I♥IQ: A SHADER GRAPHING CALCULATOR FOR SIGNED DISTANCE FUNCTIONS (SDFS)

by

Henro Kriel

A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Master of Science
Computer Science

Committee:

_____	Dr. Yotam Gingold, Thesis Director
_____	Dr. Lap-Fai (Craig) Yu, Committee Member
_____	Dr. Elizabeth White, Committee Member
_____	Dr. David Rosenblum, Chair, Department of Computer Science

Date: _____ Spring Semester 2023
George Mason University
Fairfax, VA

I♥IQ: A Shader Graphing Calculator for Signed Distance Functions (SDFs)

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Henro Kriel
Bachelor of Science
George Mason University, 2022

Director: Dr. Yotam Gingold, Professor
Department of Computer Science

Spring Semester 2023
George Mason University
Fairfax, VA

Dedication

I dedicate this thesis to my brother, Aiden, who got me a two-handed Scottish Claymore for my birthday. Without all the time I wasted playing video games and watching TV with him, I would not have been able to finish this thesis.

Acknowledgments

I would like to thank my advisor, Dr. Yotam Gingold, whose guidance and wisdom were invaluable to me and the project. I would also like to thank Yong Li, the creator of I❤LA, who has been incredibly kind and helpful in general and in helping me adapt his work.

Table of Contents

List of Figures

Abstract

I❤IQ: A SHADER GRAPHING CALCULATOR FOR SIGNED DISTANCE FUNCTIONS (SDFS)

Henro Kriel

George Mason University, 2023

Thesis Director: Dr. Yotam Gingold

Shaders are useful in real time graphics and high performance computing applications as they specify computation to be run in parallel on the GPU. In the community, these shaders are often described in mathematical syntax and translating that math to executable code can be tedious and error prone. We present I❤IQ, a graphing calculator of sorts for signed distance functions (SDFs) and materials that expedites the process of prototyping shaders. It provides a math-like syntax using I❤LA and comes with a built-in raycasting architecture, removing the overhead of translation and implementation details. I❤IQ is designed to be responsive and interactive. The system automatically detects free parameters and lets users tweak them using slider controls, allowing for seamless manipulation of the scene in real time. The code generated by I❤IQ can then be exported to third party programs and used outside of the I❤IQ environment. The I❤IQ repository can be found at `https://github.com/HenroKriel/heartdown`.

Chapter 1: Introduction

Shaders are high performance graphics programs that run on the GPU, and they are extremely common in real time graphics applications such as video games. Some popular examples of their use include the building interiors in Spiderman [1] and the liquid within bottles in Half-Life: Alyx [2], though shaders are everywhere behind the scenes. Shaders are also popular for real time computer generated art, such as those created by demosceners [3].

The art generated by shaders will typically be done by modeling objects mathematically, as opposed to using discrete data structures such as meshes. For instance, moving the camera around and into a fractal while rendering it in real time is popular in this domain [4]. The problem is that translating math into shader code takes time and is prone to errors, and shaders can be hard to debug, especially since that often entails creating an architecture like raymarching for rendering whatever was modeled. Additionally, discussion concerning the implementation of these shaders is often done with math, not shader language syntax necessarily. In this showcase, Inigo Quilez demonstrates the implementation details with traditional math syntax [5]. So, a system for automatically translating math to executable shader code would save time in translation, designing and implementing a rendering architecture, and would eliminate debugging as a result of implementation errors.

Our system, I❤IQ, compiles I❤LA, a language with math-like syntax, to executable shader code, specifically GLSL. See figure 1.1 for an example scene written in I❤IQ. Users can define shapes and materials, and then in a scene script instances of those shapes can be declared. User-defined materials and a transformation matrix are applied to each instance of a shape. I❤IQ comes with a raymarching backend that then loads the generated code and renders a scene so the user gets immediate feedback. That is the vision for I❤IQ. Users should be able to make changes quickly and easily and see the effects of those changes

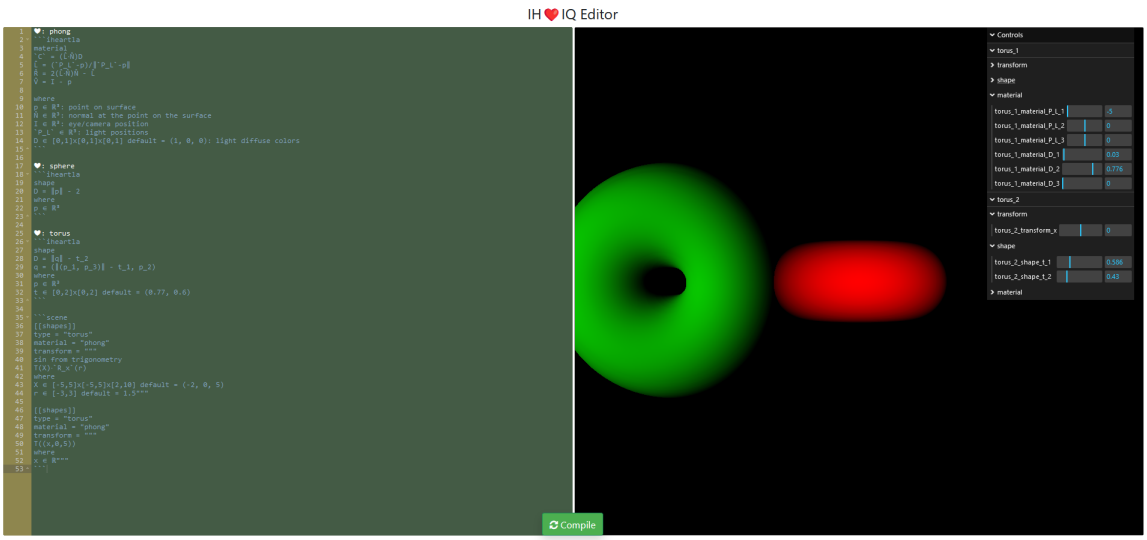

Figure 1.1: An example scene.

in close to real time. To this end, free parameters are picked up by I❤IQ and are given corresponding interactable sliders. This allows the users to tinker with the values until they get the scene that they want without having to recompile. I❤IQ is designed to expedite the prototyping stage for generating these shaders. Then, users can take the generated GLSL and use it however they want, such as in a graphics application. For this reason, code generation was made with readability in mind. This motivated features such as input and output structs, which are used as interfaces between functions.

Consider a graphing calculator, such as Desmos [6]. One of their purposes is to quickly test functions so that the user gets the one that they want. Say a user wanted to define a sine wave whose period maps to [0,1] on the x axis and whose amplitude also maps to [0,1]. A graphing calculator lets the user quickly define the function, check it for accuracy, and tweak it should they desire a variation on the curve. SDFs are the main primitive for defining surfaces in raymarched shader programs, and they look like something that one could plug into a graphing calculator. However, that graphing calculator does not exist, and I❤IQ fills this niche.

Chapter 2: Related Work

2.1 Math-Like Syntax

I❤IQ is an extension of H❤rtDown [7], which itself is built on I❤LA [8]. I❤LA embraces a syntax that closely approximates the look of conventional math while being unambiguous in interpretation. Languages in the past have experimented with introducing syntax from conventional math. Fortress [9] interpreted juxtaposition as multiplication, and Julia [10] let users define unicode math symbols as operators. In the Julia paper, the authors emphasize the ability to mimic mathematical idiosyncrasies. For instance, if f and g are both functions, $f * g(x)$ might represent a composition, $f(g(x))$. I❤LA takes this to the extreme with the hope of providing a common high level interface that can be mapped to executable relatively lower level languages such as Python.

2.2 Shader Languages

I❤IQ was not designed to be a shader language or an extension of a shader language. Rather, it is designed to be a meta-shader-language of sorts. It is a replacement for the mathematical subset of a typical shader language. Furthermore, I❤IQ does not support direct control over the rendering. Instead, rendering is handled by a raymarching back-end that takes the processed I❤IQ script as input.

An example of a proper shader language is Slang [11], which is an extension of HLSL aimed at incorporating best-practice principles into a shader language. Unlike Slang, I❤IQ is not an extension of GLSL. Instead, it takes I❤LA, which doesn't resemble GLSL, and compiles it to GLSL. Additionally, I❤IQ does not necessarily introduce new features to GLSL aside from the boilerplate needed for interactive raycasting.

However, future work might entail supporting higher level semantic error checking. For example, Geisler et al [12] somewhat recently published a paper on a language called Gator that checks geometry types for variables. Gator introduces generic-like syntax for vectors that declares which space they belong to. Similarly, for matrices the user declares which spaces the matrix transforms from and to. Type rules are then checked for operations. An error will be thrown when the user tries to subtract two vectors in different spaces, for instance. Users can also declare whether a vector describes a point or direction, and then the corresponding type rules will be checked. For example, two point-vectors can't be added.

This higher level type checking, such as direction and position types for vectors, would be a great addition to I❤IQ. However, the conversion between spaces is already handled by the backend. SDFs are defined in an object space, and all the user defined transformation matrices take the shape instances from object space to world space. Additionally, the transformation to screen space is handled implicitly by raycasting. Thus, much of the mental overhead is already removed. Gator was proposed because handling multiple different geometric spaces is hard and error prone, and since I❤IQ also prevents many of these user errors, this further cements its place as a usable prototyping tool.

2.3 Interactive Documents and Educational Frameworks

I❤IQ was made in the spirit of interactive documents. Tangle [13] is a Javascript library for this purpose. The user declares interactive components using html classes and can access variables in the interface using the data-var attribute. There is a standard library of components, but they can also be defined by the user. However, html isn't comparatively readable. So, Conlen and Heer proposed Idyll [14], which, like H❤rtDown, is an extension of markdown. It also has a standard library of components and supports custom component creation, but now with the added benefit of the readability of markdown. Streamlit [15] is a tool for building data apps that uses python as an interface instead of a markup language. Nonetheless, similar to a markup language, components appear on the document in the

order that they are declared in the script, and Steamlit will write variables on their own line in the script to the screen without a function call. Furthermore, the web app will hotload whenever a change is detected in the script file. It is designed to be usable and fast, which is what we hope I❤IQ will be.

I❤IQ was also inspired by educational graphics frameworks like Processing [16] and openFrameworks [17]. These frameworks are designed to be simple and usable but powerful tools for teaching graphics. They are not necessarily intended to be used in industry applications, but they are great tools for learning. They are still powerful and versatile, and thus make great tools for experts who want to prototype ideas. As such, these frameworks are targeted to both beginners and experts. These attributes are what we desire for I❤IQ as well.

2.4 Similar Systems

Naturally, I❤IQ takes heavy inspiration from signed distance functions (SDFs) and the work of Inigo Quilez [18], who is the IQ in I❤IQ. Signed distance functions are the bedrock of raycasting with fragment shaders, described as early as 1989 [19]. These are implicit descriptions of the surfaces of (usually) 3D shapes and enable a computationally efficient form of raymarching called sphere tracing [20], which is the mechanism that I❤IQ uses. Quilez uses SDFs in his work extensively. He is an avid demoscener and the co-creator of Shadertoy [21]. I❤IQ is meant to be a higher level version of Shadertoy where the syntax is math based and the ray marching architecture is implicit. This way, much of the implementation details are extracted away, allowing users to prototype quickly and effectively. Quilez' video on painting a character with maths [5] and his other such videos are the quintessential use case of I❤IQ. Users could then take the generated GLSL code, tweak it, and use it in other applications. As a general compiler for mathematics into GLSL it can be used for any mathematical functions to be executed on the gpu such as illumination for materials, animation curves, and potentially general purpose gpu computation.

Finally, I❤IQ is similarly motivated by the Desmos graphing calculator [6]. This online tool takes conventional math as input and graphs the equations. It's very useful for finding or checking equations that satisfy the user's needs, such as a sine wave whose period maps to [0, 1] and whose amplitude varies between [0, 1]. In a way, I❤IQ is a 3D graphing calculator with support for materials. Users should be able to quickly define and check the geometry of SDFs. The user might want to describe two interlocking box frames (see figure 6.3). I❤IQ should make it easy to generate SDFs and transformation functions that satisfy that requirement.

Chapter 3: Background

I❤IQ generates fragment shaders (also known as pixel shaders). Fragment shaders are instructions for which color to apply to a pixel on the output screen. The code for a fragment shader is run in parallel for a batch of pixels on the GPU.

Typically, fragment shaders are applied to a mesh, which is stored in object space. The positions of the vertices in world space and the normals of the surfaces on this mesh are supplied by a vertex shader, which, similar to the fragment shader, does the computations for positions, normals, and screen space positions in parallel on the GPU. The host program iterates over primitives, which are almost always triangles formed by the vertices, and uses screen space position output by the vertex shader to determine which pixels on the screen are inside the projected primitive. If primitives overlap, a depth value is used to choose which primitive determines the color of a given pixel.

However, instead of using discrete precomputed surfaces defined by meshes, one could mathematically model a surface within the fragment shader. This is usually done with implicit surfaces, where for some function $F(x, y, z)$ the surface is defined as the set of points x, y, z such that $F(x, y, z) = 0$. In a closed surface, a positive output means that the input point is outside the surface, and a negative output means the input is within the surface. Raycasting could then be used to render the surface. Raycasting is done by generating a ray that originates from the camera and passes through a pixel position on a virtual screen for each pixel. If a ray intersects a surface, then the pixel that the ray passes through is colored according to the surface's material (Figure 3.1).

It is possible to derive closed form solutions for the point at which a ray intersects a surface, but a solution would have to be derived for every surface primitive. Additionally, each surface primitive has different conditions to check whether the ray intersects the shape at all. Thus, closed form solutions are not generalizable. Raymarching, on the other hand,

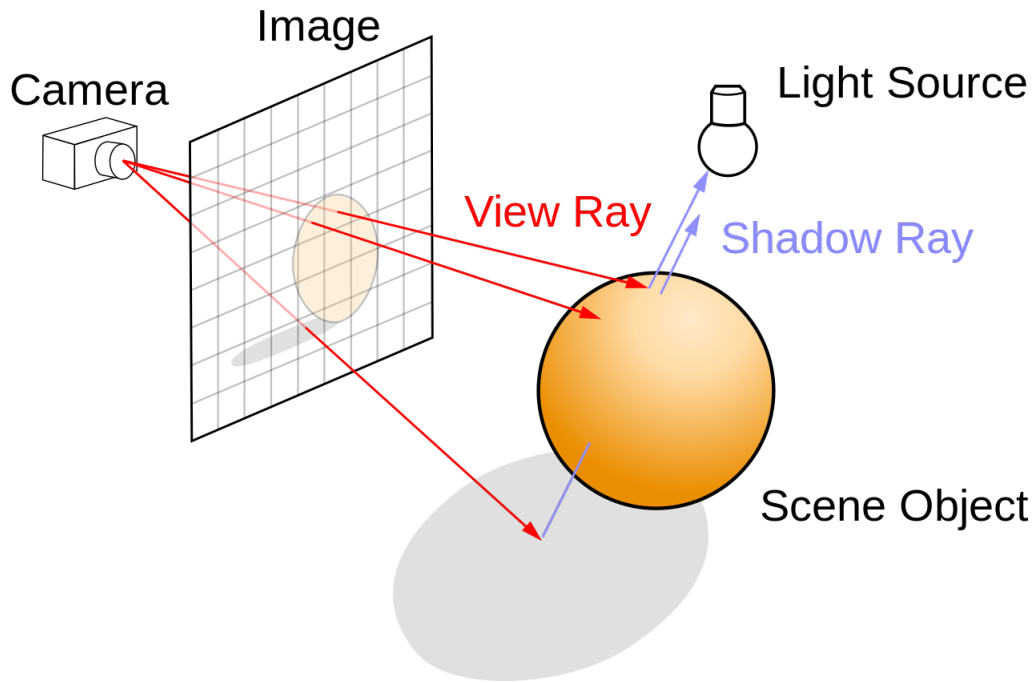

Figure 3.1: This diagram [22] depicts ray tracing, which is like ray casting but rays are also cast at points of intersection to create shadows and reflections.

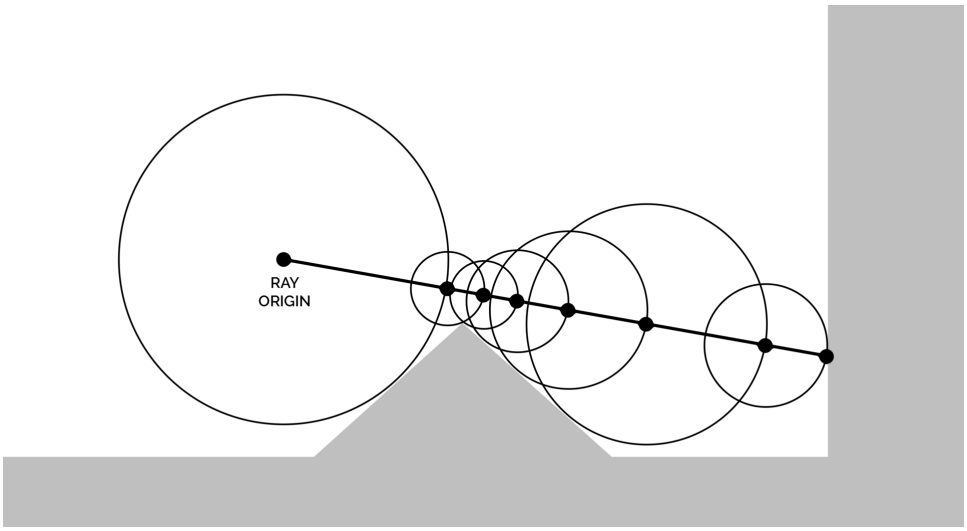

Figure 3.2: A visualization of sphere tracing [23].

is generalizable and is a more popular choice. Raymarching works as follows. A point is placed at the origin of the ray, and at each iteration the point is used to query the implicit surface function and is then moved along the ray by a small amount. If the value goes from positive to negative, or vice versa, then by the intermediate value theorem it is known that the function returns 0 at some point between, therefore the ray has crossed the boundary of the surface. Of course, this mandates that surfaces be closed, otherwise there would not be a definitive inside or outside, or respectively a positive and negative region of the implicit surface function.

Signed Distance functions (SDFs) are popular implicit surface functions for raymarching because, as mentioned in the related work section, they can be rendered efficiently using a method called sphere tracing [20]. Instead of moving forward by a small increment each iteration, the query point moves forward by the value returned by the SDF. A visualization of sphere tracing can be seen in figure 3.2.

SDFs can be composed of other SDFs by using functions to combine them. Most simply taking the min of two SDFs will generate a surface that bounds the union of the two shapes (more specifically the union of their volumes). Similarly, taking the max of two SDFs

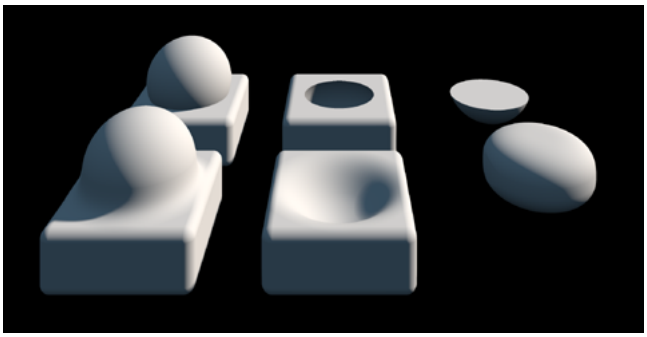

Figure 3.3: SDF combinations, including union, subtraction, intersection, and their smooth variants [24].

will take the intersection. For more combinations, see [24]. Quilez includes more complex combinations such as one that produces a smooth union, which can be seen in figure 3.3. Combinations of SDFs can yield arbitrarily complex shapes.

It is important to know how to compute the normal of a surface given its SDF. Many materials, such as the ubiquitous Phong [25], require the normal of a position as input. To compute the normal of an SDF, you simply need to calculate the gradient and normalize it [26]. Here is why that works conceptually. The gradient is the direction and rate of fastest increase, and the direction which maximizes the output of the SDF is the direction which maximizes the distance from the surface, that being the normal.

Chapter 4: Design

We decided to follow the H♥rtDown model, where the document source code is on the left half of the screen, and on the right is the output document or, in the case of I♥IQ, the scene. H♥rtDown was chosen so that in the future, users would be able to place the rendered scene as supplementary material in interactive documents.

We added a few things to the I♥LA grammar. Some were features that could be added back into I♥LA, such as max, min, and floor (see figure 6.2 for an example use of floor to define a toon shader material), while others were more structural. You can, for instance, declare I♥LA modules as shapes or materials. In addition to the main function that they generate, shape modules will generate gradient functions, which are used for finding the normal at a given point near or on the surface. Remember that the gradient is in the same direction as the normal when working with SDFs. These gradient functions are implemented using finite differencing.

Shapes and materials require that the user return information to the backend using named I♥LA variables. In shape modules, D (for distance) holds the output of the shape's SDF. C (for color) in material modules holds the resulting RGB color value for a given point on the surface. The RGB color's components range from 0 to 1 like in GLSL. Shapes and materials also have their own set of required parameters through which information is supplied by the backend. Shapes require a position parameter, which is used to query their SDFs. Materials require a parameter representing the position of a point on the surface of the shape, the surface's normal at that point, and the position of the eye/camera, which is used to calculate things like specular lighting (see figure 6.1).

The rest of the parameters defined by the user are automatically assigned sliders so that the user can tweak their values in real time. Examples of the slider controls can be found

```
48 ```scene
49 [[shapes]]
50 type = "torus"
51 material = "phong"
52 transform = """
53 T(X)·`R_x`(r)
54 where
55 X ∈ [-5,5]x[-5,5]x[2,10] default = (-1.37, 0, 5)
56 r ∈ [-3,3] default = 2.73"""
57
58 [[shapes]]
59 type = "torus"
60 material = "toon"
61 transform = """
62 T(X)·`R_x`(r)
63 where
64 X ∈ [-5,5]x[-5,5]x[2,10] default = (2, 0, 5)
65 r ∈ [-3,3] default = 2.73"""
66 ```
```

Figure 4.1: An example scene codeblock. The double bracket syntax in TOML indicates an array of tables. So, [[shapes]] is an array named "shapes" that contains tables which describe each shape. The tables contain type, material, and transform entries.

in figure 1.1 and 6.3. The upper and lower bounds of these sliders are defined using typical interval notation in place of the number space type, e.g. [0,1]. Bounds for vectors are done with x as a delimiter where each bound corresponds to a vector component, e.g. [0,1]x[0,1]. Defaults for these parameters can be set by declaring them to the right of their bounds. If no default is specified, it will be set to the average of the left and right bounds.

As of now, only number literals can be placed in these parameter definitions. Because the bounds are enforced by the sliders, which are written in JavaScript, evaluating expressions in the bounds and defaults would require a code generator for JavaScript, which is not a priority at the moment.

With the shapes and materials defined, users can then describe a scene with a code block named "scene". An example scene code block can be seen in figure 4.1. Instances of shapes are declared in sequence. Each shape instance has a type, material, and transformation matrix. Here, the type refers to the shape module that the shape instance instantiates.

For the sake of usability, we have provided a set of functions written in I❤LA that take 3D vectors as input and return 4x4 homogeneous transformation matrices. These are standard

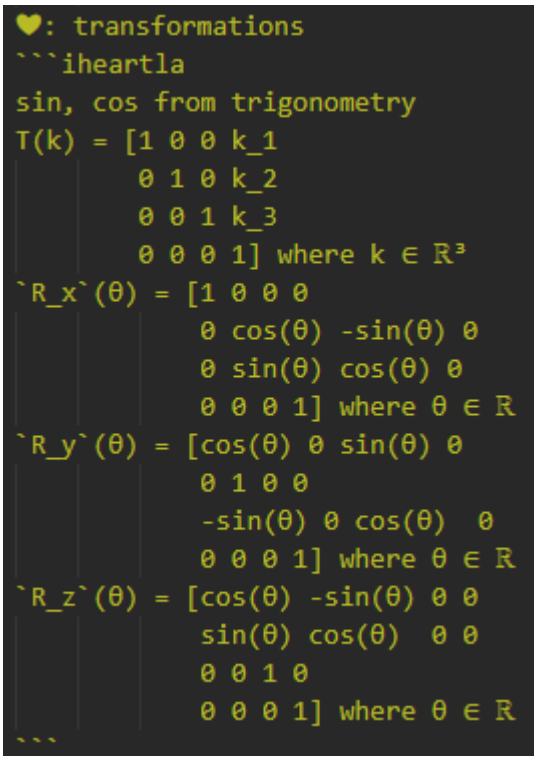

Figure 4.2: The transformations module.

transformations such as translate and rotate. The transformation matrix declarations in the scene codeblock are processed as I♥LA, so these functions can be called and chained. See figure 4.2 for their definitions.

After the user compiles the code, the scene is rendered in the pane to the right of the screen. In the top right of the rendered scene are the controls which hold the sliders. Sliders are organized in a hierarchical fashion. The top level of the hierarchy are shape instances, which are enumerated by type. Eventually we would like to allow the user to name shaped instances in the scene code block. Next, each shape instance has sliders for its shape type, transformation function, and material. Finally, scalars have their own sliders, and vectors are split into sliders for each component.

Defaults for sliders apply to shapes and materials globally. In the future, users will hopefully be able to pass default values to shape instances in the scene code block.

```
if(sect.material == 1) {
    phong_input _input;
    _input.p = sect.pos;
    _input.N_hat = sect.norm;
    _input.I = eye;
    _input.k_a = torus_1_material_k_a;
    _input.k_d = torus_1_material_k_d;
    _input.k_s = torus_1_material_k_s;
    _input.A = torus_1_material_A;
    _input.alpha = torus_1_material_alpha;
    _input.P_L = torus_1_material_P_L;
    _input.D = torus_1_material_D;
    _input.S = torus_1_material_S;

    color = phong(_input).C;
}
```

Figure 4.3: An example of use of structs as input and output.

Some design decisions were made concerning the code generation as well. It is possible for the number of parameters to become unwieldy, especially for something like materials. Figure 4.3 illustrates one such example. In these cases, it is unclear which argument maps to which parameter in a function call. As such, we have decided to use input structs to interface between functions in order to approximate named arguments in languages like Python. Structs in GLSL can be instantiated using a parenthesis enclosed list similar to arguments in a function call, so the user can still use that syntax if they desire. Additionally, a struct input for a shape needs to be reused in calls to its respective gradient function since each shape instance has its own sliders and thus its own values for free parameters. We also use structs as output like I♥LA does in its generated code. Any variable that is assigned a value in an I♥LA module is stored in an output struct, which is returned at the end of the function call, so the user can select which of the outputs they need.

Chapter 5: Implementation

5.1 Web GUI

I♥IQ compiles I♥LA into GLSL code. GLSL was chosen as the target language because it is supported by WebGL, which allows web apps access to the GPU. This is likely why GLSL was also chosen as the input language for Shadertoy. I♥IQ implements its scene output with Three.js [27], which in turn uses WebGL. Thus the generated code is run directly on the GPU. We use Three.js to render a plane with the same dimensions as the output window and apply the fragment shader to that plane. The plane is rendered orthographically so it takes up the entire output window without any perspective distortion.

The GUI for the sliders was implemented using little-gui [28], which is the same library that the authors of Three.js use for the examples on their website. The values of these sliders are passed as uniforms to the fragment shader. These uniforms can be seen in use in figure 4.3. Their naming adheres to the hierarchy described in the design section. For example, a uniform might be named torus_1_transform_X. These can be vectors or scalars. Values from a vector's component sliders are packed into a vector object before being passed to the fragment shader.

5.2 Code Generation

The text in the scene code block is interpreted as TOML. TOML [29] was chosen for its usability and minimal syntax. Unlike JSON, it supports multiline strings, which is important for declaring transformation matrices because the parameter declarations are on separate lines.

The GLSL code output by I♥IQ is injected into a fragment shader, which is run for

every pixel on the screen. The general architecture of this fragment shader is hard coded. A ray from the camera to the pixel is generated. The camera is placed at (0, 0, -1) and the virtual screen is a plane on the x and y axis at z = 0. The ray is passed to the intersection function to check whether it intersects with a shape, and if it does then the shape's material function is called and the resulting color is returned.

The distance from the origin of the ray (the viewer) to the intersection is calculated for each shape and the minimum distance, that being the intersection closest to the viewer, is returned. More precisely, an intersection struct is returned, which contains other relevant information, such as the distance to the intersection, the position of the intersection, and the normal of the surface at that intersection. It also includes whether the intersection was valid. An intersection is invalid if the ray did not intersect any shape before meeting an exit condition. Finally, the intersection struct includes the index of the shape that was intersected. The shapes are indexed in the order that they appear in the scene code block. This index is then used to determine which material to apply. The material selection is implemented as an if-ladder. Inside each if statement, the arguments are set to the corresponding slider uniforms, and the appropriate material function is called. The shape instance index is then used as a key to the if-ladder. An example of one of these if statements can be found in figure 4.3.

Ideally, the intersection function would be generalized such that it could take a shape type. Then a parent function would call this intersect function for every shape instance and return the intersect struct with the smallest intersection distance. However, GLSL doesn't support function pointers, meaning that different SDFs and their respective gradient functions could not be called in a generic way. One solution would have been to make an array for each type of shape holding the shape instance's transformation matrices. Then, for each shape type, run the sphere marching algorithm over the instances in the array. However, an unrolled loop (generating code for each shape instance) seemed like an easier and more readable solution, so that is what we decided to go with. The intersection function runs the sphere tracing algorithm for each shape instance and keeps track of the minimum

distance intersection after each check.

5.3 Sphere Tracing

The intersection function is implemented using sphere tracing. Before executing the sphere tracing algorithm, the ray is transformed into the shape's object space by applying the inverse transformation matrix which is defined in the scene block. If there is an intersection, the point and normal on the surface at the point of intersection are moved back into world space by applying the transformation matrix. To transform the normal into world space, the inverse transpose of the transformation matrix actually needs to be applied [30].

Sphere tracing is an efficient ray marching algorithm that leverages SDFs. At each iteration, the ray marches forward by the distance returned by the SDF.

There are certain necessary numerical constraints. There is the maximum distance to trace (MAX_DIST), the minimum distance before the ray is considered to intersect with the shape (MIN_HIT_DIST), and the maximum number of iterations before giving up (NUM_ITER). These are set somewhat arbitrarily to 100, 0.01, and 32 respectively. If the maximum distance or the maximum number of iterations is reached, the algorithm exits with an invalid intersection. If the absolute value of the SDF's output goes below the minimum hit distance, then the function exits with a valid intersection. For an example of a sphere tracing algorithm generated by I❤IQ, see figure 5.1

5.4 Speeding up IHLA

The parser used for I❤LA is Tatsu [31], a Python library for building a compiler. However, it is slow when parsing several modules, especially since sometimes two parsing passes are required. We optimized and compiled Tatsu using Nuitka [32], and achieved roughly a 2x speedup.

```
p = inverse(torus_1_transform(torus_1_input).ret)*vec4(ray.p, 1.0);
d = vec4(inverse(mat3(torus_1_transform(torus_1_input).ret))*ray.d, 0.0);

total_dist = 0.0;

for(int i = 0; i < NUM_ITER; i++) {
    vec4 sect = p + d*total_dist;

    torus_input _input;
    _input.p = vec3(sect);
    _input.t = torus_1_shape_t;

    //the SDF
    float dist = torus(_input).D;

    if(abs(dist) < MIN_HIT_DIST) {
        torus_1_ret.valid = true;
        torus_1_ret.t = total_dist;

        torus_1_ret.pos = (torus_1_transform(torus_1_input).ret*sect).xyz;

        vec3 norm = grad_torus(_input);
        torus_1_ret.norm = (transpose(inverse(mat3(torus_1_transform(torus_1_input).ret)))*norm.xyz);
        torus_1_ret.norm = normalize(torus_1_ret.norm);

        break;
    }

    if(total_dist > MAX_DIST)
        break;

    total_dist += dist;
}

if(torus_1_ret.t < best.t)
    best = torus_1_ret;
```

Figure 5.1: A sphere tracing algorithm generated by I♥IQ.

Chapter 6: Gallery

This section contains three figures that exemplify the use of I❤IQ.

The material depicted in figure 6.1 implements the Phong reflection model whose equation can be found on Wikipedia [33]. This implementation assumes there to be only one light source. The great number of parameters in this material was the inspiration for the use of structs as inputs and outputs.

Figure 6.2 depicts two tori, each with a different material. The torus on the left is lit with a diffuse shader, and the torus on the right is lit with a toon shader. This is an example of the use of multiple materials in a scene. The toon shader leverages the floor function to discretize the levels of brightness, which, like in the diffuse shader, is a function of the surface position, surface normal, and light position.

Figure 6.3 depicts two interlocking box frames. It would have been difficult to position the frames in this way through non-interactive means, be it via a closed-form solution or by testing values manually.

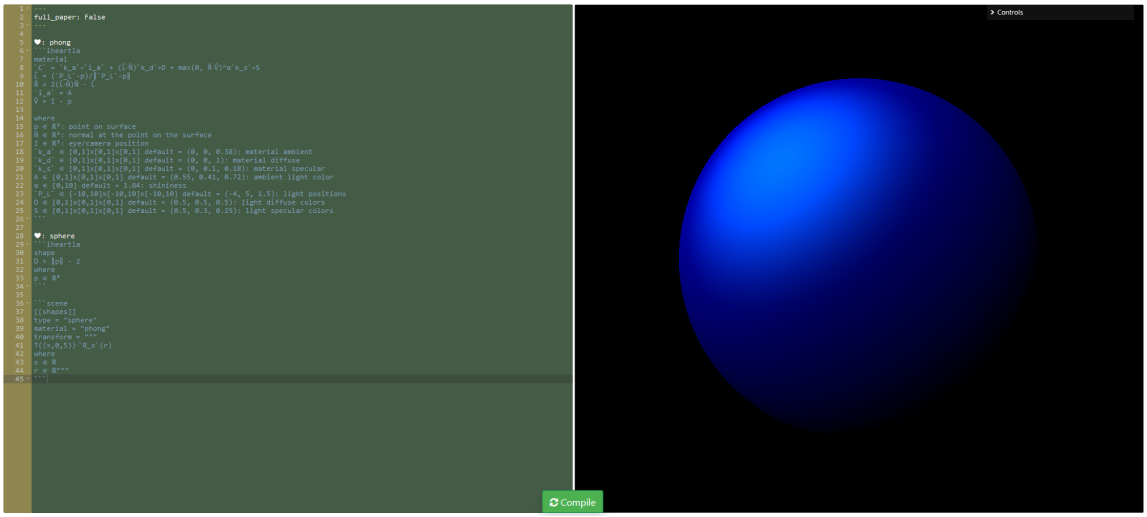

Figure 6.1: Phong material.

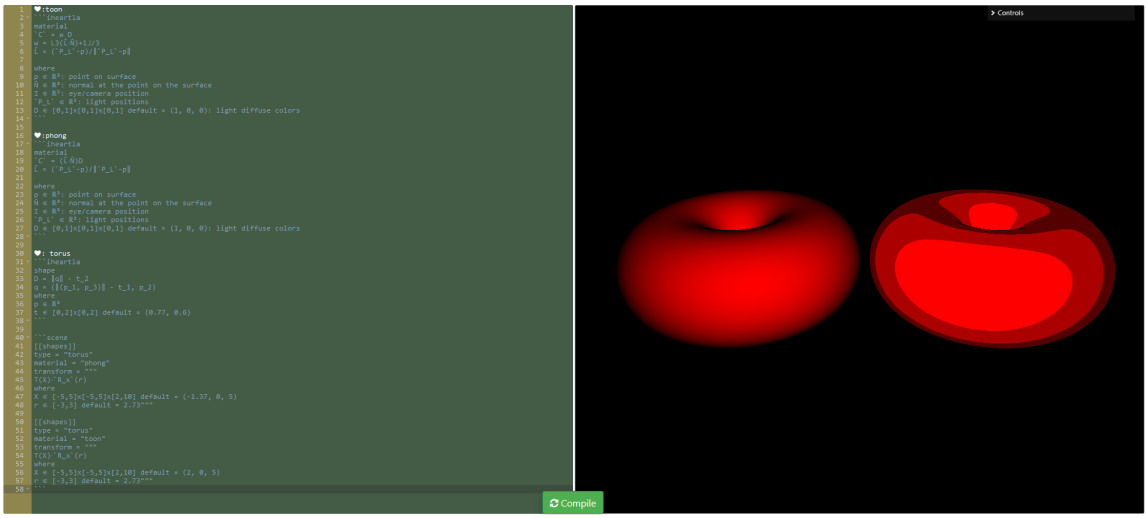

Figure 6.2: A diffuse and toon material.

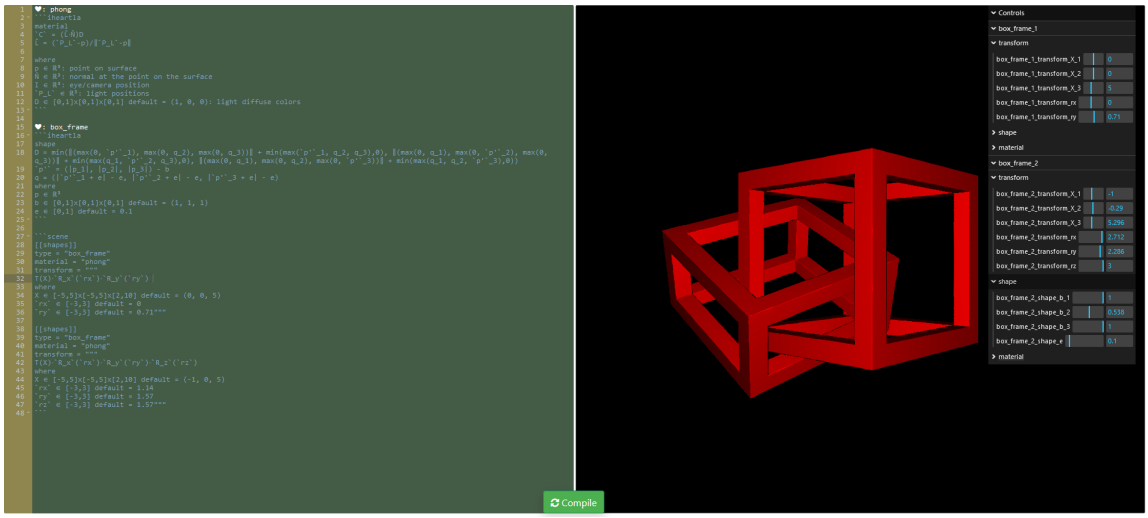

Figure 6.3: Interlocking box frames. The SDF for a box frame was taken from Inigo Quilez' website [24].

Chapter 7: Future Work

7.1 The Scene Code Block

Allowing users to name each shape instance in the scene code block is preferable to just enumerating them, like I❤IQ does now. This would allow users to attribute semantic meaning to the shapes' names instead of having to remember which numbered shape is which. Support for names would not force users to name every shape instance. I❤IQ could just enumerate the shapes which have no name.

Additionally, allowing the user to pass arguments into the material and shape declarations for each shape instance would be a good feature. As of now, the free parameters of each material and shape is instantiated with the same respective global default values, despite each shape instance having its own transformation function. For instance, if the default color of the Phong material is blue, every shape instance using the Phong material will start blue. It is impossible to completely describe a scene because of this limitation. The user would need to tweak each shape instance's shape and material parameters to get the exact scene that they want.

It would also be a good idea to allow users to define global light positions in the scene code block. Then, the light positions would become a reserved parameter for material modules. Right now, light positions are regular free parameters, and are thus different for each shape instance. Most likely, the user would want a change in the light position to affect the whole scene. Users would still be able to define per material lights if they choose.

7.2 Animation, Interaction, and Versatility

Support for time and mouse input as parameters is important. Animation and allowing users to interact with shaders are staples of the genre (see Shadertoy [21]). Noise, which also sees heavy use in fragment shaders, should be supported as well. GLSL itself doesn't natively support noise, so that would be another motivation for I❤IQ as a usable prototyping tool. Support for interacting with matrices would also be a good feature. One way to do that would be to print the matrix in the control panel and then allow users to click drag elements in the matrix up or down to increase them. This would however likely mean implementing our own control UI. It would also be nice to evaluate expressions in the bounds and default declarations.

Furthermore, we also want to allow for embedding scenes into an interactive document. That is after all what H❤rtDown was designed for. There would be some option for whether to generate a full-screen scene or to generate a document and place it inside. The scene would be placed wherever the scene code block was defined in the Markdown. Multiple scenes could be defined and placed in the document this way.

7.3 Implementation Improvements

7.3.1 The Editor

The editor used by H❤rtDown, Ace [34], struggles to handle Unicode characters appropriately. Certain characters, such as the hat character used to denote normalized vectors, cause the cursor to appear to the right of where it actually is. Unicode characters are core to the I❤LA philosophy, so it is important to support them. This can be solved by either switching the web editor to something else like CodeMirror [35], which apparently does not have this problem, or by allowing users to use a local editor of their choice.

The latter could be a better direction. The ergonomics of the editor is not the focus of this project, so it is probably best to practice modularity and focus on the usability of the syntax and the interactive scene. An architecture like that of Streamlit [15] might work

well, wherein the user modifies a source file locally, and the web-app updates when a change is detected in the file. This way, users could use whatever editor they want. Additionally, having an editor window and a scene window allows users to organize the two however they see fit. They could put the scene window on another monitor, for instance.

However, by supplying our own editor, we can provide immediate support for replacing character expressions with Unicode characters. For example, \R is automatically replaced with $\mathbb{R}$ in the current editor. This support would be lost by having users use their own editor. We could amend this by providing a Visual Studio Code extension that does this or by detailing how to set system-wide text-substitution rules in the documentation.

7.3.2 Backend

We achieved some speedup by compiling Tatsu [31] with Nuitka [32], but it still takes too long to compile. We want the compile to feel almost instantaneous. Switching to something like Tree-sitter [36] will likely achieve that.

Bibliography

[1] J. van Dongen, "Interior mapping: rendering real rooms without geometry," Sep. 2018. [Online]. Available: https://www.gamedeveloper.com/programming/interior-mapping-rendering-real-rooms-without-geometry

[2] S. de Rochefort, "Why the bottles in half-life: Alyx look so dang good," Jan. 2021. [Online]. Available: https://www.polygon.com/videos/2021/1/6/22213232/half-life-alyx-liquid-bottle-shaders

[3] "Demoscene: The underground art of real-time," Oct. 2019. [Online]. Available: https://blog.siggraph.org/2019/10/demoscene-the-underground-art-of-real-time.html/

[4] I. Quilez, "The collatz conjecture and fractals," Sep. 2016. [Online]. Available: https://youtu.be/GJDz4kQqTV4?t=300

[5] ——, "Painting a character with maths," Nov. 2020. [Online]. Available: https://www.youtube.com/watch?v=8--5LwHRhjk

[6] "Desmos," 2023. [Online]. Available: https://www.desmos.com/

[7] Y. Li, S. Kamil, A. Jacobson, and Y. Gingold, "Heartdown: Document processor for executable linear algebra papers," in *ACM SIGGRAPH Asia (Conference Papers)*, Dec. 2022.

[8] ——, "I heart la: Compilable markdown for linear algebra," *ACM Transactions on Graphics (TOG)*, vol. 40, no. 6, Dec. 2021.

[9] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr, S. Tobin-Hochstadt, J. Dias, C. Eastlund, and others, "The Fortress language specification," *Sun Microsystems*, vol. 139, no. 140, 2005.

[10] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017.

[11] Y. He, K. Fatahalian, and T. Foley, "Slang: language mechanisms for extensible real-time shading systems," *ACM Transactions on Graphics (TOG)*, vol. 37, no. 4, pp. 1–13, 2018.

[12] D. Geisler, I. Yoon, A. Kabra, H. He, Y. Sanders, and A. Sampson, "Geometry types for graphics programming," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–25, Nov. 2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3428241

[13] B. Victor, "Tangle: a JavaScript library for reactive documents," Jun. 2011. [Online]. Available: http://worrydream.com/Tangle/

[14] M. Conlen and J. Heer, "Idyll: A Markup Language for Authoring and Publishing Interactive Articles on the Web," in *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. Berlin Germany: ACM, Oct. 2018, pp. 977–989. [Online]. Available: https://dl.acm.org/doi/10.1145/3242587.3242600

[15] "Streamlit • the fastest way to build and share data apps," 2023. [Online]. Available: https://streamlit.io/

[16] "Processing," 2023. [Online]. Available: https://processing.org/

[17] "openframeworks," 2023. [Online]. Available: https://openframeworks.cc/

[18] I. Quilez, "Inigo quilez :: computer graphics, mathematics, shaders, fractals, demoscene and more," 2023. [Online]. Available: https://iquilezles.org/

[19] J. C. Hart, D. J. Sandin, and L. H. Kauffman, "Ray tracing deterministic 3-d fractals," in *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '89. New York, NY, USA: Association for Computing Machinery, 1989, p. 289–296. [Online]. Available: https://doi.org/10.1145/74333.74363

[20] J. Hart, "Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces," *The Visual Computer*, vol. 12, 06 1995.

[21] P. J. Inigo Quilez, "Shadertoy," 2022. [Online]. Available: https://www.shadertoy.com/

[22] Henrik, "Ray trace diagram," Apr. 2008. [Online]. Available: https://commons.wikimedia.org/w/index.php?curid=3869326

[23] Teadrinker, "Ray trace diagram," Feb. 2022. [Online]. Available: https://commons.wikimedia.org/w/index.php?curid=115243749

[24] I. Quilez, "distance functions." [Online]. Available: https://iquilezles.org/articles/distfunctions/

[25] B. T. Phong, "Illumination for computer generated pictures," *Commun. ACM*, vol. 18, no. 6, p. 311–317, jun 1975. [Online]. Available: https://doi.org/10.1145/360825.360839

[26] I. Quilez, "normals for an sdf," 2015. [Online]. Available: https://iquilezles.org/articles/normalsSDF/

[27] "Three.js," 2023. [Online]. Available: https://threejs.org/

[28] "lil-gui," 2023. [Online]. Available: https://lil-gui.georgealways.com/

[29] T. Preston-Werner, "Toml: Tom's obvious minimal language," 2023. [Online]. Available: https://toml.io/en/

[30] "How to transform surface-normal vectors." [Online]. Available: https://canvas.northwestern.edu/files/2006490/download?download_frd=1

[31] J. Añez, "TatSu," 2019. [Online]. Available: https://tatsu.readthedocs.io/

[32] K. Hayen, "Nuitka the python compiler," Mar. 2023. [Online]. Available: https://nuitka.net/index.html

[33] "Phong reflection model," Jan. 2023. [Online]. Available: https://en.wikipedia.org/wiki/Phong_reflection_model

[34] "Ace - the high performance code editor for the web," 2023. [Online]. Available: https://ace.c9.io/

[35] "Codemirror," 2023. [Online]. Available: https://codemirror.net/

[36] "Tree-sitter." [Online]. Available: https://tree-sitter.github.io/tree-sitter/

Curriculum Vitae

Henro Kriel graduated from James Madison High School, Vienna, Virginia, in 2019. He received a Bachelor of Science from George Mason University in 2022. He plans to work as a GPU Software Engineer at Intel after he graduates with his Master of Science.