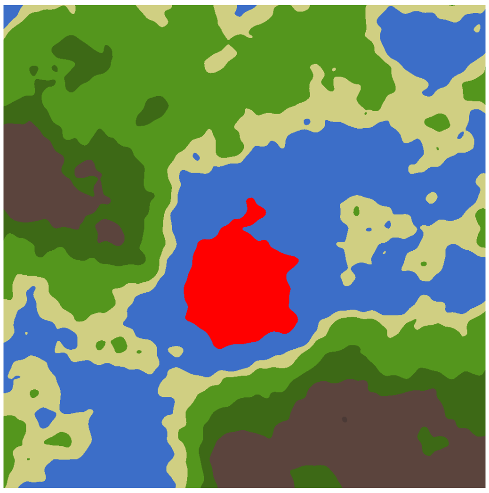


Figure 1.6: A height curve filled in red.

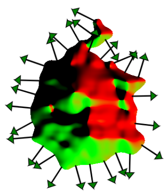


Figure 1.7: The height curve with colors and arrows showing the gradient.

By filling in one of these height curves, like in 1.6, we get some 2d shape and since we have g we can find the gradient of g, ∇g. $\nabla g : \mathbb{R}^2 \longrightarrow \mathbb{R}^2$ is a mapping from each point in the image to a vector pointing in the direction of greatest increase with a magnitude equal to the rate of increase in that direction [6]. In 1.7 ∇g has been used for the fill color and plotted as lines for some points on the boundary of the shape. The important insights here is that this height-curve-enclosed shape would be a 3d object when applying the same idea to f and that these arrows, in f, would correspond to the normal vectors of the surface of the 3d object.

The last piece of the ray marching puzzle is to speed up the marching process. Until now we have described a system where the ray is marched forward in fixed increments and tested against functions describing the scene. This is of course a slow way of doing things. Instead what we want is to be able to determine some lower bound where no object can be hit. That is, a distance d from the current ray position where we know we can move without interacting with anything.
To achieve this we can impose a requirement on f, that it needs to be a distance function. A distance function is also a function $\mathbb{R}^3 \longrightarrow \mathbb{R}$, but instead of describing a density it describes a distance to a surface.

$$sphere(\overrightarrow{p}) = ||\overrightarrow{p}|| - r. \tag{1.5}$$

1.5 is the distance function for a sphere centered at $(0, 0, 0)$ with radius r. This is also a signed distance function as it will give negative distances when we are inside the object. When ray marching a scene with this sphere object we would pass in our current ray position as $\overrightarrow{p}$ and the result would tell us how far it is safe to move. Moving any further than this safe distance could put us inside the sphere. If the scene has multiple of these distance functions describing multiple objects we would take the minimum of these distances as the maximum safe distance to move. Once a distance function returns ≈ 0 for our ray position, we know that we have hit a surface.

So now we have a way to efficiently ray march scenes described using distance functions and these functions have gradients that can be used to calculate color information. That is all the background needed for the general implementation this ray marcher.

1.2 Problem

Since we've been in a polygon based world for so long both our hardware and software has been built around it. Using a ray-based renderer opens many new possibilities, in terms of the complexity of scenes and the fidelity of effects that can be employed [7]. But the community of computer graphics creators, hobbyists and up, does not have easy access to play with this technology at a fundamental level. This can lead to interesting ideas that are hard to or impossible to achieve with polygon rasterization simply never coming to fruition because the artist didn't have the tools available to create what they envisioned.

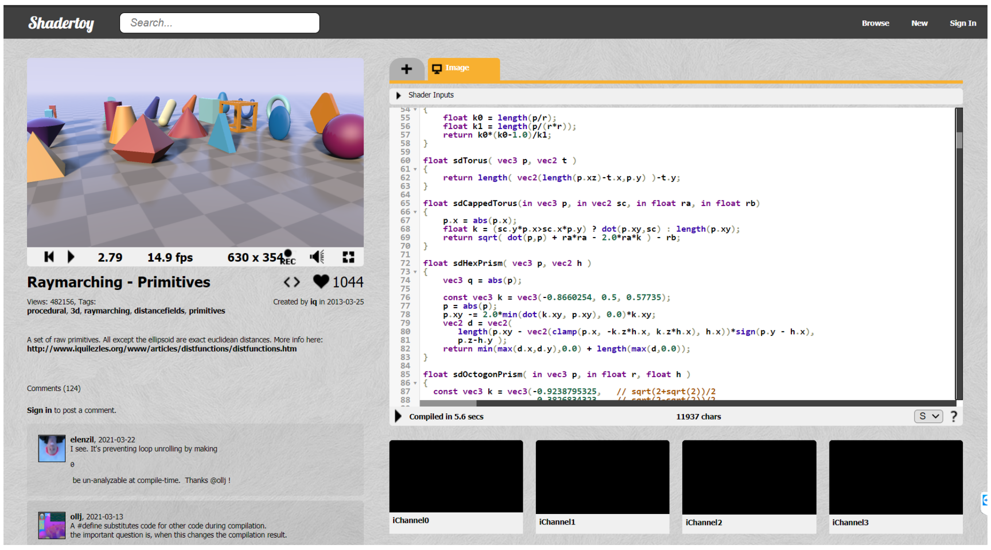
1.2.1 Existing software

There are a few existing software solutions for working with ray marching, most similar to this project is Shadertoy.com[8].

Shadertoy

Shadertoy is a website for playing around with and experimenting with shaders, hence the name. It is a creation of Inigo Quilez and Pol Jeremias. It's not primarily made for ray marching, all kinds of shader code can be written and executed. People make all kinds of graphical effects, but the ray marching techniques are among the most popular [9].

Figure 1.8 shows what the user sees when interacting with a shader. On the left side is the actual render output and on the right you can see the shader code. There is usually very little to no interactivity with the render window. You can't use the mouse to click things or look around and there is no way to move the camera freely. In this particular scene, the second most popular scene on the site [10], the camera rotates around the shapes in the middle on a predetermined track and clicking with the mouse changes your position along that track.



Raymarching - Primitives 1044

Views: 482156, Tags:
procedural, **3d**, **raymarching**, **distancefields**, **primitives**

Created by **iq** in 2013-03-25

A set of raw primitives. All except the ellipsoid are exact euclidean distances. More info here:
http://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm

Comments (124)

Sign in to post a comment.

elenzil, 2021-03-22
I see. It's preventing loop unrolling by making

0

be un-analyzable at compile-time. Thanks @ollj !

ollj, 2021-03-13
A #define substitutes code for other code during compilation.
the important question is, when this changes the compilation result.

```
    float k0 = length(p/r);
    float k1 = length(p/(r*r));
    return k0*(k0-1.0)/k1;
}

float sdTorus( vec3 p, vec2 t )
{
    return length( vec2(length(p.xz)-t.x,p.y) )-t.y;
}

float sdCappedTorus(in vec3 p, in vec2 sc, in float ra, in float rb)
{
    p.x = abs(p.x);
    float k = (sc.y*p.x>sc.x*p.y) ? dot(p.xy,sc) : length(p.xy);
    return sqrt( dot(p,p) + ra*ra - 2.0*ra*k ) - rb;
}

float sdHexPrism( vec3 p, vec2 h )
{
    vec3 q = abs(p);

    const vec3 k = vec3(-0.8660254, 0.5, 0.57735);
    p = abs(p);
    p.xy -= 2.0*min(dot(k.xy, p.xy), 0.0)*k.xy;
    vec2 d = vec2(
       length(p.xy - vec2(clamp(p.x, -k.z*h.x, k.z*h.x), h.x))*sign(p.y - h.x),
       p.z-h.y );
    return min(max(d.x,d.y),0.0) + length(max(d,0.0));
}

float sdOctogonPrism( in vec3 p, in float r, float h )
{
  const vec3 k = vec3(-0.9238795325,   // sqrt(2+sqrt(2))/2
```

Compiled in 5.6 secs 11937 chars

Figure 1.8: This is what a user sees when interacting with a shader using shadertoy.com.

This means that the main way for a user to interact with this scene is by directly modifying the shader code on the right. Code listing 1.1 shows a representative snippet of the 632 line long shader program. The program in its entirety is included in appendix A.

Code listing 1.1: Lines 200 to 223 of the shader code.

```
// vertical
float sdCone( in vec3 p, in vec2 c, float h )
{
    vec2 q = h*vec2(c.x,-c.y)/c.y;
    vec2 w = vec2( length(p.xz), p.y );

  vec2 a = w - q*clamp( dot(w,q)/dot(q,q), 0.0, 1.0 );
    vec2 b = w - q*vec2( clamp( w.x/q.x, 0.0, 1.0 ), 1.0 );
    float k = sign( q.y );
    float d = min(dot( a, a ),dot(b, b));
    float s = max( k*(w.x*q.y-w.y*q.x),k*(w.y-q.y)  );
  return sqrt(d)*sign(s);
}

float sdCappedCone( in vec3 p, in float h, in float r1, in float r2 )
{
    vec2 q = vec2( length(p.xz), p.y );

    vec2 k1 = vec2(r2,h);
    vec2 k2 = vec2(r2-r1,2.0*h);
    vec2 ca = vec2(q.x-min(q.x,(q.y < 0.0)?r1:r2), abs(q.y)-h);
    vec2 cb = q - k1 + k2*clamp( dot(k1-q,k2)/dot2(k2), 0.0, 1.0 );
    float s = (cb.x < 0.0 && ca.y < 0.0) ? -1.0 : 1.0;
    return s*sqrt( min(dot2(ca),dot2(cb)) );
}
```

This way of interacting with the shaders makes it very hard for other people to modify or build upon them. The shader exposes very complicated GLSL code directly to the user and complicated shaders often use many clever tricks to increase performance. These tricks are not hidden behind some abstraction layer so that you can interact with the rest of the code without understanding the optimization tricks used. Instead they are directly embedded in the code and very closely coupled to the functions describing the scene.

As an example of a trick used to optimize a scene, the shader in 1.8 groups the objects in the middle into different bounding boxes. A bounding box is a tool often used for collision detection, you place a bounding box around the objects you want to check for collision against and then you can use, in this case, a box distance function to see how far away from the box you are. This saves computation as you now only check for collision against a single box until you are inside that box then you start checking against the grouping of objects inside.
This means that when a user wants to, for example, move the torus up a bit. They would locate the code that calls the torus function.

Code listing 1.2: Code calling the torus function.

```
// bounding box
    if(sdBox(pos-vec3(0.0,0.3,-1.0),vec3(0.35,0.3,2.5))<res.x)
    {
    // more primitives
    res = opU(res, vec2(sdBoundingBox(pos-vec3(0.0,0.25, 0.0),
        vec3(0.3,0.25,0.2), 0.025), 16.9));
    res = opU(res, vec2(sdTorus((pos-vec3(0.0,0.80, 1.0)).xzy,
      vec2(0.25,0.05)), 25.0));
    res = opU( res, vec2( sdCone(pos-vec3(0.0,0.45,-1.0),
      vec2(0.6,0.8),0.45), 55.0));
    res = opU(res, vec2(sdCappedCone(pos-vec3(0.0,0.25,-2.0), 0.25, 0.25, 0.1
        ), 13.67));
    res = opU(res, vec2(sdSolidAngle(pos-vec3(0.0,0.00,-3.0),
        vec2(3,4)/5.0, 0.4), 49.13));
    }
```

Code listing 1.2 shows line 325 to 334 where the torus signed distance function is called. It's not clear what all the numbers mean as they are just exposed to us without much context, but one can reason that the first argument to the sdTorus function is its position since that is where the "pos", meaning "current ray position", variable is used. So lets modify the y component of the position and see what happens.

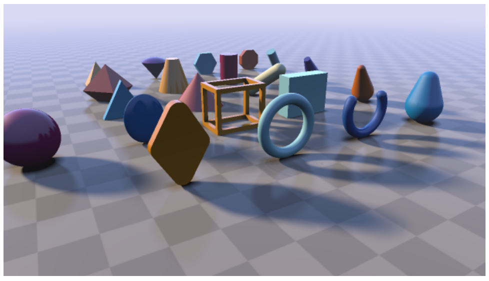


Figure 1.9: Torus with y position 0.3.

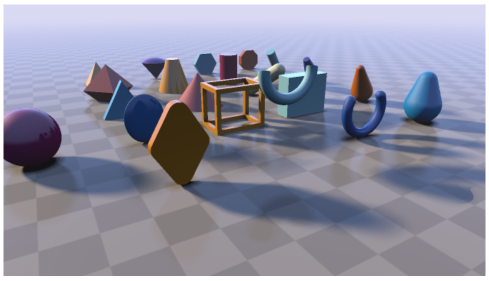


Figure 1.10: Torus with y position 0.8.

Fig 1.9 shows the normal full torus at y position 0.3 and in fig 1.10 the torus is moved to y position 0.8 and has been cut in half by a bounding box. This is where a deeper understanding of the shader code is required to make modifications. A person seeing this code for the first time might assume that it is enough to expand the bounding box on line 326 to make the torus whole, but this does not work. Instead what is required is to change a different bounding box shown in code listing 1.3 on line 401.

Code listing 1.3: Bounding box cutting the torus in half.

```
vec2 tb = iBox( ro-vec3(0.0,0.4,-0.5), rd, vec3(2.5,0.41,3.0) );
```

After changing the y of this bounding box the torus becomes visible as expected 1.11.

The Shadertoy tool is built by and for people who are very knowledgeable about shader programming, it is therefore not built with ease of use in mind.

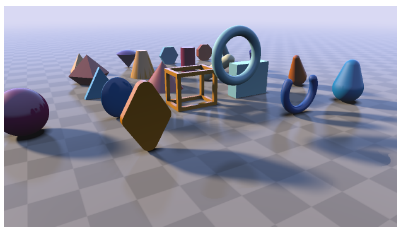


Figure 1.11: Torus with y position 0.8. The torus is no longer cut in half.

This being the most popular tool for making and sharing these types of graphics means there is a high barrier of entry into the field. As GPU computation power increases and the computation costs of ray marched graphics becomes less and less significant there will be a need for a tool with lower barriers of entry.

1.3 This thesis

This paper details my software solution to this problem. It is a web-based scripting environment for creating, interacting with and sharing ray marched graphics. The two key aspects of the software was for it to be widely available and easy to use.

1.3.1 Widely available

The software being widely available so that anyone can participate in creating, modifying or simply viewing other peoples creations was a core requirement from the start. This requirement made decisions relating to the technology stack clear. The software had to be platform independent, this would mean using some 3d framework that works on all platforms. This excluded using any OS specific Application programming interface (API) like DirectX or Metal as these only work on windows and OSX respectively and therefore multiple front-ends for the software would have to be written. Instead something like OpenGL was considered, as this would allow me to write the graphics portion of the program only once.
Having the software be platform independent is one step towards higher availability, but it would still require a good bit of work to target, compile and distribute for all possible platforms. To eliminate this issue i decided to use WebGL instead. This makes distributing the application easy, it can be hosted on a website and any modern browser on any OS can access and use it. More technical details of the solution are given in Chapter 3 and Chapter 4.

1.3.2 Easy to use

The effort to make the software easy to use is focused on to main aspects. It should be easy to create a ray marched scene and any scene created in the tool should have a high degree of modifiability and interactivity by default.

To make it easy to create scenes and make sure they fulfill the need for modifiability and interactivity a layer of abstraction was needed. Instead of interacting directly with GLSL shader code the user manipulates objects that are in turned converted to GLSL for them.

The interactivity and modifiability exists on two levels. Interactivity is created by the fact that all scenes by default have a movable camera. The camera doesn't have to be specified manually in shader code every time a user wants a movable camera in their scene. Instead if the creator wants to specify certain attributes of the camera they can do so in the script. Also creating interactivity is the fact that attributes of objects in the scene are available to be changed via a simple Graphical user interface (GUI) and the effects are visible in real-time.
Modifiability is achieved by the abstraction of raw shader code into discrete objects and a scripting language surrounding them. This means that the scene definition is easy to read as it just contains lines adding objects and lines scripting their behavior in javascript.

Chapter 2

Requirements

This section will detail some of the requirements this software was developed against and the development methodology used.

2.1 Methodology

I will not go into great detail explaining my development methodology as this was a research project and requirements and development practices changed rapidly throughout the project. The development can be characterized as agile in that it largely consisted of solving and creating new issues on the fly instead of any lengthy planning process [11]. Following a stricter methodology would have added a lot of extra managerial work to the project and as I was only one person working it felt unnecessary.

2.2 Functional requirements

Interaction

The software will allow to user to interact with the scene being displayed.

Input

The software will take user input text to write the scripts and keyboard and mouse movement to move the camera around the scene.

GUI

The software will display GUI elements to show the state of the properties of the objects in the scene.

Feedback

The software will return any error messages generated when running the script.

2.3 Non-functional requirements

Interactivity

The system must allow real time interaction, meaning the changes the user makes should take effect immediately without any long delay like a compilation step. The change should appear within the time between frames, i.e within 33.34ms.

GUI

The GUI elements presented will be tailored to the type of the property being represented. So for example editing a color will display a color picker element.

Feedback

The error messages generate when running the script shall be displayed in the "Error and Warnings" window.

Chapter 3

Language Design

This section will go into more detail about the scripting language created for this software. The language has certain semantics and syntax to use these semantics as a base. On top of this base it also supports JavaScript as a sort of embedded scripting language within itself.

The language is designed to be easy to use and to allow a new user to quickly get objects on screen. Once they have something showing it also allows them to easily add and modify the properties of the things on screen.

3.1 Semantics and syntax

This section will go over all the core semantics of the language and the related syntax. The semantics can be split into 3 main parts. Shape semantics, those semantics describing the actual shape of object, material semantics, describing the material of the object and data semantics, describing data to be used to shape or color objects.

In general to use these semantics you start with a base object, this object then affords certain properties that can be optionally added and given values.

Shape and material semantics are used directly when creating a new object for the scene.

Code listing 3.1: Syntax for creating a sphere.

```
object("My␣Sphere")
    .shape(sphere)
    .material(singleColored);
```

Each shape has some shared set of properties all shapes have and some that are unique to itself. Materials mostly have unique properties. To access these properties a builder-like syntax is used where the properties are simply added after the shape or material. There is no significant whitespace or newlines, the indentation style just makes it easier to read.

Name	Data
float	[number]
vec2	[number, number]
vec3	[number, number, number]
color	[number, number, number] (number in range 0-1)
string	string
function	function

Table 3.1: Table describing how types in our language map to actual JavaScript types/objects.

Code listing 3.2: Shape syntax for creating a sphere with a position and radius.

```
object("My␣Sphere")
    .shape(sphere)
        .position()
        .radius()
    .material(singleColored);
```

These properties are split into two groups, *variable* and *static*. Variable properties can vary without recompiling the scene and can be modified in the GUI or by some JavaScript running each frame. Static properties have to be hardcoded in the shader code and can only be changed by recompiling. Position and radius are examples of variable properties. What texture to use is an example of a static property.

Code listing 3.3: Setting a javascript function to update the position of a sphere.

```
object("My␣Sphere")
    .shape(sphere)
        .position()
            .setUpdate((dt,ft) => {
                return [Math.sin(ft), 0, 0];
            })
        .radius([2])
    .material(singleColored);
```

Listing 3.3 shows a sphere with a JavaScript function updating its position property every frame and a default value of 2 for the radius property. The values of these properties are always one of the types outlined in table 3.1. Name is the name of the datatype and Data is the JavaScript data it holds.

3.1.1 Shape

The shape of an object is the function that determines whether or not any point in space is inside or outside the object. All shape functions have a variable property called position that takes a *vec3*.

Sphere

Property	Type	Variable	Description
radius	float	yes	Determines the radius of the sphere.
UVOffset	vec2	yes	Determines what offset to add to the spheres UV coordinates.

Box

Property	Type	Variable	Description
size	vec3	yes	Determines the size of the box from its center.
UVOffset	vec2	yes	Determines what offset to add to the box's UV coordinates.

BoxFrame

Property	Type	Variable	Description
size	vec3	yes	Determines the size of the box from its center.
thickness	float	yes	Determines the thickness of the boxes making up the frame.

Height map

Property	Type	Variable	Description
size	vec3	yes	Determines the size of the box surrounding the height map.
stepSize	float	yes	Determines how far to step the ray when marching the height map.
thickness	float	yes	Determines the thickness of the surface of the height map.
data	string	no	Reference to some Data that describes the height map.
UVOffset	vec2	yes	Determines UV offset to add to the UV coordinates.

HexagonalPrism

Property	Type	Variable	Description
width	float	yes	Determines width of the prism.
depth	float	yes	Determines the dept/how long the prism is.

InterpolatedShape

Property	Type	Variable	Description
shapeA	object	no	One of the objects to interpolate between.
shapeB	object	no	One of the objects to interpolate between.
weight	float	yes	Determines how much weight to give to shape A, shape B is then weighted at 1-weight.

Torus

Property	Type	Variable	Description
thickness	float	yes	Determines the thickness of the torus ring.
size	float	yes	Determines the size of the torus, i.e the diameter of the whole object.

3.1.2 Material

The material of an object is the function that determines what color to give every point on the surface of the object.

NormalColored

Colors each pixel of the shape based on the normal at that pixel. This material has no properties.

PhongShaded

Property	Type	Variable	Description
color	vec3	yes	Determines the color used in the Phong shading model [12].

SingleColored

Property	Type	Variable	Description
color	vec3	yes	Determines the color of the material.

SquaresShaded

Produces a checkerboard pattern on a shape. This material has no properties.

Textured

Shows a texture or set of texture on the shape. By supplying only the "texture" property only a single texture will be used. If instead any combination of "ambient", "specular", "diffuse" and "shadow" is supplied the object will use a lighting model that takes into account ambient, specular and diffuse lighting as well as shadows.

Property	Type	Variable	Description
texture	string	no	Used to show only a single texture with no lighting effects.
ambient	float	yes	Ambient determines the strength of the diffuse texture.
specular	string	no	Texture used fr specular reflections of incoming light.
diffuse	string	no	The diffuse color of the shape.
shadow	string	no	If supplied this texture is displayed on the part of the shape that is in shadow.

UVColored

The color at each pixel is the shapes' UV coordinates at that pixel. This material has no properties.

3.1.3 Data

Data is used for things that can't easily be transferred using the uniforms system like the other properties does. Data is used by passing in a string referencing the name of some data object.

Texture2D

A Texture2D is a data object holding a 2d texture, this will appear as a sampler2d in the shader. There are two main ways of constructing a Texture 2d, fromCanvas constructs one using the canvas2D API and fromFetch constructs one by fetching some data using a HTML img element.

FromCanvas

Property	Type	Variable	Description
setInit	function	no	Set the initial state of the texture. The function receives the with and height of the texture and a reference to a canvas 2d context that can be used for drawing on the texture.
setUpdate	function	no	Same as setInit but is called every frame to update the texture.

FromFetch

Property	Type	Variable	Description
setInit	function	no	Set the initial state of the texture. The function receives a callback function setSrc used to set the source data of the texture. This can be a URL pointing to some image.
setUpdate	function	no	Same as setInit but is called every frame to update the texture.

3.1.4 Other

There are also some features available via free functions that can be used anywhere in the script. Table 3.2 lists all of these functions and how to use them.

3.2 Javascript

JavaScript is a dynamic scripting language that runs in the browser. As websites has added more features and in some instances replaced desktop applications JavaScript has become more and more used.
The scripting language leverages JavaScript to expand what is possible to create using the language. In general when a property is updated by calling setUpdate

Name	Usage
setSkybox	Takes either a color as an array [r, g, b] or a reference to a texture as a string and uses that as the skybox. Can only be called once, to update the skybox during runtime assign a texture to the skybox and update the texture instead.
setCameraPosition	Takes an array [x, y, z] and sets the camera's position to those coordinates.
setLightPosition	Takes an array [x, y, z] and sets the light's position to those coordinates.
lookAt	Takes an array [x, y, z] and sets the camera's rotation so that it looks at those coordinates.
onUpdate	Function for registering functions that will be called each frame. Takes in a function that takes the delta time and the frame time and returns nothing. Multiple functions can be registered by calling onUpdate multiple times.

Table 3.2: Table describing how to use the built in functions.

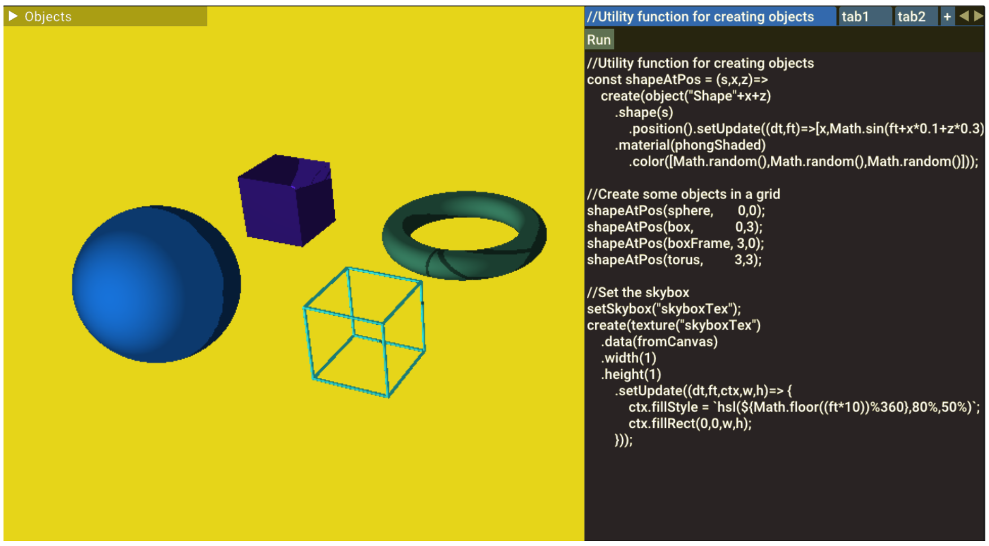
and passing in a JavaScript function to run every frame, that function has access to any computation a normal JavaScript function running in the browser would. This means that end users are free to use preexisting JavaScript libraries to manipulate the objects in the scene.

3.3 Usage

In this section we will be combining the concepts discussed in the previous two sections two create some scenes. That demonstrate some, but not all, of the capabilities of the language.

3.3.1 Multiple primitives

No special syntax is required to add multiple primitives in one scene, after finishing a call to create an object a second one can be created right after. When this is done the primitives in the scene are treated as a union of the functions. Fig 3.1 is an example showing 4 basic primitives in a single scene, the entirety of the code is visible on the right. A video of this scene is included as multiple-primitives.mp4.



```
//Utility function for creating objects
const shapeAtPos = (s,x,z)=>
   create(object("Shape"+x+z)
      .shape(s)
         .position().setUpdate((dt,ft)=>[x,Math.sin(ft+x*0.1+z*0.3)
      .material(phongShaded)
         .color([Math.random(),Math.random(),Math.random()]));

//Create some objects in a grid
shapeAtPos(sphere,     0,0);
shapeAtPos(box,        0,3);
shapeAtPos(boxFrame, 3,0);
shapeAtPos(torus,      3,3);

//Set the skybox
setSkybox("skyboxTex");
create(texture("skyboxTex")
   .data(fromCanvas)
   .width(1)
   .height(1)
      .setUpdate((dt,ft,ctx,w,h)=> {
         ctx.fillStyle = `hsl(${Math.floor((ft*10))%360},80%,50%)`;
         ctx.fillRect(0,0,w,h);
      }));
```

Figure 3.1: Multiple primitives in a scene.

3.3.2 Primitives occupying the same space

The environment is meant to be easy to use and so the default behavior should be what someone inexperienced with graphics would expect. So when placing multiple objects in the same space they should behave like real physical objects when they can. Like when the parts that intersect are empty interiors as can be seen with the box frames. When the solid parts intersect they will become a new object, forming the shape that is the union of both surfaces.

Figure 3.2 shows a screenshot from a scene where objects are inside each other. A video of this scene is included as primitives-inside.mp4 in the delivery and the full code can be found in listing C.2.

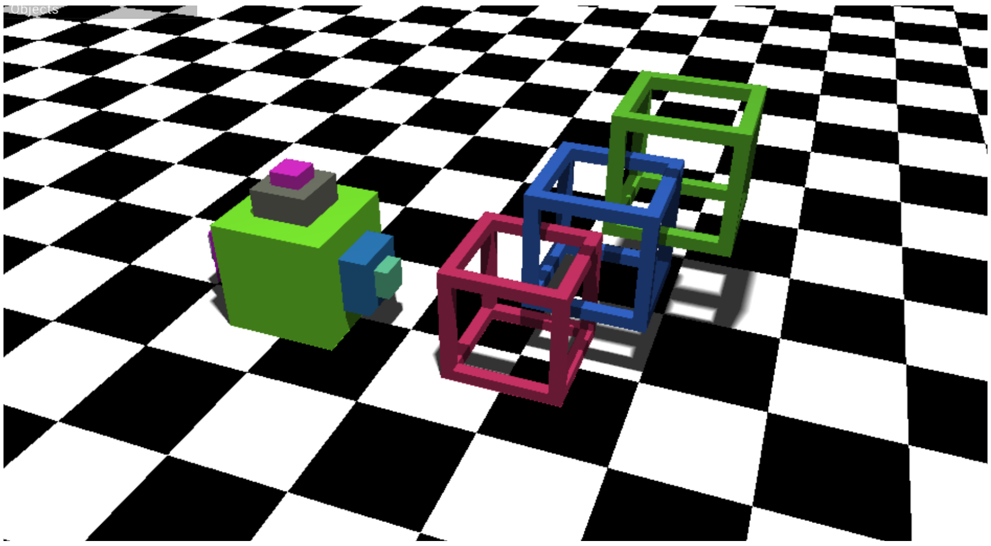


Figure 3.2: Primitives that intersect each other.

3.3.3 Primitives moving in a coordinated way

Using the scripting capabilities to set the position and other properties of the objects means it's easy to coordinate them. Figure 3.3 shows a still image from the video primitives-coordinated.mp4 that shows 3 different primitives moving and changing their properties in line with each other. The camera is also moved automatically using the *setCameraPosition* and *lookAt* functions. The code for this scene is included in listing C.3. The positions are coordinated such that the sphere-box

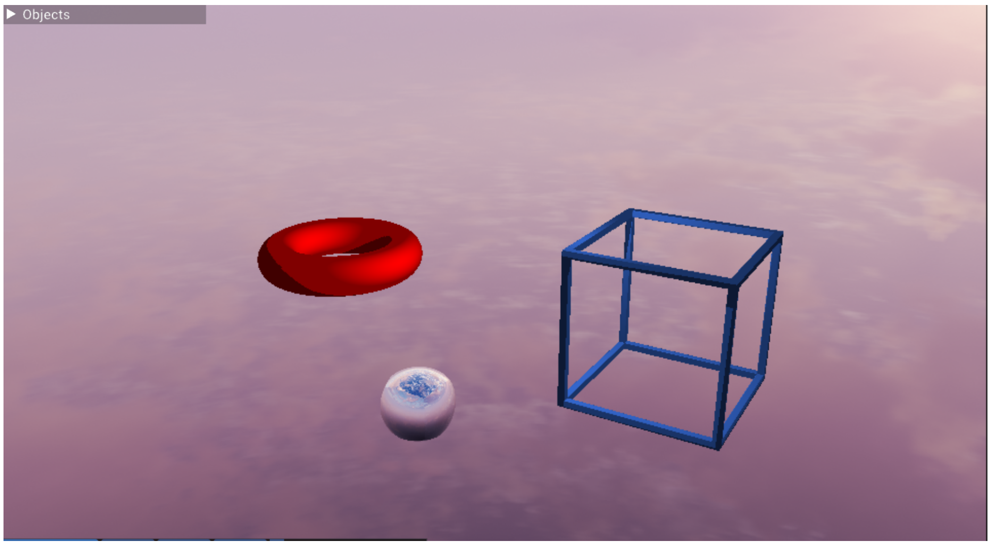


Figure 3.3: Primitives that move and change in a coordinated fashion.

moves through the hole of the torus. The size of the box frame is coordinated with the spheres movement so that it "clamps down" on it as it passes through. The weight property of the sphere-box is in line with its movement so that it is a sphere when passing the torus and a box when passing the frame.

3.3.4 Multiple textures from URLs

Adding textures via URLs is very easy to do as demonstrated in this scene, this is a feature of running in the web. In this scene a sphere is textured to look like the earth using 3 textures for the surface. An image of the earth as the diffuse texture, an image of earth at night as the shadow texture and an image with the worlds oceans colored white and the land colored black as the specular texture. There is also an image of the stars in the sky used as the skybox. Figure 3.4 shows an image of the scene.

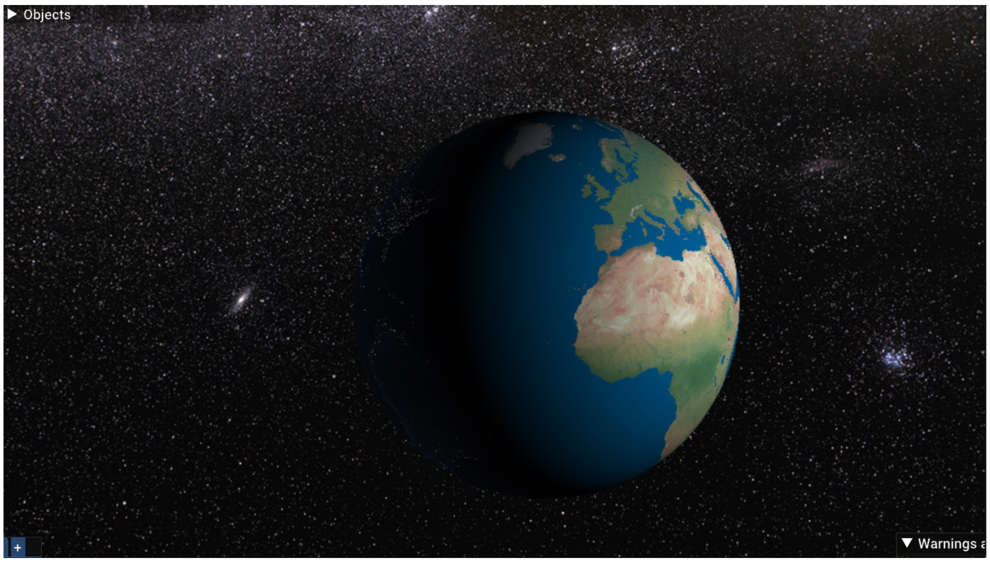


Figure 3.4: Scene where multiple textures fetched from the web are in use at once.

The full code can be found in listing C.4 and two videos of the scene are included in the delivery, texture-1.mp4 and texture-2.mp4.

3.3.5 Loading a JavaScript library

The fact that the scripting language is being run with a call to the JavaScript function *eval* means we have access to all of JavaScript, including *eval* itself. This makes the scripting environment incredibly flexible as we can now load any JavaScript library as plain text and use it in our environment.
In this example usage we load a Perlin noise [13] library, that is a library that generates smooth noise, that is being hosted as a plain text file. We then *eval* it and use it to texture a sphere. Figure 3.5 shows an image of the scene.

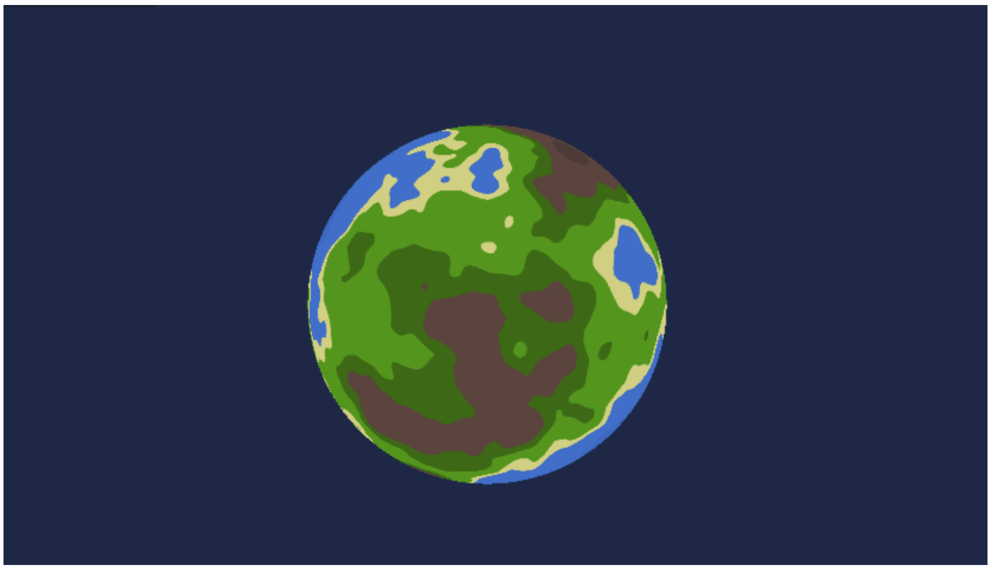


Figure 3.5: A texture created with a Perlin noise library loaded from the scripting environment.

The full code can be found in listing C.5 and a video of the scene is included in the delivery as load-library.mp4.

Chapter 4

Implementation

This section lists the main tools and technologies used in this project and explains for what purpose they were used.

4.1 Programming languages

There was three programming languages used in this project, TypeScript, JavaScript and GLSL. The main one used is TypeScript. This is what the main application is written in, the GLSL code generator, the scripting language and the renderer.

4.1.1 TypeScript

TypeScript is a language based on JavaScript [14], the main difference is that while regular JavaScript is dynamically typed and interpreted, TypeScript has a static type system and is compiled to JavaScript. This is a great advantage when it comes to working on larger projects as static typing lets you catch errors at compile time instead of runtime. It is also helpful when a project has multiple modules that does very different work but needs to work together like this one. This is because the static typing lets you define the interfaces that the modules will use to interact and it helps make sure you follow them.

The main TypeScript portion of the project is divided into 4 sections. The compiler, the language, the scene and the renderer. Fig 4.1 shows how this is structured. The *res* folder is just a general resource folder for the project, that's where fonts and test images are placed.

Compiler

The *compiler* part of the project is responsible for taking a scene with objects in it and compiling it to valid GLSL code with certain constraints. This generated code needs to have a data binding to the scene that is still being manipulated in JavaScript. This is done by passing the variable properties of the scene itself and

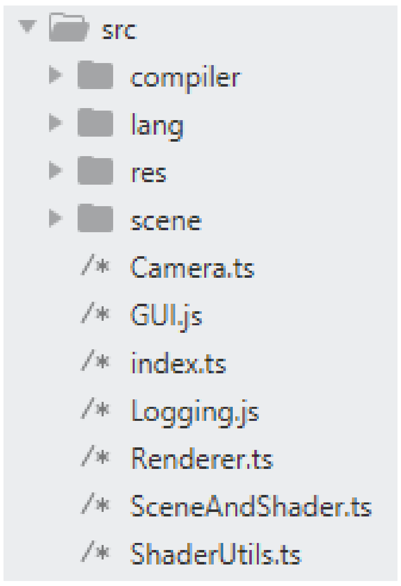


Figure 4.1: The structure of the project.

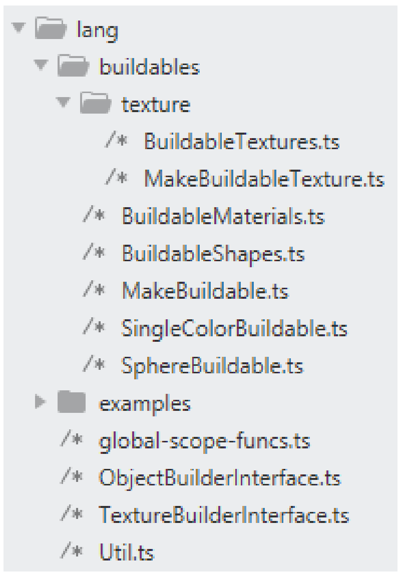
the objects in the scene to the shader via uniforms each frame.
A code example of how the compiler's `buildShader` function is used is given in listing 4.1. This function returns a string containing the correct fragment shader code for the scene. An example of this generated GLSL code can be fund in appendix B.

Code listing 4.1: Code example creating a shader from a scene.

```
//Turn the scene into a fragment shader.
const fragment = buildShader(scene, gl.canvas.width, gl.canvas.height);
const fragShader = compileShader(fragment, gl.FRAGMENT_SHADER, gl);
//Vertex shader always uses same shader code, no need to generate.
const vertShader = compileShader(vertex, gl.VERTEX_SHADER, gl);
//Create a program to use the shaders.
this.program = gl.createProgram();
//Attach and link.
gl.attachShader(this.program, vertShader);
gl.attachShader(this.program, fragShader);
gl.linkProgram(this.program);
```

Language

The language module deals with the actual scripting language the user uses. Fig 4.2 shows the content of the language module. The language works by using JavaScript's `eval` function with a context that has some extra functions.
`Eval` is a function in JavaScript that takes some string input and runs it, or evaluates it, as JavaScript code. The code execution is run using the enclosing scope. This means that we can give the script being ran access to variables and functions by declaring new aliases that points to these variables and functions in the enclos-



```
lang
  buildables
    texture
      BuildableTextures.ts
      MakeBuildableTexture.ts
    BuildableMaterials.ts
    BuildableShapes.ts
    MakeBuildable.ts
    SingleColorBuildable.ts
    SphereBuildable.ts
  examples
  global-scope-funcs.ts
  ObjectBuilderInterface.ts
  TextureBuilderInterface.ts
  Util.ts
```

Figure 4.2: The structure of the language module.

ing scope of the call to eval.
The functions and variables we give eval access to are what makes up the scripting language. The primitives, like sphere, are variables in the scope that actually points to builders that can build the correct GLSL code based on what properties the user wants active and the values of those properties. These builders are made in the `BuildableShapes.ts`, `BuildableTextures.ts` and `BuildableMaterials.ts` files.

Renderer

The renderer is in the top-level file `Renderer.ts` and it is responsible for actually rendering the scene. To render the scene it has to use the `buildShader` function to create the correct fragment shader for the scene as shown in listing 4.1. As can be seen from this code sample a reference to a WebGL context is also needed, so this needs to be passed to the renderer as well.
There are 3 steps to this process, each represented by a function on the renderer. The first step is the *compile* step, this function takes a WebGL context and a scene and creates a shader program representing the scene for the context. The next step is *attach*, this is where a target WebGL context is given. This target context is where the scene will be drawn. The final step is *update*, this updates the uniforms in the shader so that they have the latest data.
After these steps are complete the renderer's *draw* function can be called to actually run the shader, ray march the scene and output it the the specified target. This *update* $\longrightarrow$ *draw* step then happens once every frame for the entirety of the applications lifetime.

Scene

The *scene* is a class that holds the data describing the graphical objects to be rendered and functions to update them. Each scene has a camera, list of objects, a list of properties and a list of textures. All of these are used by other parts of the program in different ways. The language builds the scene and the objects, properties and textures to put in it, the compiler uses it to generate GLSL and the renderer uses it to update the shader and to draw the graphics described by the scene.

4.1.2 JavasScript

JavaScript is used in two places in the project. As a scripting language within the language and for controlling the GUI as that is loaded from an online source without type information.
The GUI code can be found in the file `GUI.js`. This file is responsible for taking a scene and rendering controls to modify the properties of the objects in the scene, a window showing errors or warnings and the script input tabs where the user can enter text. When the user clicks the run button this GUI calls a callback function that passes the script text back to the entry point of the application. From there the compiler and lang modules and the renderer is used to run the script, generate a scene and then render it.

4.1.3 GLSL

GLSL is a shading language developed by OpenGL ARB and now maintained by the Khronos group [15]. This project fundamentally revolves around generating GLSL code with certain features based on simpler constructs in a higher level language. This means that a lot of GLSL was written directly but most is generated at runtime.

4.2 Libraries

The software is mainly original code using no third party libraries or code, but for the user-facing GUI i decided to go with a simple framework to make the development process easier.

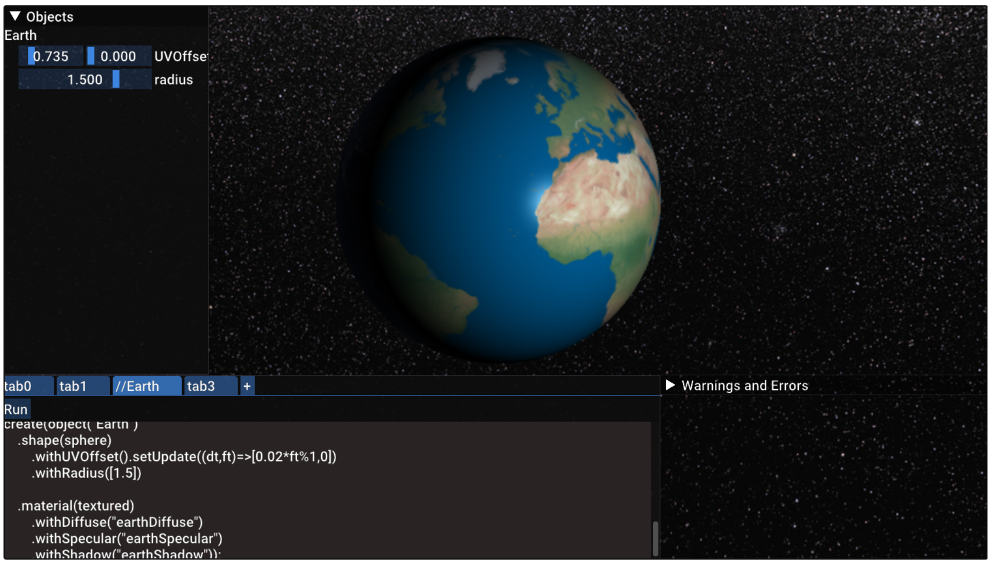
4.2.1 Dear ImGui js

Dear ImGui is a renderer agnostic immediate mode GUI library written in c++ [16]. *Immediate mode* refers to the style used when defining the GUI. The concept of an immediate mode UI vs retained UI is very big and blurry. A good intro can be found at the wiki article [17]. A short summary of how it differs from other GUI libraries is that it limints the amount of state we have to keep track of. Dear ImGui exposes very easy to use functions instead that takes care of managing state and polling for events etc. A code snipped from the projects GitHub showing how to create some text, a button and a slider in Dear ImGui is given in listing 4.2.

Code listing 4.2: Code example creating some GUI elements in Dear ImGui.

```
ImGui::Text("Hello,␣world␣%d", 123);
if (ImGui::Button("Save"))
    MySaveFunction();
ImGui::InputText("string", buf, IM_ARRAYSIZE(buf));
ImGui::SliderFloat("float", &f, 0.0f, 1.0f);
```

Dear ImGui js is a JavaScript port of the project that uses a WebGL backend. This was used to create the main GUI that the user interacts with. This consists of two main parts. The *script input* tabs and the *objects window*.



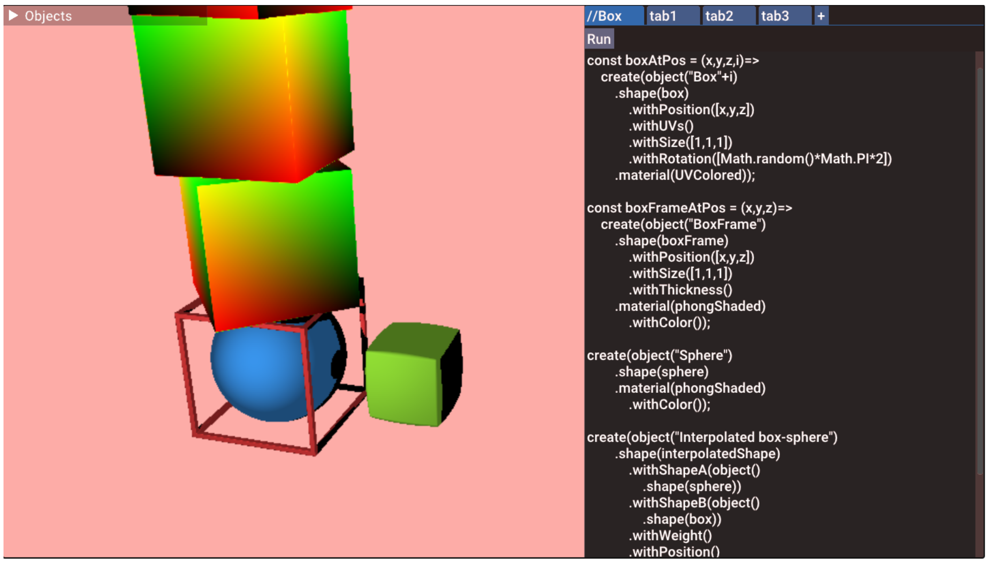
```
create(object("Earth")
  .shape(sphere)
    .withUVOffset().setUpdate((dt,ft)=>[0.02*ft%1,0])
    .withRadius([1.5])

  .material(textured)
    .withDiffuse("earthDiffuse")
    .withSpecular("earthSpecular")
    .withShadow("earthShadow"));
```

Figure 4.3: Main GUI

The *script input* is a set of tabbed text input fields where the user can enter scripts and a button to run them. The window is movable and in fig 4.4 it can be seen on the right side.

The *objects window* is a window showing all the objects in the scene and their properties that can be modified in real time. There are different GUI elements depending on the type of the variable for easy interaction. In fig 4.5 the color picker for the sphere object is visible and several other elements to adjust things like size and position can be seen as well. All of these GUI elements are standard ones that are part of the Dear ImGui js library.



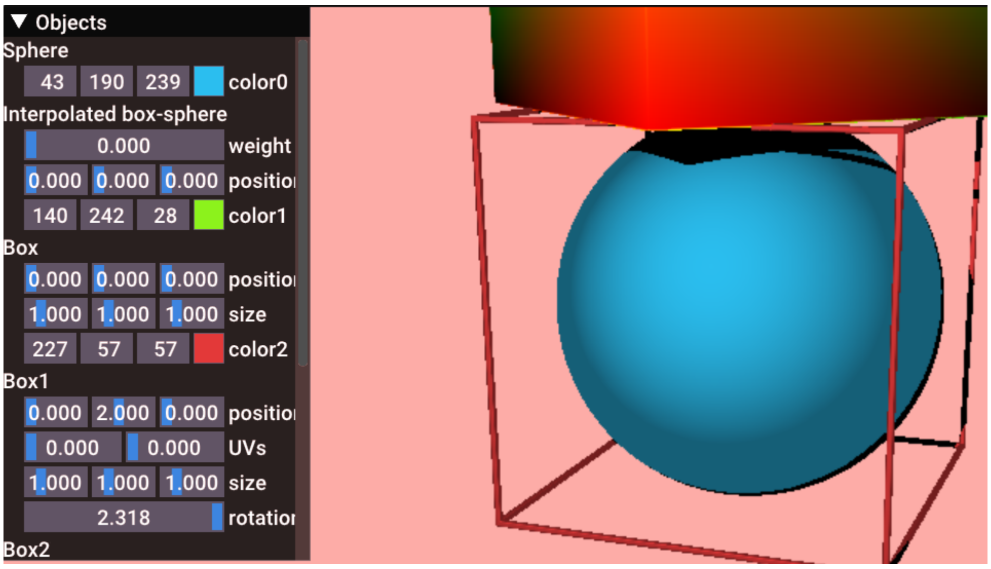
```
const boxAtPos = (x,y,z,i)=>
  create(object("Box"+i)
    .shape(box)
      .withPosition([x,y,z])
      .withUVs()
      .withSize([1,1,1])
      .withRotation([Math.random()*Math.PI*2])
    .material(UVColored));

const boxFrameAtPos = (x,y,z)=>
  create(object("BoxFrame")
    .shape(boxFrame)
      .withPosition([x,y,z])
      .withSize([1,1,1])
      .withThickness()
    .material(phongShaded)
      .withColor());

create(object("Sphere")
    .shape(sphere)
    .material(phongShaded)
      .withColor());

create(object("Interpolated box-sphere")
    .shape(interpolatedShape)
      .withShapeA(object()
        .shape(sphere))
      .withShapeB(object()
        .shape(box))
      .withWeight()
      .withPosition()
```

Figure 4.4: The script input window can be seen on the right.



▼ Objects

Sphere
43 190 239 🟦 color0

Interpolated box-sphere
0.000 weight
0.000 0.000 0.000 positio
140 242 28 🟩 color1

Box
0.000 0.000 0.000 positio
1.000 1.000 1.000 size
227 57 57 🟥 color2

Box1
0.000 2.000 0.000 positio
0.000 0.000 UVs
1.000 1.000 1.000 size
2.318 rotatio

Box2

Figure 4.5: The objects window is shown to the left.

4.3 Tools

4.3.1 Github

Github was used for version control during the development of the project. The final project source is hosted here.

4.3.2 Chrome

Chrome is the browser with the most support for needed WebGL features. In particular the shaders use GLSL version 300 which is a WebGL2 feature, this version of GLSL has different keywords from the old version 100. WegGL2 also allows more texture types in the shaders which is helpful during testing and development, it also has certain useful extensions by default. A list of useful additions in WebGL2 can be found here. Because of this Chrome was the main browser used for the entirety of the development, but other browsers has also been tested and the software also works on Firefox version 84 and Opera version 77.

Chapter 5

Deployment

5.1 Clone

The code for this project is open source and available here. You can clone it to you local machine either by using git and cloning the project via the URL on github or by navigating to the project in a browser and downloading a zip archive. This archive can then be unzip to a folder where you want to install and build the project.

5.2 Install

To install the software NPM is needed. Simply running

Code listing 5.1: NPM command to install dependencies

```
npm install
```

will install all necessary dependencies required for building the project. The project also depends on libraries hosted online that does not need to be installed to deploy.

5.3 Build

The project is written in TypeScript, JavaScript and some GLSL. The TypeScript portion has to be compiled to JavaScript before it can be run. This is done using the TypeScript compiler. Both Javascript and the GLSL code do not require any further modification before they can be run in a browser. After this compilation all the code has to be combined, or *bundled*, so that it can be referenced as a single file in HTML. This is all done using Webpack. Webpack is configured to run tsc on all referenced TypeScript files and include them in the bundle when running the command in listing 5.2.

Code listing 5.2: Running the build script with NPM from the command line

```
npm run build
```

After building you have to place the index.html file into the dist folder before you open it.

5.4 Run

To run the compiled project simply open the run script as shown in listing 5.3, this will start a server locally that only you can access.

Code listing 5.3: Running the start script with NPM from the command line

```
npm run start
```

Chapter 6

Evaluation and Discussion

6.1 Evaluation

The performance of the software is depended on the hardware and OS used as well as what version of Chrome so this evaluation is only for the specific hardware and software the evaluation was performed with. Changing any of these properties can affect the result.
The evaluation was performed on a laptop with an Intel Core i7-6700HQ at 2.60GHz CPU, 8 GB RAM and a Nvidia GeForce GTX 960M GPU. The OS was Windows 10 Home build 19041 and the browser was Chrome version 91.

Interactivity

The software easily manages to stay at withing the frame times required for 30 fps when showing the least intensive scene, a scene with only a sphere in it. As more objects are added however the fps will decrease, how many objects that can be added will depend on how powerful the hardware running the software is and how complex the objects are.
If we keep the objects as simple as possible and give them no properties the compiler will have a much easier job and we can get to 252 objects before we run into `Error: expression too complex`. Using these objects without any properties we are still staying within our frame time budget of 33.34ms. The script for this scene is in listing 6.1 and the GLSL generated is included in the file 252-spheres-glsl.txt. This however is not a very interesting scene, as there are no properties to change so the spheres are completely static and in the same spot.

Code listing 6.1: Code for creating 252 spheres.

```
const numberOfSpheres = 252;
for(let i = 0; i < numberOfSpheres; i++) {
  create(object(""+i)
    .shape(sphere)
    .material(singleColored));
}
```

If we then add a single property to the spheres, position in this case, the number changes to about 40 before rendering starts taking too long. It should be noted that there are other factors affecting the frame rate too, if we move the camera close to a single object so that it covers the entire screen that will of course dramatically increase the performance as way fewer ray marching steps has to be taken for each pixel. The code for the scene can be found in listing 6.2 and the full GLSL is in the file 40-spheres-glsl.txt.

Code listing 6.2: Code for creating 40 spheres with a position property initialized to a random position.

```
const numberOfSpheres = 40;
for(let i = 0; i < numberOfSpheres; i++) {
  create(object(""+i)
    .shape(sphere).position([Math.random()*10-5,
                             Math.random()*10-5,
                             Math.random()*10-5])
    .material(singleColored));
}
```

The next thing to look at is the impact on the frame rate of manipulating these properties. Listing 6.3 shows the code to create the scene, it gives each sphere a random start position and over time moves them into the center. This scene could only handle about 20 spheres before the frame rate dipped slightly below 30 fps, however since the spheres are very spread out at the start and then they all move to the same spot the effects of the complexity of the scene on the frame rate is noticeable too. When the spheres are very close together the frame rate increases as the number of ray marching steps required decreases.

Code listing 6.3: Code for creating 20 spheres with a position property being updated.

```
const numberOfSpheres = 20;
for(let i = 0; i < numberOfSpheres; i++) {
  const start = [Math.random()*10-5,Math.random()*10-5,Math.random()*10-5];
  create(object(""+i)
    .shape(sphere)
      .position()
        .setUpdate((dt,ft)=>[start[0]*(Math.cos(ft)*0.5+0.5),
                             start[1]*(Math.cos(ft)*0.5+0.5),
                             start[2]*(Math.cos(ft)*0.5+0.5)])
    .material(singleColored));
}
```

6.2 Discussion

How did the process work

Being that the project focused on doing something new in the space of graphics it required a good amount of research at the start of the project. This also meant that my vision of the end product changed a good bit during this research and experimentation phase of the project. However the core problem statement and what

I wanted to achieve did not change much, it was mostly the *how* that changed underway as I learned more.
As an example at the start of the project I tried to make a system where I would create GLSL and corresponding JavaScript so that the ray marching could be done both on the GPU and on the CPU. This would give the benefit of easy random access into the scene. This access could be used to automatically generate bounding boxes for the scene or to allow users to click objects to select them. This turned out to be a bit difficult and it ended up being too time consuming to implement with how I set the language up.

Choices made

A choice that was made early on was to target WebGL, I think this was a very good decision, but it did cause some problems later on. One major problem being that using WebGL gives you less control over the GPU vs using a native API and making a normal desktop application. This led to me running into the issue of long compile times early on, as modern browsers will crash if you take too long to compile. This issue is compounded by two facts of WebGL in a browser. One is that shader source has to be compiled on the users computer, you are not allowed to run pre-compiled code sent over the web on the GPU. Second is that WebGL does not support linking to other shaders, making it *almost* impossible to split the code into multiple smaller compilation units.
I say almost impossible because with WebGL 2 we now have depth buffers which means that if the bottleneck is shader compilation and not shader runtime a solution could be to split the shader into multiple shaders, each with their own ray marching loop and their own set of objects. These would then write to the depth buffer and use that to determine which shader's object should be drawn when they overlap. This would increase runtime as multiple draw calls would be needed each frame instead of just one.

An important design decision I made was how granular to make the language. At the one end of the spectrum I could have made something that was essentially just a different dialect of GLSL but equally low level. On the other end of the spectrum is what I ended up doing, which is a very high level language where the user only has access to pre-defined primitives that each represents multiple lines of GLSL that the user does not have granular control over. It should be mentioned here that via the JavaScript scripting it is technically possible for users to add their own objects with their own GLSL code, but this is not very user friendly as then all the guard rails of the higher level primitives are gone. A better solution might have been to let the user somehow voluntarily step down this spectrum at will, to obtain a slightly more granular control while still being in the easy-to-use beginner-friendly scripting environment. I did not figure out how to achieve this.

What I would have done differently

If I were to do this same project once more I would spend more time on making the language more powerful to allow users to essentially write, a subset of, GLSL in an easier more high level way. These new language features would still have to support the out of the box modifiability and interactivity that the current system has. With a more complex language I would also look into writing a parser for it instead of relying on the browser's JavaScript engine. This would also make the goal of translating the code to both shader code and JavaScript easier.

Chapter 7

Conclusion

In this thesis project a web based software for creating and interacting with ray marched graphics was created. This was done by creating a scripting environment and a new high level scripting language that compiles to GLSL so that it would be easier for beginners to use. In this scripting environment a easy to use GUI was included to interact with the properties of the objects of the scene in an intuitive way.
It seems clear from the examples given in this thesis that this is an easier and more accessible way to work with ray marched graphics and in that sense the software fulfills its goals mentioned in the introduction.

To further develop this software the next steps would be to expand the capabilities of the scripting language by adding more primitives and materials, but also by adding more granular control to make it easier to add new GLSL to the environment.

Bibliography

[1] E. Freniere and J. Tourtellott, 'Brief history of generalized ray tracing,' in *Optics & Photonics*, 1997.

[2] V. Sanzharov, V. Frolov and V. Galaktionov, 'Survey of nvidia rtx technology,' *Programming and Computer Software*, vol. 46, pp. 297–304, 2020.

[3] T. Davidovič, T. Engelhardt, I. Georgiev, P. Slusallek and C. Dachsbacher, '3d rasterization: A bridge between rasterization and ray casting,' in *Proceedings of Graphics Interface 2012*, 2012, pp. 201–208.

[4] E. Haines, J. Günther and T. Akenine-Möller, 'Precision improvements for ray/sphere intersection,' 2019.

[5] J. C. Hart, 'Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces,' *The Visual Computer*, vol. 12, no. 10, pp. 527–545, 1996.

[6] D. Bachman, 'Advanced calculus demystified,' 2007.

[7] Z. Majercik, J.-P. Guertin, D. Nowrouzezahrai and M. McGuire, 'Dynamic diffuse global illumination with ray-traced irradiance fields,' *Journal of Computer Graphics Techniques Vol*, vol. 8, no. 2, 2019.

[8] I. Quilez and P. Jeremias. (). 'Shadertoy beta,' [Online]. Available: `https://www.shadertoy.com` (visited on 10/07/2021).

[9] I. Quilez and P. Jeremias. (). 'Browse - shadertoy beta,' [Online]. Available: `https://www.shadertoy.com/results?query=&sort=popular&filter=` (visited on 10/07/2021).

[10] iq. (). 'Raymarching - primitives,' [Online]. Available: `https://www.shadertoy.com/view/Xds3zN` (visited on 10/07/2021).

[11] K. Schwaber and M. A. Beedle, 'Agile software development with scrum,' 2001.

[12] B. T. Phong, 'Illumination for computer generated pictures,' *Communications of the ACM*, vol. 18, pp. 311–317, 1975.

[13] K. Perlin, 'Improving noise,' *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, 2002.

[14] (). 'Typescript for the new programmer,' [Online]. Available: https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html (visited on 28/07/2021).

[15] (). 'Opengl overview,' [Online]. Available: https://www.khronos.org/opengl/ (visited on 28/07/2021).

[16] (). 'Dear imgui,' [Online]. Available: https://github.com/ocornut/imgui (visited on 28/07/2021).

[17] (). 'About the imgui paradigm,' [Online]. Available: https://github.com/ocornut/imgui/wiki/About-the-IMGUI-paradigm (visited on 28/07/2021).

Appendix A

Appendix A

Code listing A.1: Shadercode from shadertoy.com

```
// The MIT License
// Copyright  2013 Inigo Quilez
/* Permission is hereby granted, free of charge, to any person obtaining a copy of
this software and associated documentation files (the "Software"), to deal in the
Software without restriction, including without limitation the rights to use,
copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the
Software, and to permit persons to whom the Software is furnished to do so,
subject to the following conditions: The above copyright notice and this
permission notice shall be included in all copies or substantial portions of the
Software. THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. */

// A list of useful distance function to simple primitives. All
// these functions (except for ellipsoid) return an exact
// euclidean distance, meaning they produce a better SDF than
// what you'd get if you were constructing them from boolean
// operations.

// List of other 3D SDFs: https://www.shadertoy.com/playlist/43cXRl
//
// and http://iquilezles.org/www/articles/distfunctions/distfunctions.htm


#if HW_PERFORMANCE==0
#define AA 1
#else
#define AA 2   // make this 2 or 3 for antialiasing
#endif

//------------------------------------------------------------------
float dot2( in vec2 v ) { return dot(v,v); }
float dot2( in vec3 v ) { return dot(v,v); }
float ndot( in vec2 a, in vec2 b ) { return a.x*b.x - a.y*b.y; }

float sdPlane( vec3 p )
{
  return p.y;
}

float sdSphere( vec3 p, float s )
{
    return length(p)-s;
}

float sdBox( vec3 p, vec3 b )
{
    vec3 d = abs(p) - b;
    return min(max(d.x,max(d.y,d.z)),0.0) + length(max(d,0.0));
}

float sdBoundingBox( vec3 p, vec3 b, float e )
{
       p = abs(p  )-b;
  vec3 q = abs(p+e)-e;

  return min(min(
      length(max(vec3(p.x,q.y,q.z),0.0))+min(max(p.x,max(q.y,q.z)),0.0),
      length(max(vec3(q.x,p.y,q.z),0.0))+min(max(q.x,max(p.y,q.z)),0.0)),
      length(max(vec3(q.x,q.y,p.z),0.0))+min(max(q.x,max(q.y,p.z)),0.0));
}
float sdEllipsoid( in vec3 p, in vec3 r ) // approximated
{
    float k0 = length(p/r);
    float k1 = length(p/(r*r));
    return k0*(k0-1.0)/k1;
}

float sdTorus( vec3 p, vec2 t )
{
    return length( vec2(length(p.xz)-t.x,p.y) )-t.y;
}

float sdCappedTorus(in vec3 p, in vec2 sc, in float ra, in float rb)
{
    p.x = abs(p.x);
    float k = (sc.y*p.x>sc.x*p.y) ? dot(p.xy,sc) : length(p.xy);
    return sqrt( dot(p,p) + ra*ra - 2.0*ra*k ) - rb;
}

float sdHexPrism( vec3 p, vec2 h )
{
    vec3 q = abs(p);

    const vec3 k = vec3(-0.8660254, 0.5, 0.57735);
    p = abs(p);
    p.xy -= 2.0*min(dot(k.xy, p.xy), 0.0)*k.xy;
    vec2 d = vec2(
       length(p.xy - vec2(clamp(p.x, -k.z*h.x, k.z*h.x), h.x))*sign(p.y - h.x),
       p.z-h.y );
    return min(max(d.x,d.y),0.0) + length(max(d,0.0));
}

float sdOctogonPrism( in vec3 p, in float r, float h )
{
  const vec3 k = vec3(-0.9238795325,   // sqrt(2+sqrt(2))/2
                      0.3826834323,   // sqrt(2-sqrt(2))/2
```

```
                        0.4142135623 ); // sqrt(2)-1
  // reflections
  p = abs(p);
  p.xy -= 2.0*min(dot(vec2( k.x,k.y),p.xy),0.0)*vec2( k.x,k.y);
  p.xy -= 2.0*min(dot(vec2(-k.x,k.y),p.xy),0.0)*vec2(-k.x,k.y);
  // polygon side
  p.xy -= vec2(clamp(p.x, -k.z*r, k.z*r), r);
  vec2 d = vec2( length(p.xy)*sign(p.y), p.z-h );
  return min(max(d.x,d.y),0.0) + length(max(d,0.0));
}

float sdCapsule( vec3 p, vec3 a, vec3 b, float r )
{
  vec3 pa = p-a, ba = b-a;
  float h = clamp( dot(pa,ba)/dot(ba,ba), 0.0, 1.0 );
  return length( pa - ba*h ) - r;
}

float sdRoundCone( in vec3 p, in float r1, float r2, float h )
{
    vec2 q = vec2( length(p.xz), p.y );

    float b = (r1-r2)/h;
    float a = sqrt(1.0-b*b);
    float k = dot(q,vec2(-b,a));

    if( k < 0.0 ) return length(q) - r1;
    if( k > a*h ) return length(q-vec2(0.0,h)) - r2;

    return dot(q, vec2(a,b) ) - r1;
}

float sdRoundCone(vec3 p, vec3 a, vec3 b, float r1, float r2)
{
    // sampling independent computations (only depend on shape)
    vec3  ba = b - a;
    float l2 = dot(ba,ba);
    float rr = r1 - r2;
    float a2 = l2 - rr*rr;
    float il2 = 1.0/l2;

    // sampling dependant computations
    vec3 pa = p - a;
    float y = dot(pa,ba);
    float z = y - l2;
    float x2 = dot2( pa*l2 - ba*y );
    float y2 = y*y*l2;
    float z2 = z*z*l2;

    // single square root!
    float k = sign(rr)*rr*rr*x2;
    if( sign(z)*a2*z2 > k ) return  sqrt(x2 + z2)        *il2 - r2;
    if( sign(y)*a2*y2 < k ) return  sqrt(x2 + y2)        *il2 - r1;
                            return (sqrt(x2*a2*il2)+y*rr)*il2 - r1;
}

float sdTriPrism( vec3 p, vec2 h )
{
    const float k = sqrt(3.0);
    h.x *= 0.5*k;
    p.xy /= h.x;
    p.x = abs(p.x) - 1.0;
    p.y = p.y + 1.0/k;
    if( p.x+k*p.y>0.0 ) p.xy=vec2(p.x-k*p.y,-k*p.x-p.y)/2.0;
    p.x -= clamp( p.x, -2.0, 0.0 );
    float d1 = length(p.xy)*sign(-p.y)*h.x;
    float d2 = abs(p.z)-h.y;
    return length(max(vec2(d1,d2),0.0)) + min(max(d1,d2), 0.);
}

// vertical
float sdCylinder( vec3 p, vec2 h )
{
    vec2 d = abs(vec2(length(p.xz),p.y)) - h;
    return min(max(d.x,d.y),0.0) + length(max(d,0.0));
}

// arbitrary orientation
float sdCylinder(vec3 p, vec3 a, vec3 b, float r)
{
    vec3 pa = p - a;
    vec3 ba = b - a;
    float baba = dot(ba,ba);
    float paba = dot(pa,ba);

    float x = length(pa*baba-ba*paba) - r*baba;
    float y = abs(paba-baba*0.5)-baba*0.5;
    float x2 = x*x;
    float y2 = y*y*baba;
    float d = (max(x,y)<0.0)?-min(x2,y2):(((x>0.0)?x2:0.0)+((y>0.0)?y2:0.0));
    return sign(d)*sqrt(abs(d))/baba;
}

// vertical
float sdCone( in vec3 p, in vec2 c, float h )
{
    vec2 q = h*vec2(c.x,-c.y)/c.y;
    vec2 w = vec2( length(p.xz), p.y );

  vec2 a = w - q*clamp( dot(w,q)/dot(q,q), 0.0, 1.0 );
    vec2 b = w - q*vec2( clamp( w.x/q.x, 0.0, 1.0 ), 1.0 );
    float k = sign( q.y );
    float d = min(dot( a, a ),dot(b, b));
    float s = max( k*(w.x*q.y-w.y*q.x),k*(w.y-q.y)  );
  return sqrt(d)*sign(s);
}

float sdCappedCone( in vec3 p, in float h, in float r1, in float r2 )
{
    vec2 q = vec2( length(p.xz), p.y );

    vec2 k1 = vec2(r2,h);
    vec2 k2 = vec2(r2-r1,2.0*h);
    vec2 ca = vec2(q.x-min(q.x,(q.y < 0.0)?r1:r2), abs(q.y)-h);
    vec2 cb = q - k1 + k2*clamp( dot(k1-q,k2)/dot2(k2), 0.0, 1.0 );
    float s = (cb.x < 0.0 && ca.y < 0.0) ? -1.0 : 1.0;
    return s*sqrt( min(dot2(ca),dot2(cb)) );
}

float sdCappedCone(vec3 p, vec3 a, vec3 b, float ra, float rb)
```

```
{
    float rba  = rb-ra;
    float baba = dot(b-a,b-a);
    float papa = dot(p-a,p-a);
    float paba = dot(p-a,b-a)/baba;

    float x = sqrt( papa - paba*paba*baba );

    float cax = max(0.0,x-((paba<0.5)?ra:rb));
    float cay = abs(paba-0.5)-0.5;

    float k = rba*rba + baba;
    float f = clamp( (rba*(x-ra)+paba*baba)/k, 0.0, 1.0 );

    float cbx = x-ra - f*rba;
    float cby = paba - f;

    float s = (cbx < 0.0 && cay < 0.0) ? -1.0 : 1.0;

    return s*sqrt( min(cax*cax + cay*cay*baba,
                       cbx*cbx + cby*cby*baba) );
}

// c is the sin/cos of the desired cone angle
float sdSolidAngle(vec3 pos, vec2 c, float ra)
{
    vec2 p = vec2( length(pos.xz), pos.y );
    float l = length(p) - ra;
  float m = length(p - c*clamp(dot(p,c),0.0,ra) );
    return max(l,m*sign(c.y*p.x-c.x*p.y));
}

float sdOctahedron(vec3 p, float s)
{
    p = abs(p);
    float m = p.x + p.y + p.z - s;

    // exact distance
    #if 0
    vec3 o = min(3.0*p - m, 0.0);
    o = max(6.0*p - m*2.0 - o*3.0 + (o.x+o.y+o.z), 0.0);
    return length(p - s*o/(o.x+o.y+o.z));
    #endif

    // exact distance
    #if 1
  vec3 q;
         if( 3.0*p.x < m ) q = p.xyz;
    else if( 3.0*p.y < m ) q = p.yzx;
    else if( 3.0*p.z < m ) q = p.zxy;
    else return m*0.57735027;
    float k = clamp(0.5*(q.z-q.y+s),0.0,s);
    return length(vec3(q.x,q.y-s+k,q.z-k));
    #endif

    // bound, not exact
    #if 0
  return m*0.57735027;
    #endif
}
```

```

float sdPyramid( in vec3 p, in float h )
{
    float m2 = h*h + 0.25;

    // symmetry
    p.xz = abs(p.xz);
    p.xz = (p.z>p.x) ? p.zx : p.xz;
    p.xz -= 0.5;

    // project into face plane (2D)
    vec3 q = vec3( p.z, h*p.y - 0.5*p.x, h*p.x + 0.5*p.y);

    float s = max(-q.x,0.0);
    float t = clamp( (q.y-0.5*p.z)/(m2+0.25), 0.0, 1.0 );

    float a = m2*(q.x+s)*(q.x+s) + q.y*q.y;
  float b = m2*(q.x+0.5*t)*(q.x+0.5*t) + (q.y-m2*t)*(q.y-m2*t);

    float d2 = min(q.y,-q.x*m2-q.y*0.5) > 0.0 ? 0.0 : min(a,b);

    // recover 3D and scale, and add sign
    return sqrt( (d2+q.z*q.z)/m2 ) * sign(max(q.z,-p.y));;
}

// la,lb=semi axis, h=height, ra=corner
float sdRhombus(vec3 p, float la, float lb, float h, float ra)
{
    p = abs(p);
    vec2 b = vec2(la,lb);
    float f = clamp( (ndot(b,b-2.0*p.xz))/dot(b,b), -1.0, 1.0 );
  vec2 q = vec2(length(p.xz-0.5*b*vec2(1.0-f,1.0+f))*sign(p.x*b.y+p.z*b.x-b.x*b.y)-
      ra, p.y-h);
    return min(max(q.x,q.y),0.0) + length(max(q,0.0));
}

//------------------------------------------------------------------

vec2 opU( vec2 d1, vec2 d2 )
{
  return (d1.x<d2.x) ? d1 : d2;
}

//------------------------------------------------------------------

#define ZERO (min(iFrame,0))

//------------------------------------------------------------------

vec2 map( in vec3 pos )
{
    vec2 res = vec2( 1e10, 0.0 );

    {
      res = opU( res, vec2( sdSphere(    pos-vec3(-2.0,0.25, 0.0), 0.25 ), 26.9 ) )
          ;
    }

    // bounding box
    if( sdBox( pos-vec3(0.0,0.3,-1.0),vec3(0.35,0.3,2.5) )<res.x )
    {
    // more primitives
    res = opU( res, vec2( sdBoundingBox( pos-vec3( 0.0,0.25, 0.0), vec3(0.3,0.25,0.2), 0.025 ), 16.9 ) );
  res = opU( res, vec2( sdTorus(      (pos-vec3( 0.0,0.30, 1.0)).xzy, vec2(0.25,0.05) ), 25.0 ) );
  res = opU( res, vec2( sdCone(        pos-vec3( 0.0,0.45,-1.0), vec2(0.6,0.8),0.45 ), 55.0 ) );
    res = opU( res, vec2( sdCappedCone(  pos-vec3( 0.0,0.25,-2.0), 0.25, 0.25, 0.1 ), 13.67 ) );
    res = opU( res, vec2( sdSolidAngle(  pos-vec3( 0.0,0.00,-3.0), vec2(3,4)/5.0, 0.4 ), 49.13 ) );
    }

    // bounding box
    if( sdBox( pos-vec3(1.0,0.3,-1.0),vec3(0.35,0.3,2.5) )<res.x )
    {
    // more primitives
  res = opU( res, vec2( sdCappedTorus((pos-vec3( 1.0,0.30, 1.0))*vec3(1,-1,1), vec2(0.866025,-0.5), 0.25, 0.05), 8.5) );
    res = opU( res, vec2( sdBox(         pos-vec3( 1.0,0.25, 0.0), vec3(0.3,0.25,0.1) ), 3.0 ) );
    res = opU( res, vec2( sdCapsule(     pos-vec3( 1.0,0.00,-1.0),vec3(-0.1,0.1,-0.1), vec3(0.2,0.4,0.2), 0.1  ), 31.9 ) );
  res = opU( res, vec2( sdCylinder(    pos-vec3( 1.0,0.25,-2.0), vec2(0.15,0.25) ), 8.0 ) );
    res = opU( res, vec2( sdHexPrism(    pos-vec3( 1.0,0.2,-3.0), vec2(0.2,0.05) ), 18.4 ) );
    }

    // bounding box
    if( sdBox( pos-vec3(-1.0,0.35,-1.0),vec3(0.35,0.35,2.5))<res.x )
    {
    // more primitives
  res = opU( res, vec2( sdPyramid(    pos-vec3(-1.0,-0.6,-3.0), 1.0 ), 13.56 ) );
  res = opU( res, vec2( sdOctahedron( pos-vec3(-1.0,0.15,-2.0), 0.35 ), 23.56 ) );
    res = opU( res, vec2( sdTriPrism(   pos-vec3(-1.0,0.15,-1.0), vec2(0.3,0.05) ),43.5 ) );
    res = opU( res, vec2( sdEllipsoid(  pos-vec3(-1.0,0.25, 0.0), vec3(0.2, 0.25, 0.05) ), 43.17 ) );
  res = opU( res, vec2( sdRhombus(   (pos-vec3(-1.0,0.34, 1.0)).xzy, 0.15, 0.25, 0.04, 0.08 ),17.0 ) );
    }

    // bounding box
    if( sdBox( pos-vec3(2.0,0.3,-1.0),vec3(0.35,0.3,2.5) )<res.x )
    {
    // more primitives
    res = opU( res, vec2( sdOctogonPrism(pos-vec3( 2.0,0.2,-3.0), 0.2, 0.05), 51.8 ) );
    res = opU( res, vec2( sdCylinder(    pos-vec3( 2.0,0.15,-2.0), vec3(0.1,-0.1,0.0), vec3(-0.2,0.35,0.1), 0.08), 31.2 ) );
  res = opU( res, vec2( sdCappedCone(  pos-vec3( 2.0,0.10,-1.0), vec3(0.1,0.0,0.0), vec3(-0.2,0.40,0.1), 0.15, 0.05), 46.1 ) );
    res = opU( res, vec2( sdRoundCone(   pos-vec3( 2.0,0.15, 0.0), vec3(0.1,0.0,0.0), vec3(-0.1,0.35,0.1), 0.15, 0.05), 51.7 ) );
    res = opU( res, vec2( sdRoundCone(   pos-vec3( 2.0,0.20, 1.0), 0.2, 0.1, 0.3 ), 37.0 ) );
    }

    return res;
}

// http://iquilezles.org/www/articles/boxfunctions/boxfunctions.htm
vec2 iBox( in vec3 ro, in vec3 rd, in vec3 rad )
{
    vec3 m = 1.0/rd;
    vec3 n = m*ro;
    vec3 k = abs(m)*rad;
    vec3 t1 = -n - k;
    vec3 t2 = -n + k;
  return vec2( max( max( t1.x, t1.y ), t1.z ),
               min( min( t2.x, t2.y ), t2.z ) );
}

vec2 raycast( in vec3 ro, in vec3 rd )
{
    vec2 res = vec2(-1.0,-1.0);

    float tmin = 1.0;
    float tmax = 20.0;

    // raytrace floor plane
    float tp1 = (0.0-ro.y)/rd.y;
    if( tp1>0.0 )
    {
        tmax = min( tmax, tp1 );
        res = vec2( tp1, 1.0 );
    }
    //else return res;

    // raymarch primitives
    vec2 tb = iBox( ro-vec3(0.0,0.4,-0.5), rd, vec3(2.5,0.41,3.0) );
    if( tb.x<tb.y && tb.y>0.0 && tb.x<tmax)
    {
        //return vec2(tb.x,2.0);
        tmin = max(tb.x,tmin);
        tmax = min(tb.y,tmax);

        float t = tmin;
        for( int i=0; i<70 && t<tmax; i++ )
        {
            vec2 h = map( ro+rd*t );
            if( abs(h.x)<(0.0001*t) )
            {
                res = vec2(t,h.y);
                break;
            }
            t += h.x;
        }
    }

    return res;
}

// http://iquilezles.org/www/articles/rmshadows/rmshadows.htm
float calcSoftshadow( in vec3 ro, in vec3 rd, in float mint, in float tmax )
{
    // bounding volume
    float tp = (0.8-ro.y)/rd.y; if( tp>0.0 ) tmax = min( tmax, tp );

    float res = 1.0;
    float t = mint;
    for( int i=ZERO; i<24; i++ )
    {
    float h = map( ro + rd*t ).x;
        float s = clamp(8.0*h/t,0.0,1.0);
        res = min( res, s*s*(3.0-2.0*s) );
        t += clamp( h, 0.02, 0.2 );
        if( res<0.004 || t>tmax ) break;
    }
    return clamp( res, 0.0, 1.0 );
}

// http://iquilezles.org/www/articles/normalsSDF/normalsSDF.htm
vec3 calcNormal( in vec3 pos )
{
#if 0
    vec2 e = vec2(1.0,-1.0)*0.5773*0.0005;
    return normalize( e.xyy*map( pos + e.xyy ).x +
            e.yyx*map( pos + e.yyx ).x +
            e.yxy*map( pos + e.yxy ).x +
            e.xxx*map( pos + e.xxx ).x );
#else
    // inspired by tdhooper and klems - a way to prevent the compiler from inlining
         map() 4 times
    vec3 n = vec3(0.0);
    for( int i=ZERO; i<4; i++ )
    {
        vec3 e = 0.5773*(2.0*vec3((((i+3)>>1)&1),((i>>1)&1),(i&1))-1.0);
        n += e*map(pos+0.0005*e).x;
      //if( n.x+n.y+n.z>100.0 ) break;
    }
    return normalize(n);
#endif
}

float calcAO( in vec3 pos, in vec3 nor )
{
  float occ = 0.0;
    float sca = 1.0;
    for( int i=ZERO; i<5; i++ )
    {
        float h = 0.01 + 0.12*float(i)/4.0;
        float d = map( pos + h*nor ).x;
        occ += (h-d)*sca;
        sca *= 0.95;
        if( occ>0.35 ) break;
    }
    return clamp( 1.0 - 3.0*occ, 0.0, 1.0 ) * (0.5+0.5*nor.y);
}

// http://iquilezles.org/www/articles/checkerfiltering/checkerfiltering.htm
float checkersGradBox( in vec2 p, in vec2 dpdx, in vec2 dpdy )
{
    // filter kernel
    vec2 w = abs(dpdx)+abs(dpdy) + 0.001;
    // analytical integral (box filter)
    vec2 i = 2.0*(abs(fract((p-0.5*w)*0.5)-0.5)-abs(fract((p+0.5*w)*0.5)-0.5))/w;
    // xor pattern
    return 0.5 - 0.5*i.x*i.y;
}

vec3 render( in vec3 ro, in vec3 rd, in vec3 rdx, in vec3 rdy )
{
    // background
    vec3 col = vec3(0.7, 0.7, 0.9) - max(rd.y,0.0)*0.3;

    // raycast scene
    vec2 res = raycast(ro,rd);
    float t = res.x;
  float m = res.y;
    if( m>-0.5 )
    {
        vec3 pos = ro + t*rd;
        vec3 nor = (m<1.5) ? vec3(0.0,1.0,0.0) : calcNormal( pos );
        vec3 ref = reflect( rd, nor );

        // material
        col = 0.2 + 0.2*sin( m*2.0 + vec3(0.0,1.0,2.0) );
        float ks = 1.0;

        if( m<1.5 )
        {
            // project pixel footprint into the plane
            vec3 dpdx = ro.y*(rd/rd.y-rdx/rdx.y);
            vec3 dpdy = ro.y*(rd/rd.y-rdy/rdy.y);

            float f = checkersGradBox( 3.0*pos.xz, 3.0*dpdx.xz, 3.0*dpdy.xz );
            col = 0.15 + f*vec3(0.05);
            ks = 0.4;
        }

        // lighting
        float occ = calcAO( pos, nor );

    vec3 lin = vec3(0.0);

        // sun
        {
            vec3  lig = normalize( vec3(-0.5, 0.4, -0.6) );
            vec3  hal = normalize( lig-rd );
            float dif = clamp( dot( nor, lig ), 0.0, 1.0 );
          //if( dif>0.0001 )
                dif *= calcSoftshadow( pos, lig, 0.02, 2.5 );
      float spe = pow( clamp( dot( nor, hal ), 0.0, 1.0 ),16.0);
                  spe *= dif;
                  spe *= 0.04+0.96*pow(clamp(1.0-dot(hal,lig),0.0,1.0),5.0);
            lin += col*2.20*dif*vec3(1.30,1.00,0.70);
            lin +=     5.00*spe*vec3(1.30,1.00,0.70)*ks;
        }
        // sky
        {
            float dif = sqrt(clamp( 0.5+0.5*nor.y, 0.0, 1.0 ));
                  dif *= occ;
            float spe = smoothstep( -0.2, 0.2, ref.y );
                  spe *= dif;
                  spe *= 0.04+0.96*pow(clamp(1.0+dot(nor,rd),0.0,1.0), 5.0 );
          //if( spe>0.001 )
                  spe *= calcSoftshadow( pos, ref, 0.02, 2.5 );
            lin += col*0.60*dif*vec3(0.40,0.60,1.15);
            lin +=     2.00*spe*vec3(0.40,0.60,1.30)*ks;
        }
        // back
        {
          float dif = clamp( dot( nor, normalize(vec3(0.5,0.0,0.6))), 0.0, 1.0 )*
              clamp( 1.0-pos.y,0.0,1.0);
                  dif *= occ;
          lin += col*0.55*dif*vec3(0.25,0.25,0.25);
        }
        // sss
        {
            float dif = pow(clamp(1.0+dot(nor,rd),0.0,1.0),2.0);
                  dif *= occ;
          lin += col*0.25*dif*vec3(1.00,1.00,1.00);
        }

    col = lin;

        col = mix( col, vec3(0.7,0.7,0.9), 1.0-exp( -0.0001*t*t*t ) );
    }

  return vec3( clamp(col,0.0,1.0) );
}

mat3 setCamera( in vec3 ro, in vec3 ta, float cr )
{
  vec3 cw = normalize(ta-ro);
  vec3 cp = vec3(sin(cr), cos(cr),0.0);
  vec3 cu = normalize( cross(cw,cp) );
  vec3 cv =          ( cross(cu,cw) );
    return mat3( cu, cv, cw );
}

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 mo = iMouse.xy/iResolution.xy;
  float time = 32.0 + iTime*1.5;

    // camera
    vec3 ta = vec3( 0.5, -0.5, -0.6 );
    vec3 ro = ta + vec3( 4.5*cos(0.1*time + 7.0*mo.x), 1.3 + 2.0*mo.y, 4.5*sin(0.1*
         time + 7.0*mo.x) );
    // camera-to-world transformation
    mat3 ca = setCamera( ro, ta, 0.0 );

    vec3 tot = vec3(0.0);
#if AA>1
    for( int m=ZERO; m<AA; m++ )
    for( int n=ZERO; n<AA; n++ )
    {
        // pixel coordinates
        vec2 o = vec2(float(m),float(n)) / float(AA) - 0.5;
        vec2 p = (2.0*(fragCoord+o)-iResolution.xy)/iResolution.y;
#else
        vec2 p = (2.0*fragCoord-iResolution.xy)/iResolution.y;
#endif

        // focal length
        const float fl = 2.5;

        // ray direction
        vec3 rd = ca * normalize( vec3(p,fl) );

         // ray differentials
        vec2 px = (2.0*(fragCoord+vec2(1.0,0.0))-iResolution.xy)/iResolution.y;
        vec2 py = (2.0*(fragCoord+vec2(0.0,1.0))-iResolution.xy)/iResolution.y;
        vec3 rdx = ca * normalize( vec3(px,fl) );
        vec3 rdy = ca * normalize( vec3(py,fl) );

        // render
        vec3 col = render( ro, rd, rdx, rdy );

        // gain
        // col = col*3.0/(2.5+col);

    // gamma
        col = pow( col, vec3(0.4545) );

        tot += col;
#if AA>1
    }
    tot /= float(AA*AA);
#endif

    fragColor = vec4( tot, 1.0 );
}
```

Appendix B

Appendix B

Code listing B.1: Fragment shader generated by the compiler.

```
#version 300 es
#ifdef GL_FRAGMENT_PRECISION_HIGH
  precision highp float;
  #else
  precision mediump float;
  #endif

  vec2 UV;

  const float epsilon = 0.005;
  const vec3 lightPos = vec3(0.7, 2.0, -5.0);
  const float maxDist = 50.0;
  #define PI 3.1415

  uniform vec3 cameraPos;
  uniform vec3 bottomLeftCorner;
  uniform vec3 left;
  uniform vec3 up;
  out vec4 oCol;

  float evaluate(vec3 rayPos, out vec3 col, float epsilon);
  float density(vec3 rayPos);

  vec3 gradient(in vec3 pos) {
    vec3 ex = vec3(epsilon,0,0);
    vec3 ey = vec3(0,epsilon,0);
    vec3 ez = vec3(0,0,epsilon);

    return normalize(vec3(density(pos+ex)-density(pos-ex),
          density(pos+ey)-density(pos-ey),
          density(pos+ez)-density(pos-ez)));
  }

  uniform vec3 vector3s[12];uniform float floats[4];uniform vec2 vector2s[3];

  vec3 skybox(vec3 p) {return vec3(0.992,0.675,0.655);}

  vec2 sphereUVs(vec3 rayPos);float sphere(vec3 rayPos);float box(vec3 rayPos);vec2
       interpolatedUVs(vec3 rayPos,float weight,vec3 position);vec2 BoxUVFunc(vec3
      rayPos);vec2 BoxUVFunc(vec3 rayPos,vec3 position,vec2 UVs,vec3 size,float
      rotation);vec2 sphereUVs(vec3 rayPos){
                  vec3 d = normalize(-rayPos-vec3(0,0,0));
                  return vec2(+0.5+atan(d.z, d.x)/(2.0*PI), +0.5-asin(-d.y)/PI)+
                      vec2(0,0);}
  float sphere(vec3 rayPos){vec3 p = rayPos-vec3(0,0,0);
            return length(p)-1.0;}
  float box(vec3 rayPos){
            vec3 p = rayPos-vec3(0,0,0);
            mat2 rot = mat2(cos(0.0),-sin(0.0),
                            sin(0.0),cos(0.0));
            p.xz *= rot;
            vec3 q = abs(p) - vec3(0.5,0.5,0.5);
            return length(max(q, 0.0)) + min(max(q.x,max(q.y,q.z)),0.0);}
  vec2 interpolatedUVs(vec3 rayPos,float weight,vec3 position){
                  return weight*sphereUVs(rayPos)+(1.0-weight)*BoxUVFunc(rayPos);}
  vec2 BoxUVFunc(vec3 rayPos){
        vec3 p = rayPos-vec3(0,0,0);
        mat2 rot = mat2(cos(0.0),-sin(0.0),
                        sin(0.0),cos(0.0));
        p.xz *= rot;
        vec3 n = gradient(rayPos);
        n.xz *= rot;
        n = abs(n);
        p = p/vec3(0.5,0.5,0.5)*0.5+vec3(0.5,0.5,0.5);
        return n.x*p.zy+n.y*p.xz+n.z*p.xy;
        }
  vec2 BoxUVFunc(vec3 rayPos,vec3 position,vec2 UVs,vec3 size,float rotation){
        vec3 p = rayPos-position;
        mat2 rot = mat2(cos(rotation),-sin(rotation),
                        sin(rotation),cos(rotation));
        p.xz *= rot;
        vec3 n = gradient(rayPos);
        n.xz *= rot;
        n = abs(n);
        p = p/size*0.5+vec3(0.5,0.5,0.5);
        return n.x*p.zy+n.y*p.xz+n.z*p.xy;
        }
  float sphere0(vec3 rayPos){vec3 p = rayPos-vec3(0,0,0);
            return length(p)-1.0;}
  float interpolatedShape0(vec3 rayPos,float weight,vec3 position){rayPos = rayPos
      - position;
            return weight*sphere(rayPos)+(1.0-weight)*box(rayPos);}
  float boxFramed0(vec3 rayPos,vec3 position,vec3 size){
            vec3 p = rayPos-position;
            mat2 rot = mat2(cos(0.0),-sin(0.0),
                            sin(0.0),cos(0.0));
            p.xz *= rot;
            p = abs(p) - size;
            vec3 q = abs(p+0.01)-0.01;
            return min(min(
              length(max(vec3(p.x,q.y,q.z),0.0))+min(max(p.x,max(q.y,q.z)),0.0),
              length(max(vec3(q.x,p.y,q.z),0.0))+min(max(q.x,max(p.y,q.z)),0.0)),
              length(max(vec3(q.x,q.y,p.z),0.0))+min(max(q.x,max(q.y,p.z)),0.0));}
  float box0(vec3 rayPos,vec3 position,vec2 UVs,vec3 size,float rotation){
            vec3 p = rayPos-position;
            mat2 rot = mat2(cos(rotation),-sin(rotation),
                            sin(rotation),cos(rotation));
            p.xz *= rot;
            vec3 q = abs(p) - size;
            return length(max(q, 0.0)) + min(max(q.x,max(q.y,q.z)),0.0);}
```

```


vec3 phongShadedDefObject0(vec3 rayPos,vec3 color){vec3 surfaceN = gradient(rayPos);
    vec3 dir = normalize(lightPos-rayPos);

    float shadow = 1.0;
    // rayPos -= surfaceN*0.01;
    for(float t=20.0*epsilon; t<maxDist;) {
      float h = density(rayPos + dir*t);
      if(h < epsilon) {
        shadow = 0.0;
        break;
      }
      t += h;
    }

    float lightStrength = max(0.5, dot(surfaceN, -normalize(rayPos-lightPos)));
    return lightStrength*color*shadow;}
vec3 UVColored0(vec3 rayPos){return vec3(BoxUVFunc(rayPos,vector3s[6],vector2s[0],vector3s[7],floats[1]), 0.0);}
vec3 UVColored1(vec3 rayPos){return vec3(BoxUVFunc(rayPos,vector3s[8],vector2s[1],vector3s[9],floats[2]), 0.0);}
vec3 UVColored2(vec3 rayPos){return vec3(BoxUVFunc(rayPos,vector3s[10],vector2s[2],vector3s[11],floats[3]), 0.0);}


float density(vec3 rayPos) {return min(box0(rayPos ,vector3s[10],vector2s[2],vector3s[11],floats[3]), min(box0(rayPos ,vector3s[8],vector2s[1],vector3s[9],floats[2]), min(box0(rayPos ,vector3s[6],vector2s[0],vector3s[7],floats[1]), min(boxFramed0(rayPos ,vector3s[3],vector3s[4]), min(interpolatedShape0(rayPos ,floats[0],vector3s[1]), sphere0(rayPos ))))));}

vec3 color(vec3 rayPos) {if(sphere0(rayPos ) < epsilon) {
    return phongShadedDefObject0(rayPos ,vector3s[0]);
  }if(interpolatedShape0(rayPos ,floats[0],vector3s[1]) < epsilon) {
    return phongShadedDefObject0(rayPos ,vector3s[2]);
  }if(boxFramed0(rayPos ,vector3s[3],vector3s[4]) < epsilon) {
    return phongShadedDefObject0(rayPos ,vector3s[5]);
  }if(box0(rayPos ,vector3s[6],vector2s[0],vector3s[7],floats[1]) < epsilon) {
    return UVColored0(rayPos );
  }if(box0(rayPos ,vector3s[8],vector2s[1],vector3s[9],floats[2]) < epsilon) {
    return UVColored1(rayPos );
  }if(box0(rayPos ,vector3s[10],vector2s[2],vector3s[11],floats[3]) < epsilon) {
    return UVColored2(rayPos );
  } return vec3(0.0,0.0,1.0);}


vec3 marchRay(vec3 dir, vec3 origin, out bool hit) {
  float val = 0.0;
  for(float distanceTraced = 0.0; distanceTraced < maxDist; distanceTraced+=val) {
    vec3 pos = origin + vec3(dir*distanceTraced);
    val = density(pos);

    if(val <= epsilon) { //hit object
      hit = true;
      return pos;
    }
  }
```

```
    hit = false;
    return vec3(0.0,0.0,0.0);
  }

  void main() {
    vec3 col = vec3(0,0,1.0);
    vec2 uvs = vec2(gl_FragCoord.x/960.0,
            gl_FragCoord.y/540.0);

    vec3 ray = bottomLeftCorner - left*uvs.x + up*uvs.y;
    ray = normalize(ray);

    bool hitObject = false;
    vec3 currentPos = marchRay(ray, cameraPos, hitObject);
    if(hitObject) {
      col = color(currentPos);
    } else {
      col = skybox(ray);
    }
    oCol = vec4(col, 1.0);
  }
```

Appendix C

Appendix C

Code listing C.1: Script to generate a scene with multiple primitives.

```

//Utility function for creating objects
const shapeAtPos = (s,x,z)=>
  create(object("Shape"+x+z)
    .shape(s)
      .position().setUpdate((dt,ft)=>[x,Math.sin(ft+x*0.1+z*0.3)*0.2,z])
    .material(phongShaded)
      .color([Math.random(),Math.random(),Math.random()]));

//Create some objects in a grid
shapeAtPos(sphere,       0,0);
shapeAtPos(box,            0,3);
shapeAtPos(boxFrame, 3,0);
shapeAtPos(torus,         3,3);

//Set the skybox
setSkybox("skyboxTex");
create(texture("skyboxTex")
  .data(fromCanvas)
  .width(1)
  .height(1)
    .setUpdate((dt,ft,ctx,w,h)=> {
      ctx.fillStyle = \‘hsl(\${Math.floor((ft*10))%360},80%,50%)\‘;
      ctx.fillRect(0,0,w,h);
    }));
```

Code listing C.2: Script to generate a scene with intersecting primitives.

```
//Place the light above and slightly to the side of the objects
setLightPosition([2,10,-3]);

//Utility function for making boxes with a certain size
const boxAtPos = (x,y,size) => create(object("Box"+x+y)
  .shape(box)
    .position([x,y,0])
    .size([size,size,size])
  .material(phongShaded)
    .color([Math.random(),Math.random(),Math.random()]))
//Make 7 boxes that intersect
```

```
boxAtPos(-1,0,0.5);
boxAtPos(-1,0.5,0.25);
boxAtPos(-1,0.75,0.125);
boxAtPos(-1.5,0,0.25);
boxAtPos(-1.75,0,0.125);
boxAtPos(-0.5,0,0.25);
boxAtPos(-0.25,0,0.125);

//Three interlinked box frames
create(object("BoxFrame1")
  .shape(boxFrame)
    .position([1,0,0])
    .thickness([0.05])
  .material(phongShaded)
    .color([0.8,0.2,0.4]));

create(object("BoxFrame2")
  .shape(boxFrame)
    .position([1.5,0.5,0.5])
    .thickness([0.05])
  .material(phongShaded)
    .color([0.2,0.4,0.8]));

create(object("BoxFrame3")
  .shape(boxFrame)
    .position([2,1,1])
    .thickness([0.05])
  .material(phongShaded)
    .color([0.4,0.8,0.2]));

//Floor
create(object("Floor")
  .shape(box)
    .size([20,1,20])
    .position([0,-1.5,0])
  .material(squaresShaded));

//Make the sky a blue-ish color
setSkybox([0.6,0.6,1]);

//Move the camera automatically
onUpdate((dt,ft)=>setCameraPosition([Math.cos(ft)*5,3,Math.sin(ft)*5]));
onUpdate(()=>lookAt([0,0,0]));
```

Code listing C.3: Script to generate a scene where primitives move in a coordinated way using JavaScript scripting.

```

//BoxFrame
create(object("Frame")
  .shape(boxFrame)
    .thickness([0.05])
    .size([1.2,1.2,1.2])
      .setUpdate((dt,ft)=>{
        const d = ((ft/(Math.PI)+1/20)%1)*20;
        let s = 1.2-0.7*d*Math.exp(1-d);
        return [s,s,s]})
    .position([0,0,-4])
  .material(phongShaded)
```

```
    .color([0.2,0.4,0.8]));

//Torus
create(object("Torus")
  .shape(torus)
    .size([1.1])
    .thickness([0.33])
    .position([0,0,-10])
      .setUpdate((dt,ft)=>[2*Math.sin(3*ft),0,-10])
  .material(phongShaded));

//Sphere-box
create(object("Interpolated")
  .shape(interpolatedShape)
    .weight().setUpdate((dt,ft)=>[0.5*Math.cos(ft+Math.PI)+0.5])
    .shapeA(object("Sphere")
      .shape(sphere)
        .radius([0.77])
          .setUpdate((dt,ft)=>[0.5+0.27*(Math.cos(2*ft+Math.PI)+1)/2]))
    .shapeB(object("Box")
      .shape(box))
    .position()
      .setUpdate((dt,ft)=>[0,3*Math.sin(2*ft),3*Math.cos(ft)-7])
  .material(skyboxReflective));

setSkybox("sky")
create(texture("sky")
  .data(fromFetch)
    .setInit((dt,ft,set)=>set("https://github.com/rpgwhitelock/AllSkyFree_Godot/raw
         /master/addons/AllSkyFree/Skyboxes/AllSkyFree_Sky_EpicGloriousPink_Equirect
         .png")));
//Move the camera automatically
onUpdate(
  (dt,ft)=>setCameraPosition([Math.cos(ft/4)*13,
                              5,
                              Math.sin(ft/4)*13-7]))
onUpdate(()=>lookAt([0,0,-7]))
```

Code listing C.4: Script to create a planet using multiple textures fetched from the web.

```

//Fetch the different textures needed
create(texture("sb")
  .data(fromFetch)
    .setInit((dt,ft,setSrc)=> {
      setSrc("https://cdn.eso.org/images/large/eso0932a.jpg");
    }));
setSkybox("sb");
create(texture("earthDiffuse")
  .data(fromFetch)
    .setInit((dt,ft,setSrc)=> {
      setSrc("http://shadedrelief.com/natural3/ne3_data/8192/textures/4
           _no_ice_clouds_mts_8k.jpg");
    }));
create(texture("earthSpecular")
  .data(fromFetch)
    .setInit((dt,ft,setSrc)=> {
      setSrc("http://shadedrelief.com/natural3/ne3_data/8192/masks/water_8k.png");
    }));
create(texture("earthShadow")
  .data(fromFetch)
    .setInit((dt,ft,setSrc)=> {
      setSrc("http://shadedrelief.com/natural3/ne3_data/8192/textures/5_night_8k.
          jpg");
    }));

//Make the sphere representing the earth
create(object("Earth")
  .shape(sphere)
    .UVOffset().setUpdate((dt,ft)=>[0.02*ft%1,0])
    .radius([1.5])
  .material(textured)
    .diffuse("earthDiffuse")
    .specular("earthSpecular")
    .shadow("earthShadow"));
//Move the camera
//onUpdate((dt,ft)=>(setCameraPosition([Math.cos(-ft/5)*(5+ft/3),0,Math.sin(-ft/5)
    *(5+ft/3)]),
//                  lookAt([0,0,0])));
```

Code listing C.5: Script to load a JavaScript library from the scripting environment.

```
//Create the sphere we will texture using Perlin noise.
create(object("Planet")
    .shape(sphere)
      .UVOffset().setUpdate((dt,ft)=>[(ft/10)%1,0])
      .radius([2])
    .material(textured)
      .texture("PerlinTex"));

//Draw function that loads a Perlin noise library to draw
const drawF = (ctx,w,h) => fetch("https://pastebin.com/raw/MZWiaBL6")
  .then((r)=>r.text())
  .then((p)=>{
    let getPerlin = new Function(p+"return␣Perlin");
    const Perlin = getPerlin();
    ctx.fillStyle = "white";
    ctx.fillRect(0,0,w,h);
    ctx.fillStyle = "black";
    for(let u = 0; u < w; u++) {
      for(let v = 0; v < h; v++) {
        let theta = Math.PI*2*u/w;
        let phi = Math.PI*v/h;
        let x = Math.cos(theta)*Math.sin(phi);
        let y = Math.sin(theta)*Math.sin(phi)
        let z = -Math.cos(phi);
        let val = Perlin.noiseOctaves(x,y,4,z);
        ctx.fillStyle = getColor(val/1.8);
        ctx.fillRect(u,v,1,1);
      }
    }
});

//Utility function for mapping height values to colors.
const getColor = (height) => {
  if(height < 0.35) {
    return "rgb(60,104,192)";
  }
  if(height < 0.45) {
    return "rgb(64,110,200)";
  }
  if(height < 0.48) {
    return "rgb(208,207,130)";
  }
  if(height < 0.55) {
    return "rgb(84,150,29)";
  }
  if(height < 0.6) {
    return "rgb(61,105,22)";
  }
  if(height < 0.7) {
    return "rgb(91,68,61)";
  }
  if(height < 0.87) {
    return "rgb(75,58,54)";
  }
  return "rgb(255,254,255)";
}

//Create the actual texture using the Perlin library function.
create(texture("PerlinTex")
  .data(fromCanvas)
    .width(1024)
    .height(1024)
    .setInit((dt,ft,ctx,w,h)=>drawF(ctx,w,h)));


setSkybox([30/255,40/255,70/255])
```