Electronics and Computer Science
Faculty of Physical Sciences and Engineering
University of Southampton

John J. S. Hunter

September 10, 2024

Efficient Realtime Rendering of Complex Closed Form Implicit Surfaces Using Modern RTX Enabled GPUs

Project supervisor:
Kirk Martinez

Second examiner:
Sasan Mahmoodi

A final report submitted for the award of
BSc Computer Science

1 Abstract

Implicit surfaces are a form of defining shapes using mathematical functions instead of triangle meshes. While they tout many benefits over meshes, they have a major downside in that they are very hard to rasterize.

This paper outlines a method of rasterizing signed distance field (SDF) based implicit surfaces by using mesh shaders to generate a close hull of cuboids around the surface, with a pruned version of the distance function specific to each cuboid, then using specialised fragment shaders to sphere trace the final short distance.

This method will be shown working alongside traditional triangle based rendering to produce a mixed format image, and then it will be benchmarked against a standard brute force method. It is found that the brute force method is still more performant for many reasons, but the method outlined in the paper produces a more accurate image.

Statement of Originality

I, John Hunter, declare that this report, ‘Efficient Realtime Rendering of Complex Closed Form Implicit Surfaces Using Modern RTX Enabled GPUs’ and the work presented in it are my own. I confirm that:

- I have read and understood the ECS Academic Integrity information and the University’s Academic Integrity Guidance for Students.
- I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.
- I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.
- I have acknowledged all sources, and identified any content taken from elsewhere.
- I did all the work myself, or with my allocated group, and have not helped anyone else.
- The material in the report is genuine, and I have included all my data/code/designs.
- I have not submitted any part of this work for another assessment.
- My work did not involve human participants, their cells or data, or animals.

Signed: ____________________________

Date: ____________________________

2 Acknowledgements

This project uses code written by Inigo Quilez[1], notably signed distance functions[2] and smooth minimum[3].

I would like to thank all of the maintainers of Rust crates used in this project.

I would like to thank my supervisor Kirk Martinez, for providing support and advice.

I would like to thank various demosceners in the “moonbase” collective for providing GPU based advice and rubber ducks, along with the GLSL code to generate a normal to an SDF and the algorithm used to generate bounding boxes.

I would also like to thank Enfys Castle Roper, Violet Procyon Suchomski, Sian Brookes, Layla Sausse, Daniel David Beynon and many others for being rubber ducks and providing machines to test the project on.

Finally, I would like to thank Gray Wood for wasting my time in the most enjoyable way.

[1] See also: [1]

Contents

3 Introduction

Since the 1990s and earlier, 3D graphics that are rendered on GPUs (Graphics Processing Unit) have usually been made of small triangles. This is because triangles as a shape have multiple useful properties, notably being always non-convex and planar. These combine to create something which is easy to render with simple algorithms.

However, triangles have downsides also - most obviously that as planar shapes they cannot represent curved surfaces, only approximate them. Common approaches to this problem involve increasing the number of polygons around curved surfaces, but this increases both modelling and rendering complexity.

Implicit surfaces are a different way of defining shapes. Instead of providing an exact boundary of the shape, they provide a mathematical function, which given a point in 3D space returns whether or not the point is inside the shape. In this way, implicit surfaces do not exist until observed.

Implicit surfaces are useful for various reasons, including that as a continuous function they can provide infinite resolution (given that the input point in 3D space also has infinite resolution). This means that when defined as an implicit surface a curved surface can truly exist as a curved surface, rather than an approximation.

They can also perform Constructive Solid Geometry (CSG) operations with relative ease, something that triangle based meshes are very cumbersome at. CSG is often used in Computer Aided Design (CAD) operations as it guarantees unambiguous inside-out checking, useful for engineering applications to confirm that shapes can be manufactured and will work as specified. In the context of video games it can be used for collisions, for example to check if the player is touching or inside the floor or walls. The Source Engine by game developer Valve uses CSG for levels for this reason, and can therefore easily confirm a level is "sealed" (with no holes from the inside to outside) at compile time and if a player is out of bounds during runtime.[4]

Unfortunately, implicit surfaces are not without issue either, as due to their implicit nature they are much harder to render efficiently to a 2D screen than triangles. Often implicit surfaces will be converted to triangles before rendering via various methods, such as marching cubes. While this does allow CSG manipulations of the shape, this brings back the issue of having finite resolution. It is also difficult to do well algorithmically, often producing jagged edges and discontinuities. To this end, "retopologising" of a model is often done by hand by the artist rather than automatically by the modelling software, and is both time consuming and nigh impossible to be performed at runtime.

Another common approach to rendering implicit surfaces is sphere tracing, a variant of ray marching, where Signed Distance Functions (SDFs) are used. In this method the function used by the surface will return the distance to the closest point on the surface, and the sign of the distance indicates if the point is inside or outside. It can be trivially proven that there exists a sphere around a point in 3D space which does not intersect an implicit surface, of which the radius is the result of the implicit function when given the point. As the shape is not within this sphere, marching the ray forwards by the result each time allows faster reaching of a surface compared to fixed step marching. See figure 1 for reference. This however, like any form of ray marching, is very slow to compute.

Recently Nvidia have released "RTX" graphics cards which have a feature called Mesh Shading. Here, instead of a traditional rendering pipeline a more compute based approach to generating meshes takes place (Fig. 2). This allows much greater

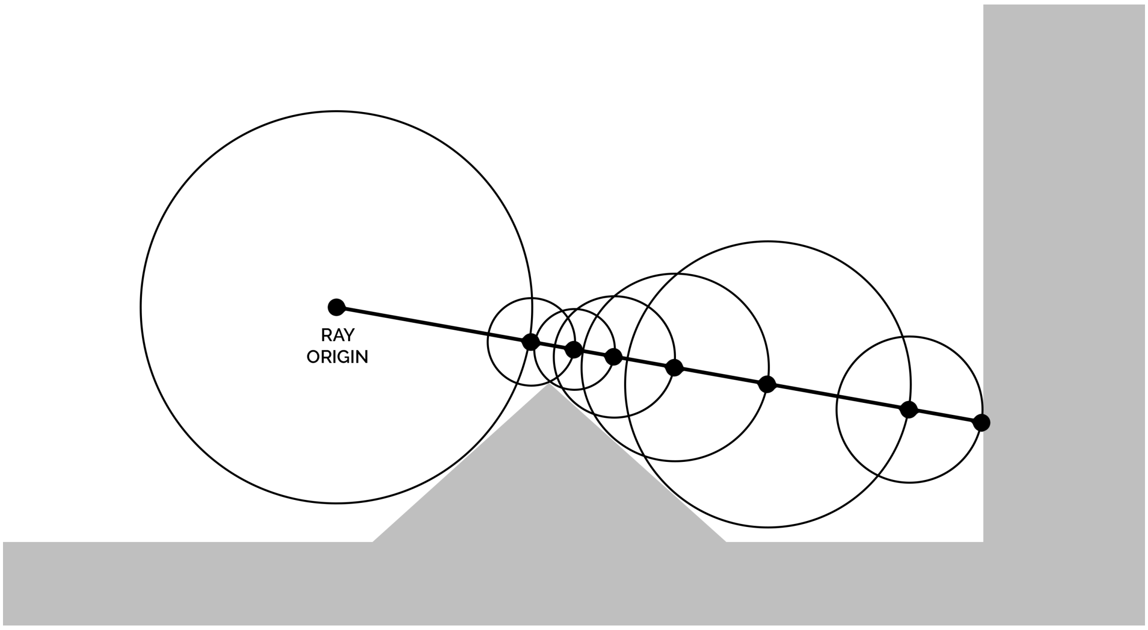


Figure 1: A diagram of a 2D slice of a 3D sphere trace. Notice how more tests are performed as the ray passes the close surface. (Reproduced from [5])

procedural mesh generation than something like tessellation or geometry shaders, similar to what is possible with compute based generation but without needing to pass all the results into memory each time they are generated.

While they are often used, actually creating implicit surfaces is not easy. Often in the demoscene[2] the shapes will be modelled entirely by hand editing GPU shader code. While this may work for the programmers who create demos, artists often use tools which allow you to manipulate the object directly rather than indirectly in code. It also brings the issue of syntax and logic errors into the creation of 3D models, which is an added layer of complexity and difficulty that makes uptake hard.

To this end, the computer graphics programmer known as Ephtracy is creating a program called MagicaCSG[7] which allows modellers to create implicit based models with ease. It uses a familiar user interface (UI) similar to other 3D modelling software, and allows users to see a path-traced version of the model at all times[3]. MagicaCSG is currently closed source and non-free, but it provides a file format for implicit shapes. Most importantly it has a good following of artists who are creating models in this file format.[8]

This paper seeks to solve these rendering issues using a method based on that of Keeter, where a grid of cubes is placed around the surface, and subdivided down with interval checks to generate a close hull of the implicit surface using triangles, which will overencompass it. During this a tape is used to render the meshes, so that they can be edited quickly by the CPU, but also so that each cube can have its own specialised version of the shape, with only the details of the shape which are relevant to that cube. Unlike Keeter, this will be performed using Mesh shading and in 3D space. The aim of this project was to do this in a way that is more performant than

[2] a computer art subculture where people make short real-time rendered “music videos” called “demos”

[3] Due to the closed-source nature of the project it is hard to tell exactly how it is path-traced, but due to observed artefacts it is likely marching cubes is applied.

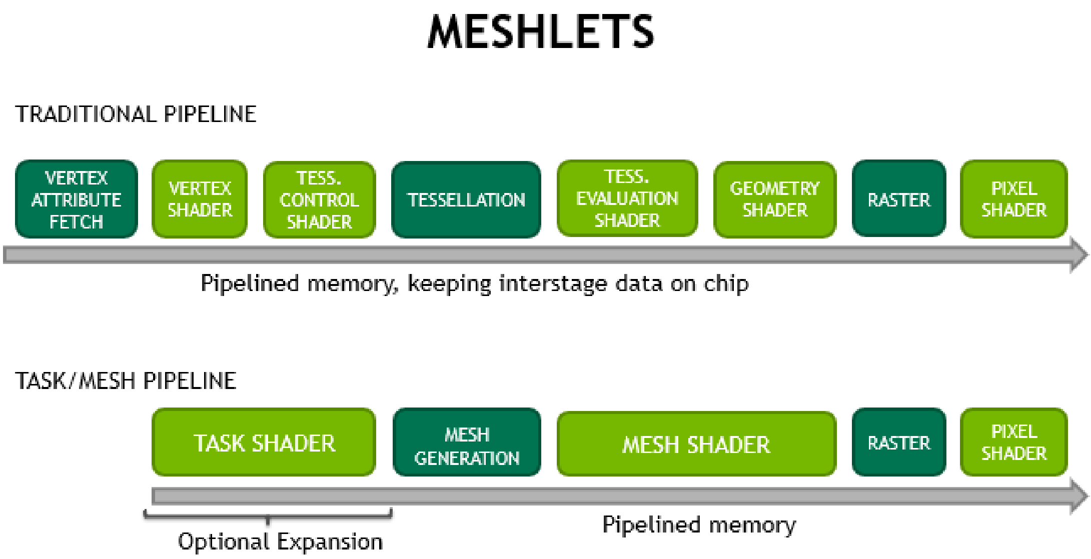


Figure 2: Differences in the traditional versus task/mesh geometry pipeline (Reproduced from figure 4 at [6])

Keeter's method or general brute force rendering of the surface.

4 Review of the Background Literature

Keeter uses a grid of tiles proportional to the screen size to segment the implicit surface on the screen.[9] They also use a special tape shortening algorithm so that the entire surface does not need to be evaluated for each pixel. This method however cannot handle triangles as well as implicit shapes in the same image, and runs as a compute shader. The opcode list is comparatively short, and only operates on single floats.

Keinert et al use a form of relaxed sphere tracing, which oversteps and then corrects itself.[10] This performs marginally better than traditional sphere tracing, but is outperformed by Balint et al.[11]

Balint and Kiglics use quadrics to approximate a surface before rendering, allowing a speedup of up to 100% in rendering static scenes.[11] This method has not been thoroughly tested on dynamic scenes, although they note that simply recomputing the quadrics every frame gives a result which is sometimes faster than Balint et al's previous work, which requires no pre-computation.[12]

5 Design, Specification and Implementation

The implmentation discussed below can be found at `https://git.soton.ac.uk/jjsh1g20/realtime-implicit-rendering`. The camera can be moved by holding right click and using WASD.

5.1 Software

The program is written in Rust and Vulkan, two industry leading and modern platforms. They allow good portability across operating systems, which Microsoft's DirectX[4] does not. OpenGl[5] was not considered as the project was only using the latest technologies. The shader code was written in GLSL. Linux was used as the testing and target OS, but the project has been confirmed to work on Windows machines during testing. Mac OS will not be considered as porting the project to the Metal API[6] is out of the scope of the project.

As an interface to Vulkan, the Rust crate Vulkano is used, as Vulkano allows easy abstractions over the Vulkan API, which saves a lot of development time and reduces errors.[13] Some parts of Vulkan (namely the mesh shader extensions) are not supported in Vulkano, so a modified fork of Vulkano is used.[14]

In order to display UI elements to the screen the immediate mode GUI egui[15] is used. This is similar to other immediate mode GUIs such as imgui, in that it redraws the entire UI each frame, but is written in Rust natively and therefore easier to use. An immediate mode GUI is useful as the program redraws the entire frame each frame, and so the overhead of a callback or state based GUI is unnecessary and is harder to code.

5.2 Opcodes

Models are defined as a tape of opcodes which are all 16 bits in length, along with 7 tapes which contain constants for GLSL types; Floats, Vec2s, Vec3s and Vec4s (shared), Mat2s, Mat3s, Mat4s, and Materials (Mat4s), a tape of precalculated dependencies used for pruning, and an array of descriptions used to specify each individual model. Opcodes are written as a single word starting with `OP`, and often contains the types it works upon, such as `OPAddVec2Float`.

The tapes are executed in a stack based interpreter with 8 stacks, of which 7 are interactable. The instructions will always take between 1 and 6 values from a stack, and then output a single value to a stack (except for duplication opcodes, which are handled specially). The 8 stacks are one for each of the previous GLSL types; most are directly accessed, but the Materials stack can only be accessed as a side effect from other instructions.

The interpreter is run by initialising all stacks to empty, and then pushing the current position of the ray in 3D space as a value to the Vec3 stack. The interpreter is then run until a "stop" opcode is executed, `OPStop`, or the per-model opcode limit is hit and an `OPStop` is implicitly appended.

Upon running the `OPStop` opcode the interpreter stops all operation and a single float value is popped from the Float stack, and returned as the output of the SDF.

There are around 220 opcodes available to execute in the example interpreter, which manage to cover most (if not all) of the GLSL required to define SDF shapes. A full listing with explanation of function is found in the appendix.

[4] A competing graphics API maintained by Microsoft for use in Windows and Xbox consoles

[5] An old, open source API

[6] Apple's proprietary API, used on nearly all and only Apple devices

These tapes are generated at runtime from a vector of Rust structs which contains each opcode, along with each input to the opcode and whether it is variable (read from a stack) or constant (read from a tape). Each instruction is then encoded as follows:

```
<- most significant bit                 least significant bit ->
                        aaaaaabbbbbbbbbb
```

'a' - first six bits, each bit encoding if the corresponding input to the opcode is constant, with 1 meaning to pull that input from the constant tapes instead of the stacks.

'b' - last ten bits, encodes an opcode in a maximum 1024 large opcode set.

For instance, the example object contains an `OPSDFSphere` opcode which takes a constant 0.5 float and a variable Vec3. This will be encoded as the following 16 bit number:

```
1000000011011001
```

Where the first bit is \1" to indicate that the first input is a constant, the second bit is \0" to indicate the second input is a variable, the next four bits are ignored, and the last ten bits are \217" in binary, the number associated with `OPSDFSphere`.

This is written as an array to a buffer to be read back by the GPU.

Along with the 16 bit tape opcode tape, if any constants are read by the encoder they are appended to the relevant constant tape, with Vec3s being Vec4s with a dummy 'w' value. A final dependency tape is generated to be used later by the pruner: it encodes the index of the opcode that uses the of the output of the current opcode, in up to two 8-bit values. For example, if an opcode at position 2 produced a value used by the opcode in position 4, the dependency value at that point would read [4,255], where 255 means no value. If the opcode is a "duplicate" type it will produce two outputs, which may be used by two opcodes, in which case both values are used (e.g. [4,6]). This tape runs in parallel to the opcode tape, and for each value in one there is a corresponding one at the exact same position in the other.

All of the above tapes are generated for each implicit object, and then each tape is inserted compressed together into memory (the Float constants of object 2 immediately follow the Float constants of object 1, etc.) A final scene array is used to link the various other tapes together using a set of pointers into the other tapes, along with a set of 6 bounds (a positive and negative in each dimension, to specify a cube). The bounds are the maximum and minimum values where the shape returns 0 in each direction, and are used as an initial bounding box.

In the reference implementation the vector of rust structs is hardcoded, but it could be set up such that they are generated from models. There is code to read the .mcsg format provided by magicaCSG, but it is incomplete and unusable due to time constraints. Some opcodes will need to be added to the project to facilitate a proper converter, including things like new shapes and deformations such as bezel, and the ability to rotate using matrices.

5.3 Start of Execution

When the program starts it first checks for a suitable GPU. The requirements it looks for in a GPU are a graphics queue for sending draw commands and a transfer queue for sending memory copies. It also looks for a GPU which supports mesh shading, and the ability to draw to a screen. It will prefer a GPU with separate graphics and transfer queue families to help with multitasking, but will use a GPU with the same queue if needed. If multiple GPUs fit the criteria above, they are sorted with a preference to dedicated cards (as they are likely to be faster), and then the first GPU is picked from the sort. Two queues are then created from the families, one for the graphics and one for transfers.

A swapchain is then created using default settings. The chain is created using a FIFO present mode, which causes the frames to not suffer from tearing, but slows down rendering slightly. It would have been preferable to use a more advanced mode such as mailbox, but this was not supported on various systems the project was tested on, and as such scrapped. The swapchain is attempted to be created triple-buffered, however in the case of maximum and minimum limits on sizes it will bow to those first.

Six different shaders are then loaded into Vulkano. These are a Task[7], Mesh, and Fragment shader for the implicit shapes, a Vertex and Fragment shader for the triangle meshes, and a debug Compute shader. The two fragment shaders are based on the same file, as this allows for mild code reuse. There is common code between the two in terms of lighting, as when the position in world space and the normal have been calculated the lighting for both implicit and triangle meshes is identical in order to have models from both look identical in the final image. The combining of the code means the lighting calculations only need to be changed in one location in order for the lighting calculations in all places to update. The compute shader runs the same interpreter as the task and mesh stages, in order to perform debug testing (More information on this shader is in the evaluation section).

A render pass is created with two passes. The first pass is rendered to a multisampled buffer, which has a corresponding depth buffer. This is where the main rendering occurs, with the drawing of the triangle meshes and implicit surfaces. The first pass is then resolved down to a single sample image performing a limited form of anti-aliasing, and then the GUI is drawn above this in a second pass. The multisampling allows some anti-aliasing to occur, however the implicit surfaces have edges not near the actual edges of the triangles they are drawn on, causing them to keep all their aliasing. The triangle shapes however are truly anti-aliased.

A `\gstate"` struct is created to hold all state relating to the GUI, and by extension most of the state relating to the scene itself. This struct is passed back and forth between the GUI code as mutable so both the GUI can access the values to edit them, and the render code can access the values to read and use them. The use of a single struct can help by providing an easy way to check if all the values are the same, such as with the boolean values in the debug section of the GUI.

At this point the default mesh, implicit surfaces and lights are loaded from the executable. The mesh is stored as a Wavefront OBJ file, which is decoded into a vertex buffer and an index buffer. The use of two buffers like this is efficient as it reduces the amount of data needed to be read from video memory each frame. It is

[7] Some citations refer to the "task" shader as the "amplification" shader. This is due to naming differences between DirectX and Vulkan, and they refer to the same stage.

also close to how OBJ stores its models internally, however the normals and vertices need to be duplicated in order to be shown, as they are stored separately and the GPU requires them to be shown together. The file loaded for the CSG is an .mcsg file, but it is overwritten during runtime with a hardcoded example, as the .mcsg loading code is not functional. The lights created are two simple warm and cold lights at opposites to each other to light up the objects as an example. See figure 5 for an example. The meshes, CSGs and lights are stored in three vectors, along with all information needed to render them such as scale, rotation and position.

5.4 Frame loop

The event loop for the window is then started, opening the window. The event loop reacts to requests to draw, resize, and close requests from the operating system. It can also react to files being dragged onto the window, which would be a good way to load OBJ and MCSG files. This is unimplemented in the reference code. It is here that the mouse and keyboard is read to provide camera movement - a simple Euler angle based camera is used, similar to the game engine Unity's editor.

5.4.1 Start of Redraw

In the event that a redraw is requested, a number of actions are performed: A boolean variable keeps track of if the swapchain needs to be recreated. This is needed in the case that the window is resized, or various other odd cases such as minimisation. This involves remaking all of the framebuffers and pipelines to accommodate the new resolution - an expensive operation, but one that speeds up a lot more of the rendering and only takes place rarely. It is here that the GPU code is compiled. The graphics pipelines are set up such that there is backface culling (removal of triangles facing away from the camera) on the mesh rendering, but not the implicit rendering as it will perform its own backface culling in the Mesh shader.

This function is called again in the case that the debug flags in the GUI have been changed, as there are specialisation constants built into the shaders. These creation of constants that change infrequently. Each time the flags are changed the pipeline is recompiled and the shaders rebuilt, but these are normally cached and do not need to be built from scratch.

5.4.2 Uniform generation

Various matrices are generated: a view matrix is used for the camera to move objects so they are in view; a world matrix is used for each object; and a projection matrix is used to take the 3D scene and flatten it into 2D. The view and projection matrix, along with the camera's actual position, are written to a uniform buffer, which allows them to be read from the GPU. The world matrix is instead written to a push constant - these are written into the command lists for the GPU rather than the video memory or GPU facing main memory, allowing them to be updated more frequently. The world matrix is updated after each model is drawn so that the next model is drawn in the correct place in 3D space.

A separate uniform buffer is generated which contains up to 32 point lights, stored as a position and a colour. The intensity of the light is encoded into the magnitude of the colour vector. These are stored in a statically sized 32 element array, with a count beside it, as this is more efficient to generate.

As well as the window-based setup which is rerun if the window is changed, there is an object-based setup which is rerun if the quantity or style of objects is changed. This involves running the encoding process for all the arrays of opcodes mentioned above, and then generating GPU side buffers for that data to be stored in. This is performed by generating a CPU side but GPU accessible buffer which the data is written to, and an identical buffer in video memory, and then submitting a memory transfer request to the transfer queue created earlier. The CPU side buffer is then released.

During this step the bounds for each CSG object are calculated. This is performed by querying the implicit surface once in each direction for a considerable distance away, then subtracting this from the distance to the origin. This works despite the query being similar to a sphere, as from an infinite distance away the edge of the sphere tends towards a plane. The difference of distance from the infinite sphere's origin and the distance of the infinite sphere to the nearest point then tends to the distance from the origin to the farthest point, which when repeated in all 6 directions from the origin along the axes generates a bounding box. This is impractical to be calculated at an infinite distance, and so it is calculated 65536 units away instead.[8] The result is also multiplied by 1.00001, to slightly inflate the box in case of mathematical error, and all calculations are performed at 64 bit precision (whereas the GPU runs at 32 bit). See figure 3 for a diagram.

In order to perform the queries a full interpreter is written in Rust, which is designed to emulate the interpreter on the GPU written in GLSL. This is the first of three interpreters in the project. Instead of interpreting the encoded arrays, it interprets the original structs, allowing it to run with less effort for the CPU. It also has various edits for coding convenience, such as using vectors rather than arrays for stacks. This is different to the GLSL version, which produces mildly slower code that is much more readable and maintainable. The instruction listings are generated in a build.rs file by reading the GLSL instruction listings and converting them. Information regarding the inputs and outputs of each function are written into the comments of the instruction listing GLSL file, and this is used to provide the dependency tape generation and rudimentary type checking during the encode stage.

The buffers are attached to a descriptor set layout, which is one per pipeline. Not all stages need all the buffers (for example the task shader does not read lighting information), but it is easier to bind them all together at once than to try and separate them. Most bindings are sequential, but jumping about indexes is possible and allows for more ease of binding.

5.4.3 Debug fuzzer

In the case that the fuzzer is enabled, it is now calculated. A preset set of 32 implicit surfaces are generated and encoded with static bounds, then sent to the GPU. The compute shader is then run over the data, and the last item on each interpreter stack is then relayed back to the CPU. The CPU waits for this to complete, then runs the CPU side interpreter over the same surfaces, and checks the GPU side's

[8] 65536 was estimated to be a good compromise between being too close to the shape to be useless and too far away to induce float point imprecision.

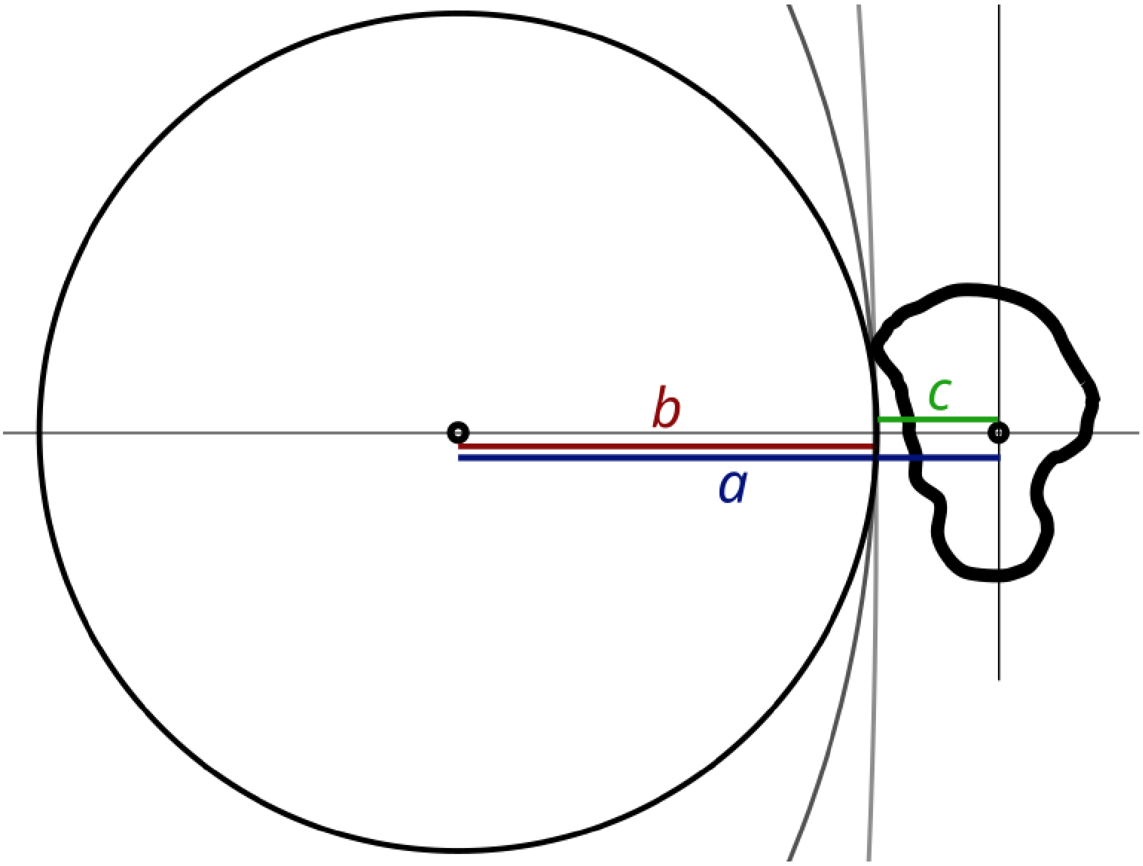


Figure 3: The calculation of the bounding box. If a query is performed at a units from the origin, and it returns b, the bounding box c is calculated to be $c = a - b$. Extra transparent circles are added to show the circle's edge approaching a plane.

results for error. This normally generates a lot of false negatives, as the CPU side is considerably more accurate in results but allows for a preliminary check. Results are then printed to the terminal for actual analysis. This whole process is slow and inefficient, and only enabled in debugging scenarios. It does however generate very useful information and is one of the only ways to debug the GPU side interpreter.

5.4.4 GUI

The GUI is updated by writing all the current `gstate` values to the screen. The camera's sensitivity and speed can be changed. For each object the position, rotation, and scale are written, along with a button to remove that object. Lights have no rotation or scale, but instead have a colour value which can be changed, and there is an ability to add a default light to the scene. The FPS scale is updated every frame with the value of $1/(timeforframe)$, which is usually the refresh rate of the monitor. There are five toggleable values which change how the algorithm itself works under "Debug", which disable various parts of the project to allow testing of one part at a time.

5.4.5 Command buffers

The command buffer for the main draw routine is then generated. The image is first cleared to a grey, and the depth is cleared. The triangle mesh objects are rendered first and as such their pipeline and descriptor sets are bound. For each mesh the world matrix is calculated and pushed, and then the vertex and index buffers are bound and a draw call is generated. Immediately after the implicit pipeline is bound and a similar process takes place, except that instead of vertex and index buffers a group count of 1 is sent. This is always set up to draw the implicit surfaces indirectly, and as such can be performed in a single call.

The next subpass is requested, and this is filled entirely with a secondary command buffer generated by egui. This renders the GUI on top of the rendered image from before, and uses the most draw requests so far.

This command buffer is built and is then submitted to the GPU, to execute as soon as the previous frame is completed. The synchronisation structs are then updated by vulkano and the loop repeats.

5.4.6 GPU triangles

On the GPU side, a variety of things will happen for each draw call. The triangle vertex shader is very simple, transforming the vertices based on the matrices sent in the uniforms, then sending a normal and position to the fragment shader. The fragment shader consists of a single call to the shared shading code, which loops over each light and calculates how close the normal is to pointing at it to produce an intensity, and is then accumulated together and output.

5.4.7 GPU Implicit

The implicit dispatch is more complicated, and takes the majority of the frame. The implicit pipeline follows a mesh shader graphics model, including a Task shader, a Mesh shader, then rasterization, then a Fragment shader. The Task shader and mesh shader run generic compute-like code, and the Task shader invokes a variable amount of Mesh shaders.

The entire pipeline is run twice per object, once for each side - this is due to GPU compute workloads. When threads are executed on a GPU, they are executed in Warps[9]. These warps share a lot of internal structure of the chip, and it's often a good idea to make them run the same instructions in the same order. In order to fit into NVIDIA's 32 wide warp, the Task shader starts by processing 32 cubes[10], and it is run twice to generate the whole 4x4x4 (64) cube grid.

5.4.8 Intervals

The Task and Mesh shaders both run an interval interpreter. This is a special type of interpreter that takes and works on intervals rather than single values. Intervals are ranges of numbers which define an uncertainty bound on what a value could be, by providing an upper and lower bound on the value. Many operations are easy to perform (especially if they are monotonic, which the majority of the opcodes were), however some had to be hand calculated.

For a vector, the upper and lower bounds are provided component wise, but operations will attempt to take all components into account. This is to help combat the dependency problem, which results from using the same value twice. For example, to calculate the normal of a vector the calculation to be performed is $n/length(n)$, where n is the vector. This is an issue as n here is used twice - to calculate the smallest bound, you would need to minimise the numerator and maximise the denominator, but these are both dependent on n. The code in the interval interpreter goes significantly out of its way to make sure that the dependency problem is solved internally within each opcode - for example, the implementation for `OPNormalizeVec4` is 32 lines long and has 16 calls to length - however some opcodes like `OPSDFBox` and `OPMod` are only partially solved.

The Task and Mesh shaders use the interval interpreter to calculate whether a surface's edge exists within a cube. They are queried with the bounds of the cube as the interval, and the result is checked:

- If both the max and min are negative the cube is inside the shape, and so will never get seen and does not need to continue.
- If both the max and min are positive then the cube is outside the shape, and so will render nothing and does not need to continue.
- If the max is positive (or 0) and the min is negative (or 0) then the cube is on the edge of the shape, and it continues to the next step.[11]

5.4.9 Task and Mesh shading

Both the Task and Mesh shaders follow a similar process:

- The current bounding box is retrieved, and a cube within that is generated based on the workgroup ID.

[9] Warps are NVIDIA's terminology. AMD calls them Wavefronts

[10] The elements of the grid are sometimes referred to as cubes, but they can be any cuboid and are always a scale of the original bounding box.

[11] If the max is negative and the min is positive then something has gone VERY wrong

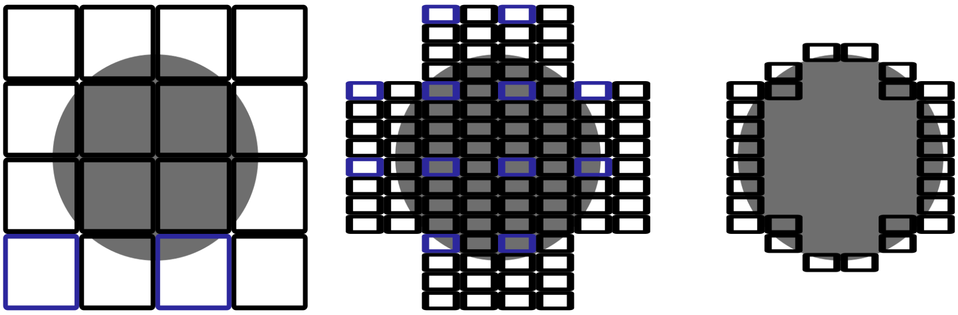


Figure 4: A diagram showing a 2D slice of the hull generation of a sphere. The three stages are Task shading, Mesh shading, and the final product. The blue squares indicate the starting points of each of the shader invocations per stage.

- The current mask state is loaded or initialised.
- The object is queried once only as an interval.
- Based on this the data generated is passed to the next stage or not.

The Mesh shader performs the cube generation twice, as it repeats the calculation performed by the Task shader. The precalculated cube is not sent from the Task shader down, as the amount of data sent between the Task shader to the Mesh shader should be as little as possible to help with performance.

When the Task shader runs, it acquires a unique index into an array using an atomic value. If it is passing data through, it will write at this index into an array to indicate to the Mesh shader which cube it is generating for (as it otherwise has no idea) and the mask generated. When the Mesh shader runs it writes a set of six triangles to the output - either as a single dot to indicate that it should not be processed, or as three squares making half of a cube. A test is performed to calculate which sides are facing the camera, and only those are generated, as the GPU has strict limits on how many triangles can be generated per shader. These triangles are moved and projected, but they are provided with their original 3D positions for later processing in the fragment stage.

The use of the two separate stages, Task and Mesh, allows for compute reuse. After the first Task stage some areas will not need to be considered. If a single stage was used, the GPU would need to continue processing these areas until the final steps - instead the Mesh stage allows the GPU to only consider the areas that have been passed down from the Task stage explicitly. This coarse and fine levels of detail helps the GPU spend time calculating what is actually important. This is illustrated in figure 4.

5.4.10 Tape Pruning

The GPU also runs a pruning process on the Task and Mesh stages. When the interpreters are run they check each instruction against a mask. This mask indicates whether that instruction should be executed or not - if the bit is "0" then the instruction is simply skipped. This allows cubes which are nowhere near certain parts of the object to ignore them, for example cubes near the sphere will not even consider the torus in the default object. If an instruction is skipped, the constants are still read as the constant tapes need to be moved forward anyway.

As the interpreter is stack based, this does not affect the flow of data through the program. Keeter uses a register based interpreter, and as such if an instruction has a different input register to output register the GPU has to replace instructions with move instructions, which does not decrease the number of instructions executed. The use of a stack means that even if an instruction is removed the data is still in the correct location (the top of the stack). The other side to this is that a stack interpreter is inherently slower than a register based one.

The pruning process is performed by keeping an array of values, one for each bit in the mask plus an extra to avoid out of bound memory accesses, which are either 0, 1, or 2. Each value indicates that the position in the tape is some amount of pruned:

- 0 means that the relevant instruction is not to be pruned.
- 1 means that the instruction itself is to be pruned, but the children of the instruction are not.
- 2 means that the instruction is to be pruned and the children should also be recursively pruned.

Here "children" means instructions that generate inputs to the current instruction, and the inputs to those instructions, and so on recursively. When an instruction is marked to be pruned, the function `prunesome` is called, with a mask of inputs to prune. This starts at the beginning of the opcode tape and works forward looking through the dependency tape for any instructions that are outputs to the current instruction. It works forwards, as when it encounters a relevant instruction it can then look into the provided mask of inputs to see if it is to prune or not. If it is to be pruned, it is set to 2 in the pruning array. The instruction itself is then set to 1 in the pruning array. In the case that the instruction found has two outputs, the instruction is incremented 1 for each output that is the current instruction. This means that if only one output is disabled, the instruction will disable only itself - in the case of the duplicate that makes it a passthrough, allowing one output to retain an input but disabling the other. However if both are disabled it will disable the children also, causing a normal reaction.

When a stop instruction is run the `pruneall` function is called, which runs in reverse. This starts at the end of the tape (or rather the stop instruction) and works backwards, reading the pruning array and editing the mask. The extra value at the end of the array is there to always be 0, to give a value to the implicit stop instruction that is sometimes needed.

For each value, the mask is disabled if either its own pruning array is above 0, or the pruning array of the parent is above 1. If the parent is above 1, the child is set to 2 to propagate it. In the event that a duplicate is hit, a complicated switch statement is executed which decides the correct actions based on the two output's value in the pruning array and the instruction's own. The switch will: disable the instruction entirely if it is 2 in the array or both of the parents are; set itself to 1 if only one parent is (and other edge cases); or otherwise ignore it. At the end of this process the mask is passed to the next stage either by shared task payloads or by writing out to main memory.

5.4.11 Sphere Tracing

At this point all the information required by the fragment shader is calculated, and the final sphere trace can be performed. The pixel knows its own position in 3D space because the shader will linearly interpolate the position values from the Mesh shader across the surface. This will be the ray start point. The ray direction is found by subtracting the ray's position from the camera's position, and then normalising it. Here the world matrix is also multiplied in - this allows the entire object to be moved and scaled, where the internal coordinate system stays still but the external one moves around it. This is a simple way of implementing the translation, rotation, and scaling.

Then the sphere trace itself occurs. The ray can start at the surface rather than the camera as it assumes there is nothing between the two for optimisation. It then performs up to 50 steps into the object, querying the entire interpreter each time and stepping forward the result. Most rays will be very close to the surface, and so the stepping forward will last less than 50 steps. The ray stops once the distance to the object is less than some epsilon (0.001), and returns the distance it travelled from the ray start and the final position. There is also a hard farplane, which is calculated as the farthest distance between two vertices of the cube. If the ray passes the farplane it is automatically terminated, as it must then be outside the cube and there is no point continuing to trace it.

A normal is then calculated for the surface of the object at this intersection. It is performed numerically as this allows the shape to take many more complex forms and is easier to implement. It is however sometimes slower than an analytical calculation, and slightly less accurate. An analytical solution might be written for this project, but it would require a dedicated normal interpreter to just calculate the normal. It also requires that the SDF can be symbolically differentiated, which may not be feasible.

This normal is passed with the world space position into the exact same lighting calculation as the mesh triangle based fragment shader, generating a final colour. The shader also corrects the depth value, so that when it is written to the depth buffer it correctly intersects the triangle geometry's values.

5.4.12 Materials

Contained within the implementation is some code relating to materials. This code is not finished due to time constraints, but the concept of how they would function is fully formed. Below details how this would work.

The scene function in the third, final interpreter takes a single extra boolean value to enable materials. When this is set, along with the rest of the processing, material processing and calculation is performed. Instructions which start `\OPSDF"` are SDFs. As well as returning a float distance, in the event of materials being enabled they push the material of the SDF to the material stack. This stack is inaccessible by normal means, only used as a side effect of other instructions. This material stack could contain any value, but in the example implementation they are Mat4s. Most instructions will ignore this stack, but when an instruction which has `Material` in the name is encountered and the material processing is enabled it will perform calculations on this stack also. The most common use is `OpMinMaterialFloat`, which as well as performing a minimum of two floats, will take two materials from the stack and write back only the one which corresponds to the float which was lower. At the end of processing, the top material on the stack will be the material of the surface at that point in the object, and the values within it can be used to calculate lighting and shading.

One advantage of having materials as actual structs or matrices is that they can be interpolated. Take `OpSmoothMinMaterialFloat` for example - instead of returning the material as is, if the surface is between the two SDFs, the material is linearly interpolated between the two inputs. This allows for materials which flow between individual primitives.

6 Testing Strategy and Results

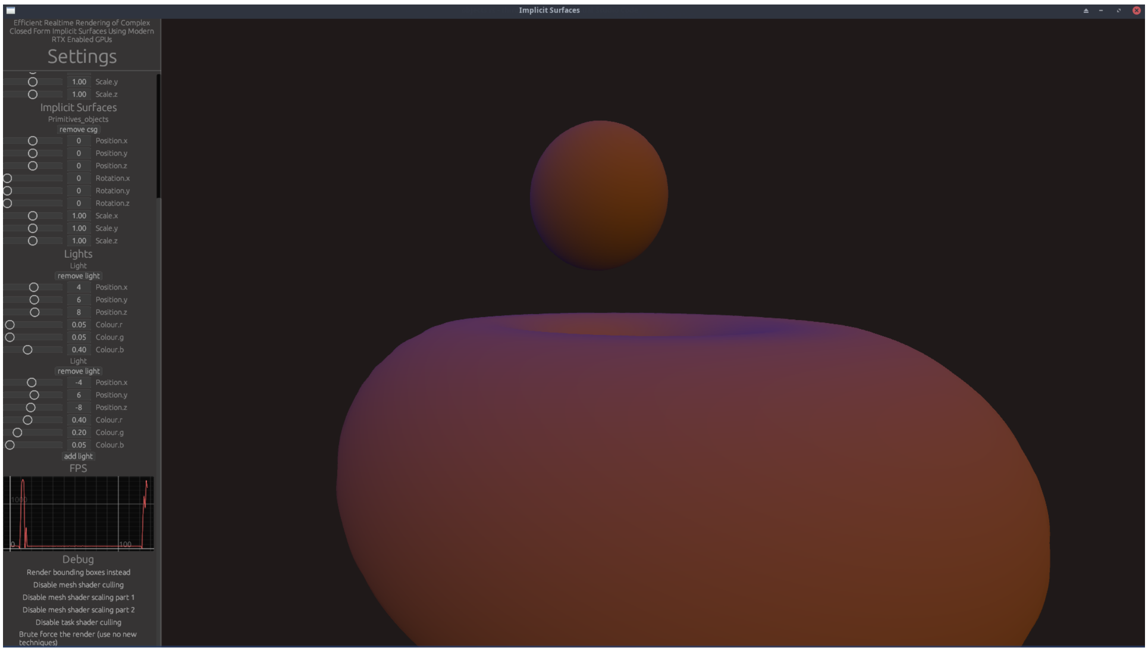
The project was set up with an example scene of an implicit sphere and a torus, along with a mesh based rabbit. The camera was moved in towards the object until the framerate started to drop, and then the graph was read over the course of 3 seconds on both normal execution and with brute force enabled.

The computer used for testing was running Endeavour OS (Arch Linux) with latest updates as of 27-04-2023, with an MSI GeForce RTX™ 3070 GAMING X TRIO GPU connected at PCIe x16 Gen 3 speeds running the Nvidia 530.41.03 drivers and an Intel i7-6700 CPU running linux 6.2.12. It was compiled in release mode with rustc 1.71.0-nightly (1c42cb4ef 2023-04-26).

Average framerate for both modes from various angles (frames per second):

	Framerate	
Angle	Normal	Brute Force
1	38	43
2	28	37
3	58	60
4	38	52

Not once does the method detailed in this paper achieve a higher framerate than the simple bounding box.



Efficient Realtime Rendering of Complex Closed Form Implicit Surfaces Using Modern RTX Enabled GPUs

Settings

1.00 Scale.y
1.00 Scale.z

Implicit Surfaces

Primitives_objects

remove csg

0 Position.x
0 Position.y
0 Position.z
0 Rotation.x
0 Rotation.y
0 Rotation.z
1.00 Scale.x
1.00 Scale.y
1.00 Scale.z

Lights

Light

remove light

4 Position.x
6 Position.y
8 Position.z
0.05 Colour.r
0.05 Colour.g
0.40 Colour.b

Light

remove light

-4 Position.x
6 Position.y
-8 Position.z
0.40 Colour.r
0.20 Colour.g
0.05 Colour.b

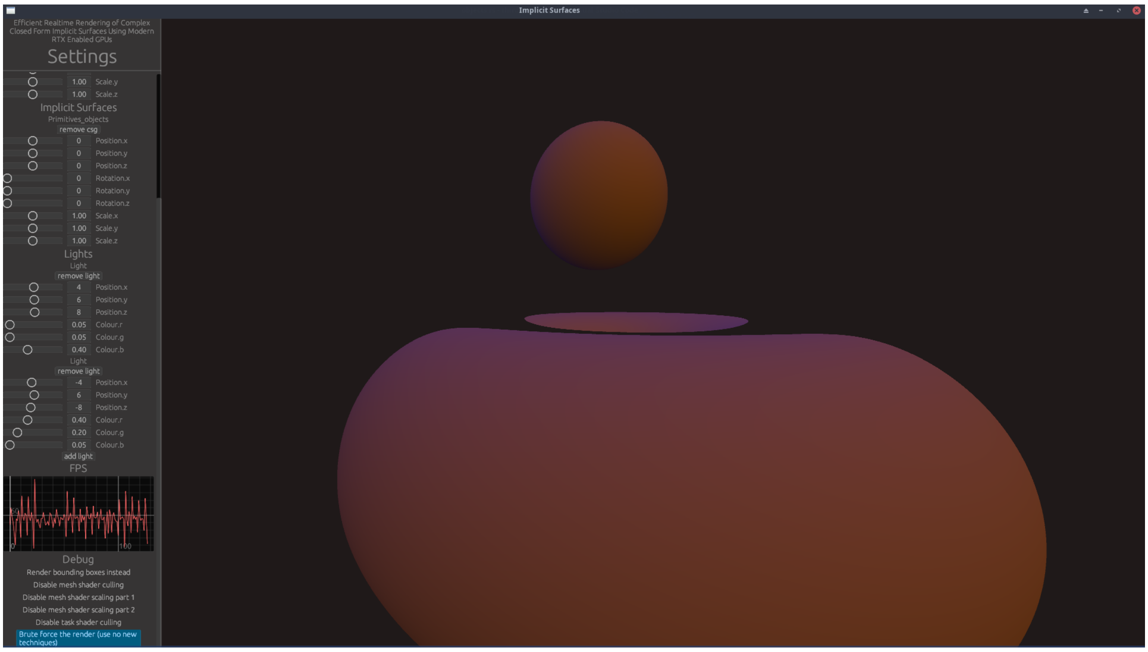
add light

FPS

Debug

Render bounding boxes instead
Disable mesh shader culling
Disable mesh shader scaling part 1
Disable mesh shader scaling part 2
Disable task shader culling
Brute force the render (use no new techniques)

Figure 5: Testing angle 3, with normal rendering



Efficient Realtime Rendering of Complex Closed Form Implicit Surfaces Using Modern RTX Enabled GPUs

Settings

1.00 Scale.y
1.00 Scale.z

Implicit Surfaces

Primitives_objects

remove csg

0 Position.x
0 Position.y
0 Position.z
0 Rotation.x
0 Rotation.y
0 Rotation.z
1.00 Scale.x
1.00 Scale.y
1.00 Scale.z

Lights

Light

remove light

4 Position.x
6 Position.y
8 Position.z
0.05 Colour.r
0.05 Colour.g
0.40 Colour.b

Light

remove light

-4 Position.x
6 Position.y
-8 Position.z
0.40 Colour.r
0.20 Colour.g
0.05 Colour.b

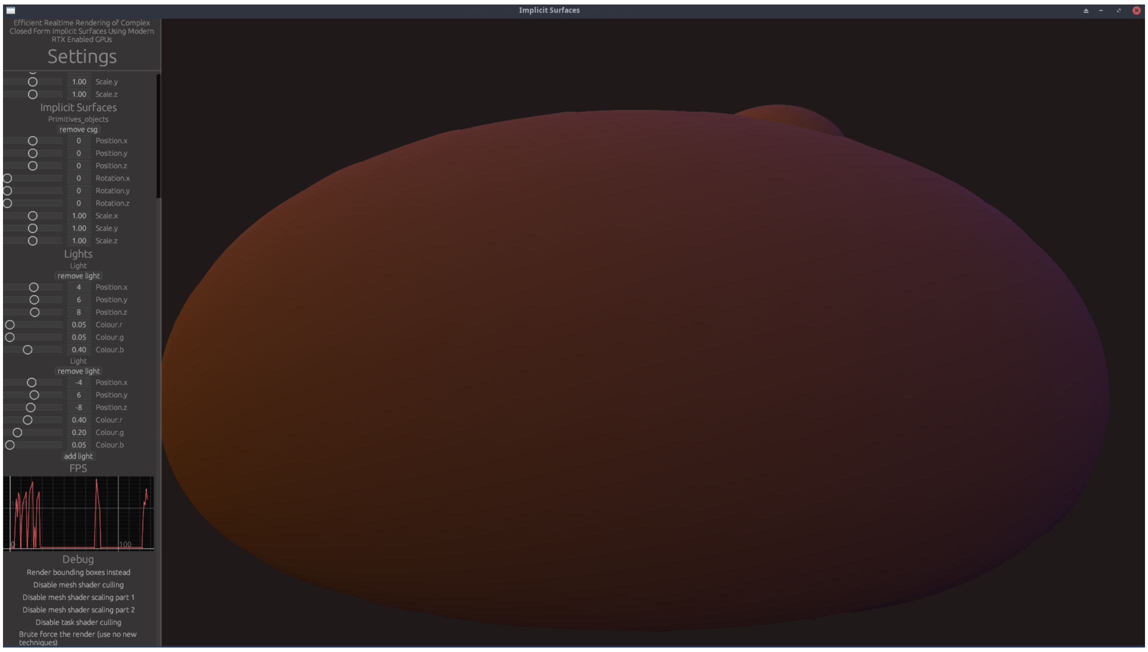
add light

FPS

Debug

Render bounding boxes instead
Disable mesh shader culling
Disable mesh shader scaling part 1
Disable mesh shader scaling part 2
Disable task shader culling
[✓ Brute force the render (use no new techniques)]

Figure 6: Testing angle 3, with brute force rendering



Efficient Realtime Rendering of Complex Closed Form Implicit Surfaces Using Modern RTX Enabled GPUs

Settings

1.00 Scale.y
1.00 Scale.z

Implicit Surfaces

Primitives_objects

remove csg

0 Position.x
0 Position.y
0 Position.z
0 Rotation.x
0 Rotation.y
0 Rotation.z
1.00 Scale.x
1.00 Scale.y
1.00 Scale.z

Lights

Light

remove light

4 Position.x
6 Position.y
8 Position.z
0.05 Colour.r
0.05 Colour.g
0.40 Colour.b

Light

remove light

-4 Position.x
6 Position.y
-8 Position.z
0.40 Colour.r
0.20 Colour.g
0.05 Colour.b

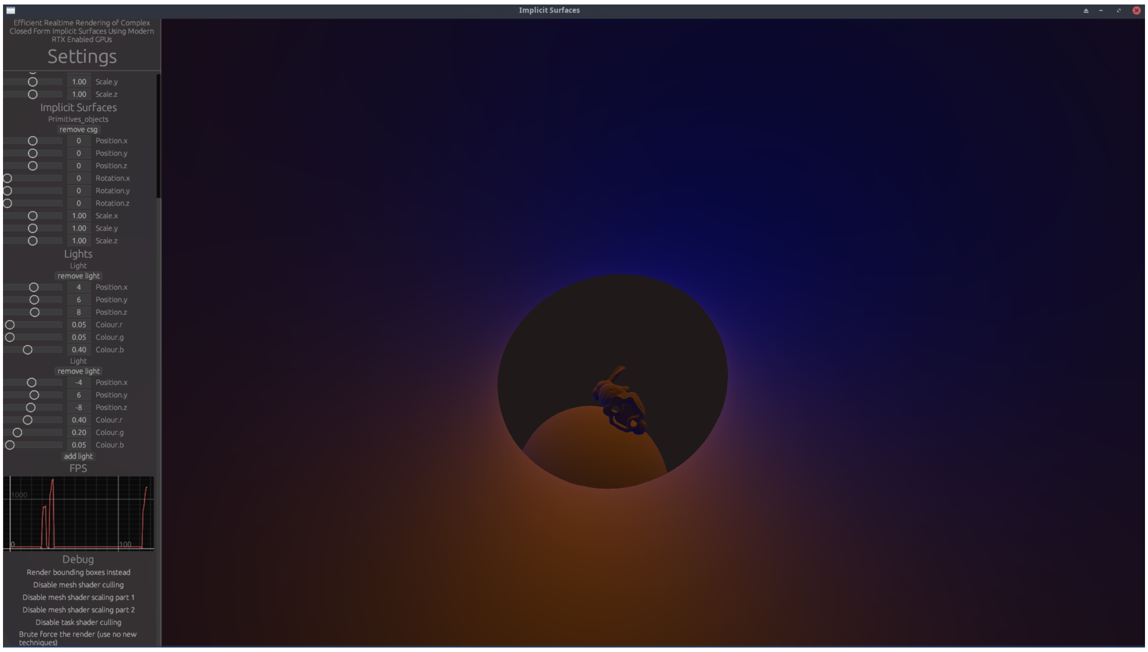
add light

FPS

Debug

Render bounding boxes instead
Disable mesh shader culling
Disable mesh shader scaling part 1
Disable mesh shader scaling part 2
Disable task shader culling
Brute force the render (use no new techniques)

Figure 7: Testing angle 2



Efficient Realtime Rendering of Complex Closed Form Implicit Surfaces Using Modern RTX Enabled GPUs

Settings

1.00 Scale.y
1.00 Scale.z

Implicit Surfaces

Primitives_objects

remove csg

0 Position.x
0 Position.y
0 Position.z
0 Rotation.x
0 Rotation.y
0 Rotation.z
1.00 Scale.x
1.00 Scale.y
1.00 Scale.z

Lights

Light

remove light

4 Position.x
6 Position.y
8 Position.z
0.05 Colour.r
0.05 Colour.g
0.40 Colour.b

Light

remove light

-4 Position.x
6 Position.y
-8 Position.z
0.40 Colour.r
0.20 Colour.g
0.05 Colour.b

add light

FPS

Debug

Render bounding boxes instead
Disable mesh shader culling
Disable mesh shader scaling part 1
Disable mesh shader scaling part 2
Disable task shader culling
Brute force the render (use no new techniques)

Figure 8: Testing angle 4

7 A Critical Evaluation of the Project

7.1 Testing Analysis

In the testing, the fourth angle (Fig. 8) has the worst results. This is due to being from the bottom up, and having most of the object in shot but a large surface near the camera. The brute force version has a close object near the camera which is easy to intersect (the torus). The other version has to render all the cubes with massive overdraw, and so is not very performant.

The brute force implementation, while running faster, has issues itself. Because the ray starts a lot closer to the surface in the normal implementation, the ray iteration count can be a lot lower than in the brute force version in order to correctly intersect. This causes visible errors in the brute force version where the ray is glancing against a surface, which increases the number iterations needed to hit a surface. These rays often fail early, leaving a ring around the torus where you can see the background. The normal version of the iterator still suffers from this, but it is hidden significantly and harder to notice. This issue is visible in angle 3 (Figures 5 and 6)

The brute force implementation also suffers from a major issue, which is that the camera cannot be placed within the massive bounding box. There is a large area above the torus where the bounding box extends but there is no surface, and moving the camera there moves it within the box which makes all the faces face the wrong direction and disappear. The normal implementation fixes this by moving the hull much closer to the surface, but still suffers from this.

Another, small issue is that the normal implementation slightly incorrectly over-inflates the edges of objects. It seems to mostly happen where the faces of the cubes meet the surface of the implicit object. It is unclear what is causing this, and while the brute force version does suffer from this issue the single cube makes it much less obvious.

7.2 Programming Issues

In the original project brief I stated that I would use Ash as an intermediary state for accessing Mesh shading, as Vulkano did not support it at the time. This was unusable, as while there was inter-operation between the two apis, they did not allow creation of command buffers safely between the two which was needed. Instead, the code uses a special branch of Vulkano itself where I have manually implemented all the necessary code to use Mesh shading. Because this is an extremely up to date branch, the interop code between Vulkano and egui has also had to be updated manually, and also sits in a separate special repository on my GitHub.[14]

When designing the project I originally intended to pass the mesh pruning data between the stages using only shared data. This proved difficult, as GPUs have quite strict limits on how much data can be passed this way, and it is in the range of about 16kB. This is not enough to hold all the data needed for a true pruning whilst also having reasonably complex shapes. Instead, the Task shader passes data to the Mesh shader via shared cache, but between the Mesh and Fragment stages the data is written and read from an SSBO mapped in GPU memory.

7.3 Tape Issues

A major issue I encountered was that multiple adjustments were needed to the tape. When the tape was first written, it followed a purely stack based interpretation. It was one tape, full of 16 bit floats, which encoded everything. If the float was a standard normal float then it would be pushed to the top of the stack. However, if the float was not a number (NaN), the payload bits of the float would instead be read as an opcode to run. This allowed for efficient encoding of both data types onto the same type in a way that allowed quick hardware accelerated decoding.

Unfortunately, this system had issues: it was at this point that I realised I had much less cache space to work with than originally thought, and so the size of the tape was much more limited. This was an issue if all the constants were stored on the tape beside the opcodes - for example, to push a 4x4 matrix to the stack the tape would need to use 16 different opcodes, one for each element of the matrix. This was infeasible for such a short tape.

I made the decision to split the constants out from the main tape onto separate tapes, and then using the now unused bits in the opcode to indicate if an input was to pull from the normal stacks or the constant tape. This did solve the issue of inefficient tape use, but on the other hand the new implementation has major issues with cache hits. Due to the separation of constants the spatial locality of the tape had dramatically decreased, meaning that cache misses were becoming much more frequent - especially if an entire tape is laid out before the next.

There is a theoretical solution to both of these problems: encode the tape as a single tape with both opcodes and constants, but also have the information on if the next input is a constant or from the stack. Then, when a constant value is seen to be needed for an opcode, it can pull from the main tape instead. The trick of encoding the opcodes inside of NaNs will likely need to be used, so that if the prune is working backwards on the tape it can easily see what is and isn't an instruction, and so that when the interpreter masks an instruction you can see when the next instruction happens instead of trying to execute a constant. However, due to the lack of bits here, the entire tape will likely need to be upgraded to 32 bit floats. This is not a large problem, as the constant tapes are already this width. It also solves the issue of the constants needing to be read even if the instruction is skipped to make sure the constant tapes stay in sync, as the constants are easily identified and ignored if needed.

7.4 Specification Issues

After splitting the tape out into constants, I encountered a separate problem, but rather than it being a logical issue it was instead due to a misassumption within the GLSL specification. After creating separate tapes for each vector, a test scene was rendering incorrectly. After much trial and error, I decided the only real way to diagnose the issue was to actually perform some form of fuzzing on the interpreter, and so created a compute shader that ran inline with the rest of the code performing fuzz operations. This compute shader was able to identify that the issue came with reading vector 3 elements from the tape itself, which was odd as all other constants seemed to work fine. Researching the issue, I found a very large rabbit hole.

In GLSL 1.4 types are always aligned to a 16 byte boundary, which is an issue if a type is not a multiple of 16 bytes long and you are not writing it as that from the CPU side. However, introduced in GLSL 4.3 was a special qualifier to tell the GPU to ignore this and access them as packed as possible. This allowed the constant tapes to work correctly - except for vector 3s. It turns out that these specific types ignore the GLSL 4.3 specification entirely and default to GLSL 1.4, but even then the definition is vendor specific and hard to predict from the CPU side.[16] As a precaution I merged the vector 3 and vector 4 tapes, and now to read a vector 3 a vector 4 is read and the `w` component is ignored. This is minorly less space efficient, but should be more time efficient - and at least it works.

7.5 Opcode Issues

The opcode set used by the project has various issues.

Firstly, it is massive. This makes the project very versatile, to the point where you could implement most GLSL programs in it, but means that the code required to interpret it is also massive. The standard interpreter is over 2000 lines of GLSL, and the interval interpreter over 3000, not including the Rust interpreter. This leads to massive issues in code upkeep, where all three need to be edited, and if all instructions need a change up to 6000 lines of code need to be modified. It also caused issues in writing the project - a significant amount of time was spent writing and debugging code related to the interpreters, most of which will never be run due to them being contained within obscure opcodes which are not needed.

The massive files mean that they also compile to massive executables. This takes both time and space. In terms of time, the actual compile from a developer perspective is not terrible. However, a first time boot from a user perspective has to wait whilst the driver compiles all the SPIR-V code from several files, resulting in megabytes of code. This takes many minutes, and (at one point during development) up to a quarter of an hour. This leads to a terrible user experience.[12]

The six shaders are loaded using a parallel iterator over multiple cores. It is unclear how much this helps - this was written to attempt to perform the shader compile in parallel across multiple cores but this function is performed at a later stage in the driver, which makes this useless. The later stage is not parallelizable as it takes place in the driver itself, not the user code.

In terms of space, the massive compiled code means that the GPU has to attempt to load it all into cache, something it likely cannot do. On the NVIDIA RTX 3070 this project was developed the shader cannot fit into either L0 or L1 cache, having to fit into larger, shared L2. When it is run it needs to be pulled down to L0, but due to the nature of the execution path it is not very cache friendly and will likely often be removed and rewritten many times. A lower opcode count could reduce this by naturally increasing space locality.

GPUs are also simply not designed to handle such large branches and switches. Inspecting the code generated by the AMD Radeon driver when compiling the SPIR-V using the Radeon Graphics Analyzer (RGA) toolkit, I found that the switch statement had been rewritten as a 300-long series of if-statements, which one after the other checked the opcode against a constant. This is a disaster execution wise, es-

[12] During the writing of this report the game The Last of Us: Part 1 for PC was released, which requires the user to wait whilst all shaders are precompiled on the main menu. The game allows you to skip this, but suffers frequent crashing as a result. Many users (understandably) did not want to wait for this, and so the game has now “Mixed” reviews on the Steam store page due to bad user experience. The use of this project in a product could cause similar issues.

pecially as there are opcodes in the Radeon instruction set which can be used to implement a jump table, something which could dramatically increase speed. As someone without extreme low level knowledge in GPU architecture, it is unclear however if a jump table would cause issues with other parts of the system, for example predictive execution or pipelining, causing it to be somehow slower - but on the scale of 300 sequential checks it is at least feasible that it could be faster. It is likely that this is not handled by the driver simply because someone has never tried it before.

7.6 GLSL Issues

The GPU code was written in GLSL, which presented some issues. As part of the interpreter some functions need to perform vectorwise boolean functions, but these do not exist. They exist in HLSL, a different language, and the underlying SPIR-V supports it, but GLSL itself does not. Instead a mix is used in a similar way to that of a transistor, switching between another value and a constant in order to provide AND and OR boolean operators. The compile stage also is not smart enough to convert these to the underlying SPIR-V operations, and so they are left as mixes.

SPIR-V also supports multiple entrypoints, something that GLSL does not; however not very many high-level graphics programming languages support this anyway. One notable example of a language which does is rust_gpu, which compiles special rust code into SPIR-V, and was considered for use in the project multiple times to help also with code reuse and general debugging. However the language currently has little support for compute shaders and by extension mesh and task shaders, and as such would not be suitable for the project. Multiple entrypoints would allow the things like the interval interpreter code to be shared across the Task and Mesh shaders rather than being duplicated entirely.

7.7 Performance Issues

In terms of actual issues regarding performance, a major issue is overdraw. There is backface culling performed on the cubes, but the cubes overlap each other a lot which is unneeded. A common technique to combat this is called early depth test, where the depth check is performed before the fragment is calculated, which can mean the fragment is thrown out before any complicated calculations need to be performed. This would be useful, but early depth test is disabled due to the depth being changed midway through the fragment shader. This means that even if the GPU goes into the fragment knowing that it is behind something, it still goes through the trouble of calculating it all before it decides it doesn't need it and discard it. This is catastrophic, resulting in many entire shapes being rendered at least double what is needed (for the back of the shape). A solution to this could involve writing a specialised depth buffer which is a normal buffer, and then using atomic operations to check against this and implement the functionality manually. This was considered for the project but ultimately scrapped due to time constraints. This is inspired by the game Teardown.[17]

7.8 Miscellaneous Issues

The Dup2, Dup3, and Dup4 opcodes do not work. This is due to their nature of ignoring how a stack works, and instead accessing values further down than the top value. They do not do anything majorly different than Duplicate however, and so an edge case can be added to make them work within the interpreter itself and the dependency generator. They do not add much help at the moment however, as there is no way to remove items from the stack.

The uniform buffers do not change often, however they are still written every frame. These could be stored more efficiently (especially the light buffer which rarely changes) by keeping track of when the data needs to be updated, and storing it in video memory. Storing to video memory rather than GPU accessible CPU memory is a long process, and needs to be completed for 10 different buffers, but the benefits of having the data in the fast VRAM rather than the slow main RAM outweigh this. An example of this is the buffers for the implicit surfaces models.

The frames per second graph is functional but not good, as it fluctuates often and is hard to read. A better graph could be written using smoothing algorithms, but was unneeded for the project.

The command buffers could be precalculated for each swapchain image for efficiency, but in the case of the project they are recalculated each frame for ease of use. Some data for rendering is sent via a push constant, but could be performed indirectly. Indirection may mildly increase performance, but is harder to write and out of the scope of this project.

A very simple lighting model is used, which is not realistic. Any lighting model can be substituted, as this is just a standard pipeline and it is designed to show that the implicit surface can return anything required for standard shading if needed.

8 Conclusions and Future Work

Overall the project was a learning experience, but did not produce a result that would be useful in an actual product. There are several things that make this project slower than current methods, including the interpreter and the overdraw problem. A future solution could involve inlining the tape better to increase cache coherency, implementing a depth buffer manually, and porting to rust_gpu. Another option is to try ignoring the pruning process entirely and focusing on using the interval arithmetic and mesh shading to generate a hull - this would allow use of pre-coded functions rather than an interpreter, which would dramatically reduce code complexity and runtime overhead at the cost of artist's time. A tool to generate GLSL, or possibly SPIR-V from models could be created but this then needs to be compiled causing loading time issues.

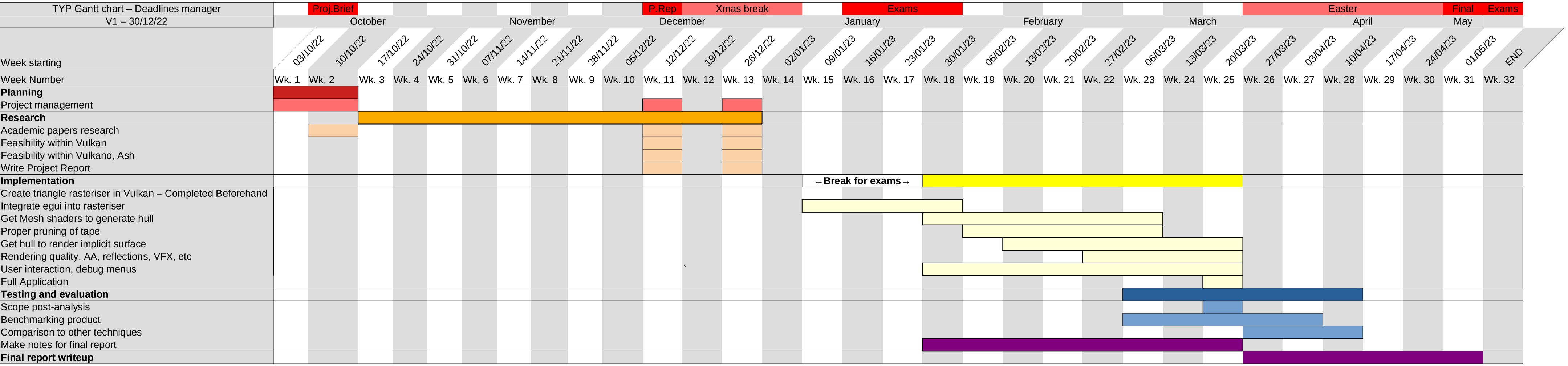
Fixing the Dup2, Dup3, and Dup4 opcodes would allow certain operations to be performed involving the position value, however this is not essential. As well as the duplicate a drop instruction would need to be added, to allow removal of item from the stack. This adds a significant amount of complexity to the pruner: any item that is not a duplicate has to be removed between the drop and the last duplicate in the tree, then the duplicate has to be informed it is working with a drop. Then, if the other branch is either also a drop or to be pruned, both the duplicate and the drop need to be removed. This allows them to act as if they are a drag to front instruction - a dedicated “drag to front” instruction may also be added. All this may be alleviated altogether by simply adding a “push p” instruction, that re-adds p to the stack - as this is the only reason why a duplication instruction is required.[13]

The GPU uses a stack based interpreter. More research could be done into if a register based interpreter is actually more performant, taking into account the need to prune instructions.[18] A mixed approach may be useful, where the Task and Mesh shaders use a stack based interpreter and then when being sent to the Fragment shader it is converted to a register based approach, as past that point no more pruning needs to be performed, and performance is more important.

9 Gantt Chart

On the following page is a Gantt chart of the work I have performed. It is the same chart as used on the progress report, as it was followed reasonably accurately, aside from the features I was unable to implement. It includes many gaps due to other commitments outside of the project, including other course-works.

[13] A duplicate instruction can also of course make code smaller and more efficient, but as p is only available once it is the only thing to necessitate its inclusion.



Other notable dates: 22/11/22 – Game dev – CW 1 due 10/01/23 – Game dev – CW 2 due

Need like a month+ to writeup report properly

References

[1] L. Sorcery, "This SDF Paper Does Not Cite Inigo Quilez," *Twitter*, vol. 69, p. 420, Dec. 2022. [Online]. Available: https://twitter.com/lunasorcery/status/1608843929918803968

[2] I. Quilez, "Distance Functions." [Online]. Available: https://iquilezles.org/articles/distfunctions/

[3] ——, "Smooth Minimum." [Online]. Available: https://iquilezles.org/articles/smin/

[4] ".bsp - Valve Developer Community." [Online]. Available: https://developer.valvesoftware.com/wiki/.bsp

[5] Teadrinker, "English: Visualization of SDF ray marching algorithm," Feb. 2022. [Online]. Available: https://commons.wikimedia.org/wiki/File:Visualization_of_SDF_ray_marching_algorithm.png

[6] C. Kubisch, "Introduction to Turing Mesh Shaders," Sep. 2018. [Online]. Available: https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/

[7] ephtracy, "MagicaCSG." [Online]. Available: https://ephtracy.github.io/index.html?page=magicacsg

[8] "MagicaCSG - Twitter Search / Twitter." [Online]. Available: https://twitter.com/search?q=%23MagicaCSG

[9] M. J. Keeter, "Massively parallel rendering of complex closed-form implicit surfaces," *ACM Transactions on Graphics*, vol. 39, no. 4, Aug. 2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3386569.3392429

[10] B. Keinert, H. Schäfer, J. Korndörfer, U. Ganse, and M. Stamminger, *Enhanced Sphere Tracing.* The Eurographics Association, 2014, accepted: 2014-12-16T07:17:30Z. [Online]. Available: https://diglib.eg.org:443/xmlui/handle/10.2312/stag.20141233.001-008

[11] C. Bálint and M. Kiglics, "A Geometric Method for Accelerated Sphere Tracing of Implicit Surfaces," *Acta Cybernetica*, vol. 25, no. 2, pp. 171–185, Aug. 2021, number: 2 Publisher: University of Szeged. [Online]. Available: https://cyber.bibl.u-szeged.hu/index.php/actcybern/article/view/4203

[12] C. Bálint and G. Valasek, "Accelerating Sphere Tracing," 2018, accepted: 2018-04-14T18:32:46Z Publisher: The Eurographics Association. [Online]. Available: https://diglib.eg.org:443/xmlui/handle/10.2312/egs20181037

[13] "Vulkano." [Online]. Available: https://vulkano.rs/

[14] "Molive-0/vulkano: Safe and rich Rust wrapper around the Vulkan API." [Online]. Available: https://github.com/Molive-0/vulkano

[15] E. Ernerfeldt, "emilk/egui." [Online]. Available: https://github.com/emilk/egui

[16] "Interface Block (GLSL) - OpenGL Wiki." [Online]. Available: https://www.khronos.org/opengl/wiki/Interface_Block_(GLSL)#Memory_layout

[17] J. Ryan, "Teardown Teardown." [Online]. Available: https://juandiegomontoya.github.io/teardown_breakdown.html

[18] Y. Shi, D. Gregg, A. Beatty, and M. A. Ertl, "Virtual Machine Showdown: Stack Versus Registers."

[19] A. Mihut, C. Kubisch, and M. Kraemer, "Advanced API Performance: Mesh Shaders," Oct. 2021. [Online]. Available: https://developer.nvidia.com/blog/advanced-api-performance-mesh-shaders/

[20] M. Keeter, "gpu_opcode.hpp," Dec. 2022, original-date: 2020-04-25T13:03:47Z. [Online]. Available: https://github.com/mkeeter/mpr/blob/eb63defa00239571984e579039166ea575da0dad/inc/gpu_opcode.hpp

[21] "Encoding | Protocol Buffers." [Online]. Available: https://developers.google.com/protocol-buffers/docs/encoding

[22] M. Klein and M. Suijten, "Ash," Dec. 2022, original-date: 2016-08-13T23:13:25Z. [Online]. Available: https://github.com/ash-rs/ash

Appendices

A Original Progress Report

A.1 Abstract

Implicit surfaces are a form of defining shapes using mathematical functions instead of triangle meshes. While they tout many benefits over meshes, they have a major downside in that they are very hard to rasterise.

This paper outlines a method of rasterising SDF based implicit surfaces by using mesh shaders to generate a close hull of cuboids around the surface, with a pruned version of the distance function specific to each cuboid, then using specialised fragment shaders to sphere trace the final short distance.

This method will be shown working along side traditional triangle based rendering to produce a mixed format image, and then it will be benchmarked against various other similar techniques.

A.2 Introduction

Since the 1990s and earlier 3D graphics that are rendered on GPUs (Graphics Processing Unit) have usually been made of small triangles. This is because triangles as a shape have multiple useful properties, notably being always non-convex and planar. These combine to create something which is easy to render with simple algorithms.

However, triangles have downsides also - most obviously that as planar shapes they cannot represent curved surfaces, only approximate them. Common approaches to this problem involve increasing the number of polygons around curved surfaces, but this increases both modelling and rendering complexity.

Implicit surfaces are different way of defining shapes. Instead of providing an exact boundary of the shape, they provide a mathematical function which given a point in 3D space returns if it is inside the shape or not. In this way, implicit surfaces do not exist until observed.

Implicit surfaces are useful for various reasons, including that as a continuous function they can provide infinite resolution (given that the input point in 3D space also has infinite resolution). This means that when defined as an implicit surface a curved surface can truly exist as a curved surface, rather than an approximation.

They can also perform Constructive Solid Geometry (CSG) operations with relative ease, something that triangle based meshes are very cumbersome at. CSG is often used in Computer Aided Design (CAD) operations as it guarantees unambiguous inside-out checking, useful for engineering applications to confirm that shapes can be manufactured and will work as specified. In the context of video games it can be used for collisions, for example to check if the player is touching or inside the floor or walls. The Source Engine by game developer Valve uses CSG for levels for this reason, and can therefore easily confirm a level is sealed at compile time and if a player is out of bounds during runtime.

Unfortunately, implicit surfaces are not without issue either, as due to their implicit nature they are much harder to render efficiently to a 2D screen than triangles. Often implicit surfaces will be converted to triangles before rendering via various methods, such as marching cubes. While this does still allow CSG manipulations of the shape, this brings back the issue of the shape having finite resolution. It also is very hard to do well algorithmically, often producing jagged edges and discontinuities. To this end, ”retopologising” of a model is often done by hand by the artist rather than automatically by the modelling software, and is both time consuming and cannot be performed at runtime.

Other common approaches to rendering implicit surfaces include sphere tracing, a variant of ray marching, where Signed Distance Functions (SDFs) are used. In this method the function used by the surface will return the distance to the closest point on the surface, and the sign of the distance indicates if the point is inside or outside. It can be trivially proven that there exists a sphere around a point in 3D space which does not intersect an implicit surface, of which the radius is the result of the implicit function when given the point. As the shape is not within this sphere, marching the ray forwards by the result each time allows faster reaching of a surface compared to fixed step marching. This however, like any form of ray marching, is very slow to compute.

Recently Nvidia have released graphics cards with dedicated hardware to accelerate raytracing. This hardware will only allow intersections between rays and triangles, and cannot handle rays with arbitrary mathematical functions. They do

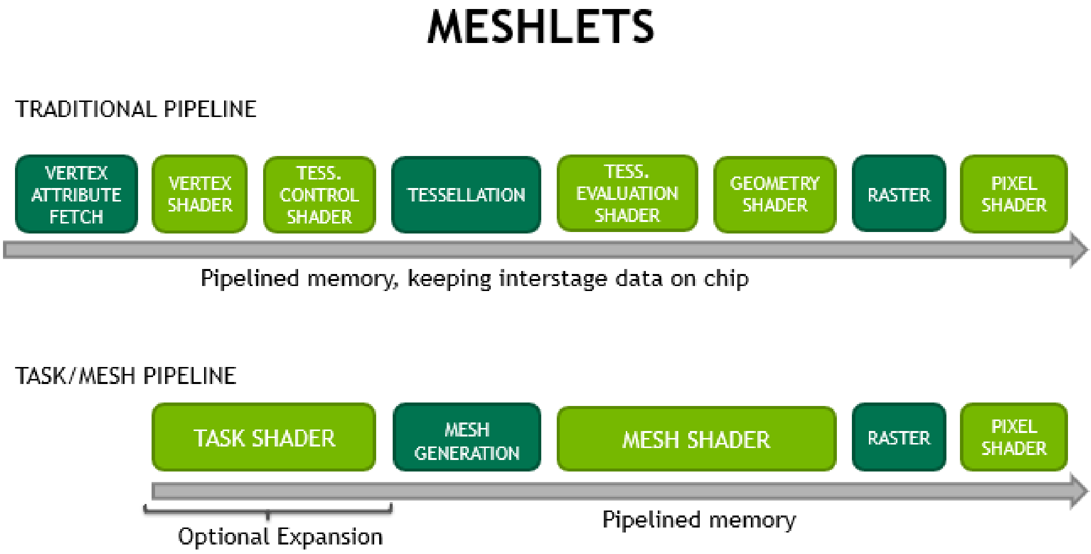


Figure 9: Differences in the traditional versus task/mesh geometry pipeline (Reproduced from figure 4 at [6])

allow hit shaders, which allow you to run traditional programmable shaders when a ray hits a surface, and can mark the surface as transparent. This is meant to help with transparent materials such as leaves, but it can also be used to drop down to software marching implementations to render implicit surfaces.

Along side the raytracing capabilities, these ”RTX” GPUs also have a feature called Mesh Shading where instead of a traditional rendering pipeline a more compute based approach to generating meshes takes place (Fig. 9). This allows much greater procedural mesh generation than something like tessellation or geometry shaders, similar to what is possible with compute based generation but without needing to pass all the results into memory each time they are generated.

While they are often used, actually creating implicit surfaces is not easy. Often in the demoscene (a computer art subculture where people make short real-time rendered ”music videos” called ”demos”), the shapes will be modelled entirely using GPU shader code and viewed in real time in the final engine. While this may work for the programmers who create demos, this is not a good way to create models from the perspective of an artist - they often use tools which allow you to manipulate the object directly rather than indirectly in code. It also brings the issue of syntax and logic errors into the creation of 3D models, which is an added layer of complexity and difficulty that makes uptake hard.

To this end, the computer graphics programmer Ephtracy is creating a program called MagicaCSG[7] which allows modellers to create implicit based models with ease. It uses a familiar UI similar to other 3D modelling software, and allows users to see a path-traced version of the model at all times[14]. MagicaCSG also is currently closed source and non-free, but it provides a common (although not industry accepted) file format for implicit shapes. Most importantly it has a good following of artists who are creating models in this file format.[8]

[14] Due to the closed-source nature of the project it is hard to tell exactly how it is path-traced, but due to observed artifacts it is likely marching cubes is applied.

A.3 Reviewing approaches

Keeter uses a grid of tiles proportional to the screen size to segment the implicit surface on the screen.[9] They also use a special tape shortening algorithm so that the entire surface does not need to be evaluated for each pixel. This method however cannot handle triangles as well as implicit shapes in the same image, and runs as a compute shader.

Keinert et al use a form of relaxed sphere tracing, which oversteps and then corrects itself.[10] This performs marginally better than traditional sphere tracing, but is outperformed by Balint et al.[11]

Balint and Kiglics use quadrics to approximate a surface before rendering, allowing a speedup of up to 100% in rendering static scenes.[11] This method has not been thoroughly tested on dynamic scenes, although they note that simply recomputing the quadrics every frame gives a result which is sometimes faster than Balint et al's previous work, which requires no pre-computation.[12]

A.4 Implementation

In this paper we propose a system based on one by M. Keeter[9], where the model is split into a grid of tiles, and then rendered with interval evaluation. Unlike Keeter, the tiles will be placed and rotated in world space rather than screen space, and the model will be rendered using the mesh graphics pipeline such that it can be rendered alongside traditional triangle meshes.

Rather than increasing the number of tiles per subdivision of the grid, the shape of the tiles will be changed such that the model fits - the cubes for each tile will be stretched into cuboids. A first interval evaluation will be performed on the model to determine the domain that gives the range of [0, -inf). This can then be used as a tight bounding box for the grid. This evaluation will be performed in a compute shader one frame ahead, with one thread per instance of an implicit surface.

The grid will then be evaluated using mesh shaders in a similar way to that of the original design, to generate a voxel shape which is a tighter fit to the implicit surface than a cuboid. First off a task shader is generated for each instance, for each direction in the x and y directions. This will be performed by using workgroup indices, where x indicates the instance to work on, and y and z map to the x and y index within the cuboid. Each task shader will read from the shared mesh memory to find the full tape, along with how many tiles there are in the z direction. It then iterates over each of these z direction tiles and works out each pruned tape and size. In the event of an ambiguous tile, this information is then passed as a payload into several mesh shaders, more than one for each top level tile - any unambiguous tile generates no geometry. When passing this information, it is important that the payload be as small as possible, as the payload size has a dramatic effect on performance.[19].[15]

When the mesh shaders are started they perform a similar task to the task shaders before them, but they act on a much smaller scale and with a pre-pruned tape. These shaders will vastly outnumber the task shaders, but due to the pruning and discarding of unambiguous tiles it will be much fewer than theoretically needed for the given resolution. These shaders will also operate on a line in the z direction, like the task shaders, and again if they find unambiguous tiles no geometry will be generated.

[15] Some citations refer to the "task" shader as the "amplification" shader. This is due to naming differences between DirectX and Vulkan, and they refer to the same stage.

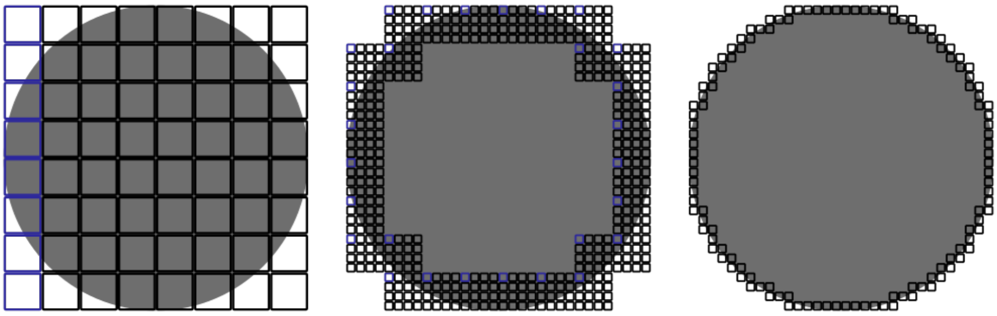


Figure 10: A diagram showing a 2D slice of the hull generation of a sphere. The three stages are Task shading, Mesh shading, and the final product. The blue squares indicate the starting points of each of the shader invocations per stage.

However, if the mesh shaders find an ambiguous tile at this resolution a cuboid is instead output, taking up the full extent of the tile. This cuboid will be rendered with a special fragment shader to produce the original implicit surface. Each bottom level cuboid will store the final pruned tape required to evaluate it as a pointer in shared memory, such that when rendered in the fragment shader it does not need to consider the whole shape.

The tape used will be a byte level opcode based instruction tape, with variable length instructions. It will have opcodes based on that from Keeter,[20] but will also include smin and smax from Quilez.[3] This is so that blending operations, which may be common, are sped up. The tape will attempt to be as compressed in memory as possible, with slight preference to space compared to decoding speed. As part of space saving VarInts will be used.[21]

Then, we render all models in the scene, both triangle and implicit. On rasterisation of the tiles, we perform a sphere trace into the pruned tape stored with the tile we are rendering. If the trace passes through the shape then we simply discard the pixel, otherwise it is rendered as if it is a real shape in the scene description. While tracing the current depth buffer value for the pixel will be used as the far plane. This means that if a triangle based mesh intersects the implicit surface or passes very close in front of it the sphere trace won't accidentally render the implicit shape in front of the triangle based one. Rays will also be started at the surface of the cuboid, which places them very close to the implicit surface compared to the camera for significant speedups. There will also be a very low limit on iterations performed by the trace, as the surface is expected to be close to the ray starting point.

After this some sort of anti-aliasing should be applied to lessen the hard edge generated from discarding fragments. It is not feasible to use MSAA due to discarding, so either DLSS, TAA, SSAA, or FXAA will be used, listed here in order of descending preference.

DLSS is Deep Learning Super Sampling, which uses AI to upscale images (and provide anti-aliasing) based on previous frames and motion inputs - this would require both generating the motion inputs and then hooking up DLSS to the project as an external library, which isn't easy. It theoretically produces the best result.

TAA is Temporal Anti Aliasing, which uses standard algorithms based on the previous frame and motion inputs to reduce aliasing on the current frame. By shaking the screen slightly sub pixel data can be gathered. This would also require generating motion vectors, but would not require linking a proprietary library.

SSAA is Super Sampling Anti Aliasing, which is the simple in its design, and the most accurate. The image is simply rendered at a higher resolution, and then scaled down to fit the screen. It is by far the best result in terms of accuracy, but dramatically increases the computational cost.

FXAA is the simplest and cheapest option: it is a single post process pass over the image where edges are identified and blurred. It looks the worst out of the above options.

MSAA uses extra samples to shade overlaps between shapes more than once. This works well when all triangles are fully opaque, but the triangles as part of the implicit surface will be semi-opaque and therefore this may not work. A technique called alpha to coverage may work here, where alpha values are used to fake MSAA samples. However, we do not know the correct alpha values for the shape.

The scene will be rendered photorealistically using physically based shaders. Reflections will be supported based on screen space techniques, but mainly materials will be diffuse and opaque.

The program will be able to load scene files in as Waveform OBJ files for triangles, and MagicaCSG[7] MCSG files for implicit surfaces - these formats have been chosen as they are widely used by artists and have many example models. Placement in the scene can be performed via the GUI, which will be written in egui[15], a Rust based immediate-mode GUI. egui was picked because of its ease of use, as a complicated GUI system is not needed for a small app. It was also picked over imGUI as it is written in Rust, and therefore no wrapper is needed. The user will be able to fly around the scene using the mouse and keyboard, and change a few parameters such as light position in the GUI. Rendering will be performed as quickly as possible, with real-time limits.

The program will also obey OBJ and MCSG materials, such that the objects will have the right colours and surface textures. It is unclear if the renderer will support image textures at this point. Actual functions for the default MCSG shapes will be sourced from Inigo Quilez's online repository of examples.[2]

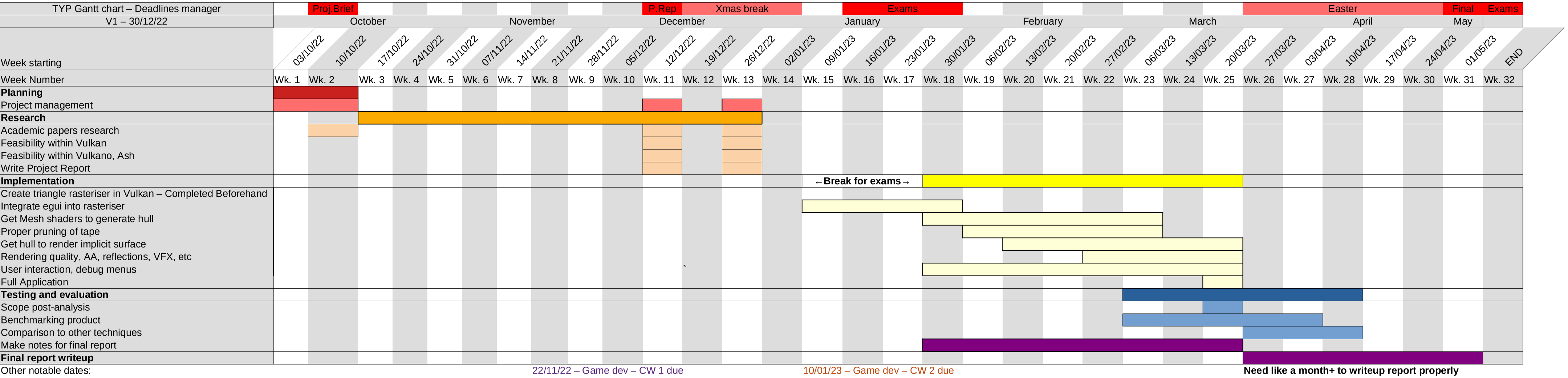
The program will be written in Rust and Vulkan, as these are industry leading and modern platforms. They also allow good portability across operating systems, which DirectX does not. OpenGl will not be used because it is a old API and does not support the required features. Linux will be used as a target OS, but all techniques should be applicable on Windows as well. Mac OS will not be considered as I do not own an Apple computer.

As an interface to Vulkan, the Rust crate Vulkano will be used. This is because Vulkano allows easy abstractions over the Vulkan API, which saves a lot of development time and reduces errors.[13] Some parts of Vulkan (namely the mesh shader extensions) are not supported in Vulkano. In order to use those the lower level Ash crate will be used, along with the built-in Vulkano to Ash inter-op.[22]

The program will be tested using an Nvidia RTX 3070 graphics card on the latest Linux driver. The code will be written in a vendor agnostic way, such that a Linux computer with a 7000 series Radeon card should also be able to run it, but this will not be verified as I do not have access to any of those cards. It should also be reasonably simple to port the code to Windows (due to the nature of Vulkan only the windowing extensions should need to be changed), but this also will not be tested.

A.5 Gantt Chart

On the following page is a Gantt chart of the work I have performed so far and the work I intend to do. The work so far has included many gaps due to other commitments outside of the project, including other course-works.


TYP Gantt chart – Deadlines manager
V1 – 30/12/22
Proj.Brief
P.Rep
Xmas break
Exams
Easter
Final
Exams
October
November
December
January
February
March
April
May
Week starting
03/10/22
10/10/22
17/10/22
24/10/22
31/10/22
07/11/22
14/11/22
21/11/22
28/11/22
05/12/22
12/12/22
19/12/22
26/12/22
02/01/23
09/01/23
16/01/23
23/01/23
30/01/23
06/02/23
13/02/23
20/02/23
27/02/23
06/03/23
13/03/23
20/03/23
27/03/23
03/04/23
10/04/23
17/04/23
24/04/23
01/05/23
END
Week Number
Wk. 1
Wk. 2
Wk. 3
Wk. 4
Wk. 5
Wk. 6
Wk. 7
Wk. 8
Wk. 9
Wk. 10
Wk. 11
Wk. 12
Wk. 13
Wk. 14
Wk. 15
Wk. 16
Wk. 17
Wk. 18
Wk. 19
Wk. 20
Wk. 21
Wk. 22
Wk. 23
Wk. 24
Wk. 25
Wk. 26
Wk. 27
Wk. 28
Wk. 29
Wk. 30
Wk. 31
Wk. 32
Planning
Project management
Research
Academic papers research
Feasibility within Vulkan
Feasibility within Vulkano, Ash
Write Project Report
Implementation
←Break for exams→
Create triangle rasteriser in Vulkan – Completed Beforehand
Integrate egui into rasteriser
Get Mesh shaders to generate hull
Proper pruning of tape
Get hull to render implicit surface
Rendering quality, AA, reflections, VFX, etc
User interaction, debug menus
Full Application
Testing and evaluation
Scope post-analysis
Benchmarking product
Comparison to other techniques
Make notes for final report
Final report writeup
Other notable dates:
22/11/22 – Game dev – CW 1 due
10/01/23 – Game dev – CW 2 due
Need like a month+ to writeup report properly

B Data Archive Contents

Filename	Filetype	Description
egui_winit_vulkano/	Directory	egui fork
vulkano/	Directory	vulkano fork
build/erroccfisumreg.linux	Linux Executable	Prebuilt Linux executable
build/erroccfisumreg.exe	Windows Executable	Prebuilt Windows executable
third_year_project/build.rs	Rust source	Code that runs at compile time, generates Rust opcode set
third_year_project/Cargo.lock	Cargo config	Configuration for Rust's Cargo
third_year_project/Cargo.toml	Cargo config	Configuration for Rust's Cargo
third_year_project/src/bunny.obj	Wavefront OBJ	Stanford Bunny triangle model
third_year_project/src/frag.glsl	GLSL source	Combined Fragment shader
third_year_project/src/fuzz.comp.glsl	GLSL source	Fuzzing Compute shader
third_year_project/src/gui.rs	Rust source	GUI related code
third_year_project/src/implicit.mesh.glsl	GLSL source	Implicit pipline Mesh shader
third_year_project/src/implicit.task.glsl	GLSL source	Implicit pipline Task shader
third_year_project/src/include.glsl	GLSL source	Various useful GLSL snippets
third_year_project/src/instructionset.glsl	GLSL source	Listing of opcodes for the interpreters
third_year_project/src/instructionset.md	Markdown	Outdated instruction set planning
third_year_project/src/interpreter.glsl	GLSL source	Standard GLSL implicit interpreter
third_year_project/src/interpreter.rs	Rust source	Standard Rust implicit interpreter
third_year_project/src/intervals.glsl	GLSL source	Interval GLSL implicit interpreter
third_year_project/src/main.rs	Rust source	Vulkano related code, application entry point
third_year_project/src/mcsg_deserialise.rs	Rust source	Functions relating to reading .mcsg files
third_year_project/src/objects.rs	Rust source	Definitions and functions relating to meshes, CSGs, lights, etc.
third_year_project/src/primitive.mcsg	MagicaCSG file	Test mcsg file
third_year_project/src/frag.glsl	GLSL source	Triangle pipeline Vertex shader

C Opcode Listing and Explanation

- F: Float
- V2: Vector 2
- V3: Vector 3
- V4: Vector 4
- M2: Matrix 2x2
- M3: Matrix 3x3
- M4: Matrix 4x4
- M: Material

Opcode	Inputs	Outputs	Description
OPNop	None	None	Do nothing
OPStop	F	None	Stop execution, return a Float as output
OPAddFloatFloat	F F	F	Add a Float to a Float
OPAddVec2Vec2	V2 V2	V2	Add a Vec2 to a Vec2
OPAddVec2Float	V2 F	V2	Add a Float to all elements of a Vec2
OPAddVec3Vec3	V3 V3	V3	Add a Vec3 to a Vec3
OPAddVec3Float	V3 F	V3	Add a Float to all elements of a Vec3
OPAddVec4Vec4	V4 V4	V4	Add a Vec4 to a Vec4
OPAddVec4Float	V4 F	V4	Add a Float to all elements of a Vec4
OPSubFloatFloat	F F	F	Subtract a Float from a Float
OPSubVec2Vec2	V2 V2	V2	Subtract a Vec2 from a Vec2
OPSubVec2Float	V2 F	V2	Subtract a Float from all elements of a Vec2
OPSubVec3Vec3	V3 V3	V3	Subtract a Vec3 from a Vec3
OPSubVec3Float	V3 F	V3	Subtract a Float from all elements of a Vec3
OPSubVec4Vec4	V4 V4	V4	Subtract a Vec4 from a Vec4
OPSubVec4Float	V4 F	V4	Subtract a Float from all elements of a Vec4
OPMulFloatFloat	F F	F	Multiply a Float with a Float
OPMulVec2Vec2	V2 V2	V2	Multiply a Vec2 with a Vec2
OPMulVec2Float	V2 F	V2	Multiply all elements of a Vec2 with a Float
OPMulVec3Vec3	V3 V3	V3	Multiply a Vec3 with a Vec3
OPMulVec3Float	V3 F	V3	Multiply all elements of a Vec3 with a Float
OPMulVec4Vec4	V4 V4	V4	Multiply a Vec4 with a Vec4
OPMulVec4Float	V4 F	V4	Multiply all elements of a Vec4 with a Float
OPDivFloatFloat	F F	F	Divide a Float by a Float
OPDivVec2Vec2	V2 V2	V2	Divide a Vec2 by a Vec2
OPDivVec2Float	V2 F	V2	Divide all elements of a Vec2 by a Float
OPDivVec3Vec3	V3 V3	V3	Divide a Vec3 by a Vec3
OPDivVec3Float	V3 F	V3	Divide all elements of a Vec3 by a Float
OPDivVec4Vec4	V4 V4	V4	Divide a Vec4 by a Vec4
OPDivVec4Float	V4 F	V4	Divide all elements of a Vec4 by a Float
OPModFloatFloat	F F	F	Find a Float modulo a Float
OPModVec2Vec2	V2 V2	V2	Find a Vec2 modulo a Vec2
OPModVec2Float	V2 F	V2	Find all elements of a Vec2 modulo a Float
OPModVec3Vec3	V3 V3	V3	Find a Vec3 modulo a Vec3
OPModVec3Float	V3 F	V3	Find all elements of a Vec3 modulo a Float
OPModVec4Vec4	V4 V4	V4	Find a Vec4 modulo a Vec4
OPModVec4Float	V4 F	V4	Find all elements of a Vec4 modulo a Float
OPPowFloatFloat	F F	F	Raise a Float to the power of a Float
OPPowVec2Vec2	V2 V2	V2	Raise a Vec2 to the power of a Vec2 (component-wise)
OPPowVec3Vec3	V3 V3	V3	Raise a Vec3 to the power of a Vec3 (component-wise)
OPPowVec4Vec4	V4 V4	V4	Raise a Vec4 to the power of a Vec4 (component-wise)
OPCrossVec3	V3 V3	V3	Find the cross product of two Vec3s
OPDotVec2	V2 V2	F	Find the dot product of two Vec2s
OPDotVec3	V3 V3	F	Find the dot product of two Vec3s
OPDotVec4	V4 V4	F	Find the dot product of two Vec4s
OPLengthVec2	V2	F	Find the length (magnitude) of a Vec2
OPLengthVec3	V3	F	Find the length (magnitude) of a Vec3
OPLengthVec4	V4	F	Find the length (magnitude) of a Vec4
OPDistanceVec2	V2 V2	F	Find distance between two Vec2s (Subtract, then magnitude)
OPDistanceVec3	V3 V3	F	Find distance between two Vec3s (Subtract, then magnitude)
OPDistanceVec4	V4 V4	F	Find distance between two Vec4s (Subtract, then magnitude)
OPNormalizeVec2	V2	V2	Normalize a Vec2
OPNormalizeVec3	V2	V2	Normalize a Vec3
OPNormalizeVec4	V4	V4	Normalize a Vec4
OPAbsFloat	F	F	Find the absolute value of a Float
OPSignFloat	F	F	Find the sign of a Float (returns -1, 0, or 1)
OPFloorFloat	F	F	Find the floor of a Float
OPCeilFloat	F	F	Find the ceiling of a Float
OPFractFloat	F	F	Find the fractional component of a Float

Opcode	Inputs	Outputs	Description
OPSqrtFloat	F	F	Find the square root of a Float
OPInverseSqrtFloat	F	F	Find the inverse of the square root of a Float
OPExpFloat	F	F	Find e to the power of a Float
OPExp2Float	F	F	Find 2 to the power of a Float
OPLogFloat	F	F	Find the natural logarithm of a Float
OPLog2Float	F	F	Find the base 2 logarithm of a Float
OPSinFloat	F	F	Find the sine of a Float
OPCosFloat	F	F	Find the cosine of a Float
OPTanFloat	F	F	Find the tangent of a Float
OPAsinFloat	F	F	Find the arcsine of a Float
OPAcosFloat	F	F	Find the arccosine of a Float
OPAtanFloat	F	F	Find the arctangent of a Float
OPMinFloat	F F	F	Find the minimum of two Floats
OPMaxFloat	F F	F	Find the maxmimum of two Floats
OPSmoothMinFloat	F F F	F	Find the minimum of two Floats, interpolating between the two when they are close based on a Float
OPSmoothMaxFloat	F F F	F	Find the maximum of two Floats, interpolating between the two when they are close based on a Float
OPMinMaterialFloat	F F	F	Find the minimum of two Floats, and perform material calculations
OPMaxMaterialFloat	F F	F	Find the maxmimum of two Floats, and perform material calculations
OPSmoothMinMaterialFloat	F F F	F	Find the minimum of two Floats, interpolating between the two when they are close based on a Float, and perform material calculations
OPSmoothMaxMaterialFloat	F F F	F	Find the maximum of two Floats, interpolating between the two when they are close based on a Float, and perform material calculations
OPDupFloat	F	F F	Push an extra of the Float at the top of the stack to the stack
OPDup2Float	F	F F	Push an extra of the Float one below the top of the stack to the stack
OPDup3Float	F	F F	Push an extra of the Float two below the top of the stack to the stack
OPDup4Float	F	F F	Push an extra of the Float three below the top of the stack to the stack
OPAbsVec2	V2	V2	Find the absolute value of the elements of a Vec2
OPSignVec2	V2	V2	Find the sign of the elements of a Vec2 (returns -1, 0, or 1)
OPFloorVec2	V2	V2	Find the floor of the elements of a Vec2
OPCeilVec2	V2	V2	Find the ceiling of the elements of a Vec2
OPFractVec2	V2	V2	Find the fractional component of the elements of a Vec2
OPSqrtVec2	V2	V2	Find the square root of the elements of a Vec2
OPInverseSqrtVec2	V2	V2	Find the inverse of the square root of the elements of a Vec2
OPExpVec2	V2	V2	Find e to the power of the elements of a Vec2
OPExp2Vec2	V2	V2	Find 2 to the power of the elements of a Vec2
OPLogVec2	V2	V2	Find the natural logarithm of the elements of a Vec2
OPLog2Vec2	V2	V2	Find the base 2 logarithm of the elements of a Vec2
OPSinVec2	V2	V2	Find the sine of the elements of a Vec2
OPCosVec2	V2	V2	Find the cosine of the elements of a Vec2
OPTanVec2	V2	V2	Find the tangent of the elements of a Vec2
OPAsinVec2	V2	V2	Find the arcsine of the elements of a Vec2
OPAcosVec2	V2	V2	Find the arccosine of the elements of a Vec2
OPAtanVec2	V2	V2	Find the arctangent of the elements of a Vec2
OPMinVec2	V2 V2	V2	Find the minimum of the elements of two Vec2s
OPMaxVec2	V2 V2	V2	Find the maxmimum of the elements of two Vec2s
OPDupVec2	V2	V2 V2	Push an extra of the Vec2 at the top of the stack to the stack
OPDup2Vec2	V2	V2 V2	Push an extra of the Vec2 one below the top of the stack to the stack
OPDup3Vec2	V2	V2 V2	Push an extra of the Vec2 two below the top of the stack to the stack
OPDup4Vec2	V2	V2 V2	Push an extra of the Vec2 three below the top of the stack to the stack
OPAbsVec3	V3	V3	Find the absolute value of the elements of a Vec3
OPSignVec3	V3	V3	Find the sign of the elements of a Vec3 (returns -1, 0, or 1)
OPFloorVec3	V3	V3	Find the floor of the elements of a Vec3
OPCeilVec3	V3	V3	Find the ceiling of the elements of a Vec3
OPFractVec3	V3	V3	Find the fractional component of the elements of a Vec3
OPSqrtVec3	V3	V3	Find the square root of the elements of a Vec3
OPInverseSqrtVec3	V3	V3	Find the inverse of the square root of the elements of a Vec3
OPExpVec3	V3	V3	Find e to the power of the elements of a Vec3
OPExp2Vec3	V3	V3	Find 2 to the power of the elements of a Vec3
OPLogVec3	V3	V3	Find the natural logarithm of the elements of a Vec3
OPLog2Vec3	V3	V3	Find the base 2 logarithm of the elements of a Vec3
OPSinVec3	V3	V3	Find the sine of the elements of a Vec3

Opcode	Inputs	Outputs	Description
OPCosVec3	V3	V3	Find the cosine of the elements of a Vec3
OPTanVec3	V3	V3	Find the tangent of the elements of a Vec3
OPAsinVec3	V3	V3	Find the arcsine of the elements of a Vec3
OPAcosVec3	V3	V3	Find the arccosine of the elements of a Vec3
OPAtanVec3	V3	V3	Find the arctangent of the elements of a Vec3
OPMinVec3	V3 V3	V3	Find the minimum of the elements of two Vec3s
OPMaxVec3	V3 V3	V3	Find the maxmimum of the elements of two Vec3s
OPDupVec3	V3	V3 V3	Push an extra of the Vec3 at the top of the stack to the stack
OPDup2Vec3	V3	V3 V3	Push an extra of the Vec3 one below the top of the stack to the stack
OPDup3Vec3	V3	V3 V3	Push an extra of the Vec3 two below the top of the stack to the stack
OPDup4Vec3	V3	V3 V3	Push an extra of the Vec3 three below the top of the stack to the stack
OPAbsVec4	V4	V4	Find the absolute value of the elements of a Vec4
OPSignVec4	V4	V4	Find the sign of the elements of a Vec4 (returns -1, 0, or 1)
OPFloorVec4	V4	V4	Find the floor of the elements of a Vec4
OPCeilVec4	V4	V4	Find the ceiling of the elements of a Vec4
OPFractVec4	V4	V4	Find the fractional component of the elements of a Vec4
OPSqrtVec4	V4	V4	Find the square root of the elements of a Vec4
OPInverseSqrtVec4	V4	V4	Find the inverse of the square root of the elements of a Vec4
OPExpVec4	V4	V4	Find e to the power of the elements of a Vec4
OPExp2Vec4	V4	V4	Find 2 to the power of the elements of a Vec4
OPLogVec4	V4	V4	Find the natural logarithm of the elements of a Vec4
OPLog2Vec4	V4	V4	Find the base 2 logarithm of the elements of a Vec4
OPSinVec4	V4	V4	Find the sine of the elements of a Vec4
OPCosVec4	V4	V4	Find the cosine of the elements of a Vec4
OPTanVec4	V4	V4	Find the tangent of the elements of a Vec4
OPAsinVec4	V4	V4	Find the arcsine of the elements of a Vec4
OPAcosVec4	V4	V4	Find the arccosine of the elements of a Vec4
OPAtanVec4	V4	V4	Find the arctangent of the elements of a Vec4
OPMinVec4	V4 V4	V4	Find the minimum of the elements of two Vec4s
OPMaxVec4	V4 V4	V4	Find the maxmimum of the elements of two Vec4s
OPDupVec4	V4	V4 V4	Push an extra of the Vec4 at the top of the stack to the stack
OPDup2Vec4	V4	V4 V4	Push an extra of the Vec4 one below the top of the stack to the stack
OPDup3Vec4	V4	V4 V4	Push an extra of the Vec4 two below the top of the stack to the stack
OPDup4Vec4	V4	V4 V4	Push an extra of the Vec4 three below the top of the stack to the stack
OPPromoteFloatFloatVec2	F F	V2	Convert two Floats into a Vec2
OPPromoteFloatFloatFloatVec3	F F F	V3	Convert three Floats into a Vec3
OPPromoteFloatFloatFloatFloatVec4	F F F F	V4	Convert four Floats into a Vec4
OPPromoteVec2FloatVec3	V2 F	V3	Convert a Vec2, then a Float into a Vec3
OPPromoteVec2FloatFloatVec4	V2 F F	V4	Convert a Vec2, then two Floats into a Vec4
OPPromoteVec2Vec2Vec4	V2 V2	V4	Convert two Vec2s into a Vec4
OPPromoteVec3FloatVec4	V3 F	V4	Convert a Vec3, then a Float into a Vec4
OPAcoshFloat	F	F	Find the hyperbolic arccosine of a Float
OPAcoshVec2	V2	V2	Find the hyperbolic arccosine of the elements of a Vec2
OPAcoshVec3	V3	V3	Find the hyperbolic arccosine of the elements of a Vec3
OPAcoshVec4	V4	V4	Find the hyperbolic arccosine of the elements of a Vec4
OPAsinhFloat	F	F	Find the hyperbolic arcsine of a Float
OPAsinhVec2	V2	V2	Find the hyperbolic arcsine of the elements of a Vec2
OPAsinhVec3	V3	V3	Find the hyperbolic arcsine of the elements of a Vec3
OPAsinhVec4	V4	V4	Find the hyperbolic arcsine of the elements of a Vec4
OPAtanhFloat	F	F	Find the hyperbolic arctangent of a Float
OPAtanhVec2	V2	V2	Find the hyperbolic arctangent of the elements of a Vec2
OPAtanhVec3	V3	V3	Find the hyperbolic arctangent of the elements of a Vec3
OPAtanhVec4	V4	V4	Find the hyperbolic arctangent of the elements of a Vec4
OPCoshFloat	F	F	Find the hyperbolic cosine of a Float
OPCoshVec2	V2	V2	Find the hyperbolic cosine of the elements of a Vec2
OPCoshVec3	V3	V3	Find the hyperbolic cosine of the elements of a Vec3
OPCoshVec4	V4	V4	Find the hyperbolic cosine of the elements of a Vec4
OPSinhFloat	F	F	Find the hyperbolic sine of a Float
OPSinhVec2	V2	V2	Find the hyperbolic sine of the elements of a Vec2

Opcode	Inputs	Outputs	Description
OPSinhVec3	V3	V3	Find the hyperbolic sine of the elements of a Vec3
OPSinhVec4	V4	V4	Find the hyperbolic sine of the elements of a Vec4
OPTanhFloat	F	F	Find the hyperbolic tangent of a Float
OPTanhVec2	V2	V2	Find the hyperbolic tangent of the elements of a Vec2
OPTanhVec3	V3	V3	Find the hyperbolic tangent of the elements of a Vec3
OPTanhVec4	V4	V4	Find the hyperbolic tangent of the elements of a Vec4
OPRoundFloat	F	F	Round a Float to the nearest integer
OPRoundVec2	V2	V2	Round the elements of a Vec2 to the nearest integer
OPRoundVec3	V3	V3	Round the elements of a Vec3 to the nearest integer
OPRoundVec4	V4	V4	Round the elements of a Vec4 to the nearest integer
OPTruncFloat	F	F	Remove the fractional component of a Float
OPTruncVec2	V2	V2	Remove the fractional component of elements of a Vec2
OPTruncVec3	V3	V3	Remove the fractional component of elements of a Vec3
OPTruncVec4	V4	V4	Remove the fractional component of elements of a Vec4
OPFMAFloat	F F F	F	Multiply two Floats, then add a third
OPFMAVec2	V2 V2 V2	V2	Multiply two Vec2a, then add a third
OPFMAVec3	V3 V3 V3	V3	Multiply two Vec3s, then add a third
OPFMAVec4	V4 V4 V4	V4	Multiply two Vec4s, then add a third
OPClampFloatFloat	F F F	F	Clamp a Float between a minimum and maximum Float
OPClampVec2Vec2	V2 V2 V2	V2	Clamp the elements of a Vec2 between a minimum and maximum Vec2, component-wise
OPClampVec2Float	V2 F F	V2	Clamp the elements of a Vec2 between a minimum and maximum Float
OPClampVec3Vec3	V3 V3 V3	V3	Clamp the elements of a Vec3 between a minimum and maximum Vec3, component-wise
OPClampVec3Float	V3 F F	V3	Clamp the elements of a Vec3 between a minimum and maximum Float
OPClampVec4Vec4	V4 V4 V4	V4	Clamp the elements of a Vec4 between a minimum and maximum Vec4, component-wise
OPClampVec4Float	V4 F F	V4	Clamp the elements of a Vec4 between a minimum and maximum Float
OPMixFloatFloat	F F F	F	Interpolate between two Floats, as defined by a Float
OPMixVec2Vec2	V2 V2 V2	V2	Interpolate between two Vec2s, as defined by elements of a Vec2
OPMixVec2Float	V2 V2 F	V2	Interpolate between two Vec2s, as defined by a Float
OPMixVec3Vec3	V3 V3 V3	V3	Interpolate between two Vec3s, as defined by elements of a Vec3
OPMixVec3Float	V3 V3 F	V3	Interpolate between two Vec3s, as defined by a Float
OPMixVec4Vec4	V4 V4 V4	V4	Interpolate between two Vec4s, as defined by elements of a Vec4
OPMixVec4Float	V4 V4 F	V4	Interpolate between two Vec4s, as defined by a Float
OPSquareFloat	F	F	Find the square of a Float
OPCubeFloat	F	F	Find the cube of a Float
OPSquareVec2	V2	V2	Find the square of the elements of a Vec2
OPCubeVec2	V2	V2	Find the cube of the elements of a Vec2
OPSquareVec3	V3	V3	Find the square of the elements of a Vec3
OPCubeVec3	V3	V3	Find the cube of the elements of a Vec3
OPSquareVec4	V4	V4	Find the square of the elements of a Vec4
OPCubeVec4	V4	V4	Find the cube of the elements of a Vec4
OPSDFSphere	F V3	F	Calculate the distance to a sphere of Float radius at the origin to a Vec3
OPSDFBox	V3 V3	F	Calculate the distance to an axis aligned box of Vec3 dimensions at the origin to a Vec3
OPSDFTorus	V2 V3	F	Calculate the distance to an axis aligned torus of Vec2 radii at the origin to a Vec3
OPInvalid	None	None	Placeholder invalid instruction, triggers an exception/crash