Parameterised 3D Modelling with Distance Fields

Teemu Lindborg, Philip Gifford

Bournemouth University, UK

Software Development of Animation Games and Effects

January 23, 2017

Abstract

When approaching 3D modelling, it requires a large amount of expertise to develop complex scenes in quick succession. Powerful modelling tools are required for the user to carry out these demands and artists often look for different ways to achieve quicker and more accurate results. In this paper, we present a visual, node-based approach to 3D modelling using signed distance fields. The node-based approach is similar to those employed by Houdini and Grasshopper and allows for easy parameterisation of the scene. By using signed distance functions to represent the scene, complex objects can be modelled using a combination of set-theoretic operations and simple deformations. Using an already functional and tested scene builder method it grants us a secure building block to base our tool around. This allows the artist to have a more streamlined and user-friendly application which is essential when constructing 3D scenes. Furthermore, the user can provide multiple instructions without having to understand the mathematical operations needed to generate the scene, thus making the application accessible to a wide variety of artists. In our implementation, we create a bridge between the visual front end and the technical back end to represent the scene.

1 Introduction

Signed distance fields (referred to as SDF from this point onwards) is a subject that has gained interest through the development of computer hardware. Especially the development of graphics units makes rendering more complex scenes in real-time possible. Using SDFs for modelling allows the users to effortlessly produce complex objects that would otherwise require an intermediate knowledge of modelling to develop.

SDF's are mathematical operations that are used to define implicit shapes such as spheres, triangular prisms and toruses too name a few. Here it removes the inaccuracy provided by explicit modelling which is often recognised through its use of polygons. With explicit modelling you have the addition of user error such as incorrect tessellation across the mesh. By having an application where objects are defined implicitly it adds an element of security in terms of model accuracy which other explicit applications struggle to provide.

In addition to modelling, the user can alter the primitives material to better outline its design. Through the use of the blending operation, mixing of materials between objects can be produced.

Developing a 3D interactive environment with SDF is one that has not been implemented using this format of parameterisation. One of the reasons is due to real-time rendering issues as the application is required to provide constant video feedback to the user.

Ray marching is a rendering technique that can be used to visualise signed distant primitives in real-time. However, limitations and constraints do arise in relation to our application. This technique is based on an estimation of the distance between the ray and the surface closest to it. The ray continues to extend before it collides with a surface following a series of uniform steps. To counter the limitations of ray marching (explained more in section 3.3), sphere tracing can be used. Sphere tracing is an adaptation of the ray marching algorithm that uses the distance estimations as its marching steps. In this paper we will outline the properties of these methods and our justification to use sphere tracing.

In this paper, we develop a 3D application that uses SDF functions such to represent a 3D model. We then combine it with a node-based editor to create a parameterised 3D modelling environment which functions in real time. Thus, the user can produce models using the SDF operations as well as other utility operations.

2 Related Work

The method to create 3D models using ray marching has been in development since 1972. Ray marching was first implemented by Synthavision using a constructive solid geometry system (CSG) which used the method of boolean operations to produce high-quality images. A good example for CSG can be found in [8] where a blob tree is used to generate geometry.

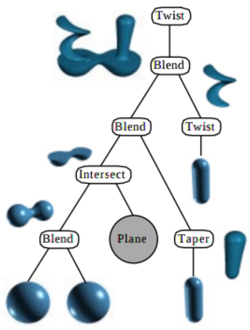

Figure 1: *Represents the outline of the CSG format from Brian Wyvill and Andrew Guy blob tree [8]*

The use of CSG has been involved in applications such as the Quake engine to develop geometry where mathematical accuracy is important. Other areas such as engineering software have been known to use this method where accuracy is at the highest priority.

Other related work includes Tim Reiner's Interactive Modelling of Implicit Surfaces with Signed Distance Functions [1] in which virtual scenes are defined using signed distance functions. Here they produce an interactive application for the artist that removes the requirement of mathematical knowledge for the artist to use. With the use of sphere tracing to achieve real-time results it demonstrates how compositions of signed distance functions can be constructed and used to represent complex implicit objects.

Enhanced Sphere tracing is a rendering technique designed for signed distance bounds [5]. It introduces an over-relaxation method to accelerate the original sphere tracing algorithm by Hart [6]. This method was discussed as a possible ray marching algorithm for the rendering process. Despite taking the standard sphere tracing approach it possessed interesting elements for our application.

Íñigo Quílez has produced a library [13] with many of the signed distance primitives and operations which gave us a strong framework build upon. This also allowed us to start bridging the front end and back end together early on.

Distance rendering has began to be implemented in game engines software such Unity. We found a video during our research [17] where someone had created there very own node base SDF modeller in the Unity game engine. Here they incorporated many of the SDF operations such as Union, intersection etc. This provided us with great reference for our own stand alone application.

Further related work included paper [2] Space-Time Transfinite Interpolation of Volumetric Material Properties which deals with interpolating materials between objects by using a method known as transfinite interpolation. Furthermore it proceeds to compute these material in real-time. More of this paper is discussed in section 3.2 as well as the paper.

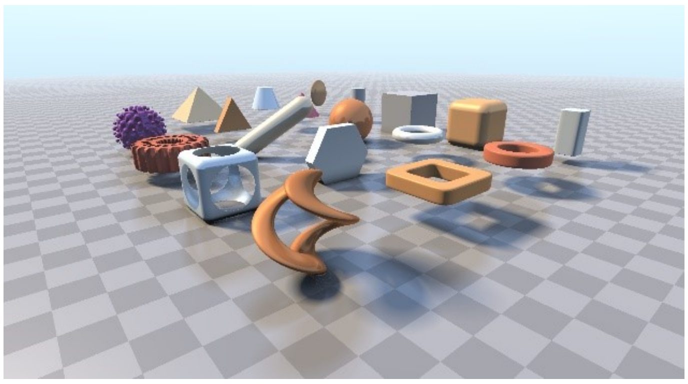

Figure 2: *This is a demonstration of IQ signed distance primitives with sphere tracing. [14]*

3 Background

3.1 Properties of Signed Distance Fields

Before discussing the properties of SDF surfaces, the fundamentals of implicit surfaces should be outlined to benefit the understanding of the signed distant field functions later. As described in Jaemin Shin Mesh generation with a signed distance function [10] the implicit function is used to determine whether we are inside, outside or on the interface of the domain. In general, for x ∈ Rn which represents the spatial dimension we define the interface as

$$\partial\Omega = \vec{x}|\phi(\vec{x}) = 0$$

The interface is described as the point on the surface of the implicit surface while this represents the inside region of the domain

$$\Omega- = \vec{x}|\phi(\vec{x}) < 0$$

and this equation is described as the outside of the domain.

$$\Omega+ = \vec{x}|\phi(\vec{x}) > 0$$

Information on implicit surfaces can be found in [10] and [11] however we find the information provided above demonstrates the general principles of implicit surfaces since signed distance functions follow a similar approach.

In our system, we use signed distance functions (sometimes this referred to as fields) which can be used to create complex objects as previously described in this paper. Signed distance fields, as mentioned share similar fundamentals of implicit functions in terms of being defined to be positive on the exterior, negative on the interior and zero on the interface [9] and [7]. The signed distance function can be determined as $\phi(\vec{x}) = d(\vec{x})$ for all $\vec{x}$ where Ω- and Ω+ determine the space in and out of the implicit surface (Ω+ determines if it is out and Ω- determines if it is in) and ∂Ω as the interface between the surface. Overall resulting in

$$\phi(\vec{x}) = d - (\vec{x}) \text{ if } \vec{x} \in \Omega+$$

$$\phi(\vec{x}) = d - (\vec{x}) = 0 \text{ if } \vec{x} \in \partial\Omega$$

$$\phi(\vec{x}) = d - (\vec{x}) \text{ if } \vec{x} \in \Omega-$$

And d being a Euclidean distance. Demonstrating how SDF contain similar properties to that of implicit functions. Other additional properties of SDF is their monotonic like nature across the interface. Meaning the function is either entirely non-increasing or non-decreasing [1]. The benefit of this is that it uses central differences that are beneficial for illumination and shading. Signed distance primitives are beneficial since many useful primitives such as a box, torus, or sphere to name a few examples can be generated. For example, as provided by Íñigo Quílez library of SDF a sphere can be defined as.

$$\phi(\vec{x}, d) = \vec{x} - d$$

Where ϕ represents the SDF function for a sphere while $\vec{x} - d$ represents the distance from the surface. Using the properties of SDFs as explained earlier, a negative distance indicates that we are inside the sphere. A positive indicates we are not hitting the sphere and 0 is the space which makes up the surface of the sphere. Another SDF of similar simplicity is the torus found in [12] while others e.g. capped cone triangle are more complex to solve. However, despite that each primitive is straight forward and once one has been implemented successfully the rest are trivial to implement. Other benefits of SDF primitives is that they retain their detail no matter how much you scale the primitive. SDF will always produce a smooth surface regardless of the distance between the camera and the primitive. Compared to other 3D modelling applications which require a large amount of detail to be applied to the object. Which in turn results in extremely high costs in terms of polygons.

One of the main reasons SDFs are used is due to their amenable composition. This is because it is easy to merge two (or more) SDF primitives into one single primitive using what is known as a blend function. The blend function is just one of the distance operations that are used to develop complex objects which are an example of constructive solid geometry (CSG) as mentioned in Section 2.

$$\phi = (1 - \beta) * \phi d1(\vec{x}) + \beta * \phi d2(\vec{x})$$

Here the blend operation is represented by ϕ. While β represents the blend factor. The first object where the blend function is initiated is represented as d1 while the second is represented as d2. The blend operation contains significant importance to our modelling application as the goal of the SDF was to grant the user ability to control the blend threshold. By giving them access to this functionality will, in turn, provide them with more control when creating complex models. Other SDF operations are union, intersection and subtraction which still contain significant importance in our application. For union, the formula is known as

$$\phi d1 \cup d2(\vec{x}) = min(\phi d1(\vec{x}), \phi d2(\vec{x})) = 0$$

Here the union operation is represented by ϕ while d1 and d2 represent the objects. However, a new attribute min is discovered. This checks for the minimum value between the two distances. The way in which these algorithms work is when camera rays are projected it performs its marching steps to detect the number of objects in the scene. If there is more than one object located within the scene we the proceed to evaluate the distance between the two objects and proceed to return the union. For further SDF you have intersection which performs the opposite of union function and proceeds to remove any element outside of the intersection. This relates to the union function since the main element that needs to change is min to max meaning the function is as follows

$$\phi d1 \cap d2(\vec{x}) = max(\phi d1(\vec{x}), \phi d2(\vec{x})) = 0$$

As $\cap$ represents the symbol for intersection (where $\cup$ was used for the union) This further demonstrates simplicity to perform these SDF to generate complex objects. Other methods such as subtraction can also be defined as

$$\phi d1 - d2(\vec{x}) = max(\phi d1(\vec{x}), -\phi d2(\vec{x})) = 0$$

With this operation, the user can take 2 SDF primitives and use one to remove a section from the other object. Example of this is provided in Section 4

3.2 Colour Blending

In this section, we will describe the effects of colour blending between two SDF primitives. Following the properties as proposed in paper [2] which uses an extension on transfinite interpolation where it takes the distance property of the object it contains and proceeds to interpolate its attributes to that of another object. The basic formula for attribute interpolation is described in the paper [2] However, this technique is linear interpolation and described as unintuitive. Paper [2] builds on this formula known as the space-time transfinite interpolation formula. With the interval as $0 \leq 1$

$$g\_1(x, y, z, t) = f1(x, y, z)^0(-t)$$

$$g\_2(x, y, z, t) = f2(x, y, z)^0(t - 1)$$

Here ^{0} represents the intersection, g_1 represents object1 with a boundary of t = 0. The function g_2 defines object2 with a boundary t = 1. Although the blending method used in paper [2] does not represent our SDF approach entirely, the formula can be further applied to colour distribution by allowing it to interpolate as the meshes blend. Overall this paper presented us with a valuable insight into the blending and mixing of colours.

By giving the user access to colour blending it gives them more control over their asset creation. For example, using the blend function, the two primitives which satisfy the equation will proceed to also blend their colours together along with the meshes.

3.3 Ray Marching

As briefly explained in Section 1. Ray marching is a 3D rendering method which casts multiple rays from the eye to detect the objects in the scene. Once the surface of and object is detected it proceeds to compute the attributes of the detected pixel e.g. its colour/shader etc. However, to remove computational costs ray marching casts rays based on approximation by what is known as 'marching'.

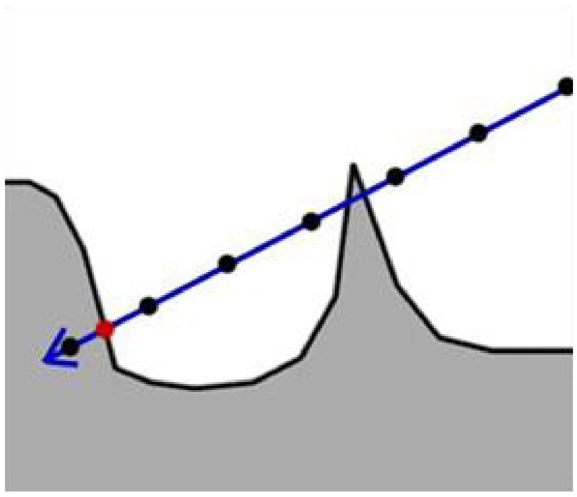

Figure 3: *Ray passing through thin geometry due to step size [16]*

As explained in [3] marching a ray takes a specific number of steps input by the developer and uses those the determine the number of checks before the step interval contains a zero value. Therefore, to determine if the end contains a zero value the end points of the steps are analysed and if a zero value is discovered it means the boundary was discovered at that location. If a negative or positive value is discovered it represents the boundary of the geometry to be within that step. With the ability to control the step amount it allows the developer to balance the priorities of the result in terms of favouring real-time speed over accuracy and vice versa. This is due to the ray marching algorithm being based on a camera dependent approximation of the surface location. [3] With the use of step sizes described in section 1. The total approximation of the geometry is based on several steps the ray marching algorithm takes.

Therefore, by having vastly reduced step sizes could result in ray to pass thin geometry and thus causing inaccurate results seen in figure 3. This is due to ray marching providing fewer restrictions on how it detects the intersection with geometry. If a ray is close to a surface and detects a collision within its threshold it proceeds to fill in the data hence why having too big of a step size increases the chances of inaccurate results.

This method is not to be mistaken with ray tracing. Although elements such as the casting of rays to detect geometry are similar. Ray tracing deals with what is known as explicit geometry such as triangle or quadrilateral meshes. Despite that having a good knowledge of ray tracing allows for better of understanding of the overall process of this paper.

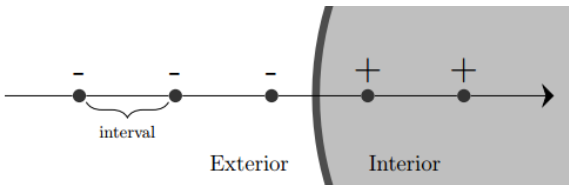

Figure 4: *Ray marching process*

3.4 Sphere Tracing

Due to the inaccurate nature that discussed in Section 6. We found that sphere tracing introduced by Hart [6] would be beneficial to use for our 3D application. Paper [5] demonstrates the properties of the sphere tracing formula.

$$pi + 1 = pi + d * f(pi)$$

Here by taking the starting position named $p0 = o$, a new position $pi + 1$ which is determined by advancing the previous position represented as pi. And the ray direction is shown as d Here the radius of the unbounding sphere at the position is known as $f(pi)$ creates the iteration. We used this algorithm to implement sphere tracing into our application. Furthermore, as suggested in paper [5] by making including a max iteration threshold to prevent the algorithm to prevent performance issues.

Since sphere tracing is a variation of the ray marching algorithm it contains similar techniques to ray marching. However, what makes sphere tracing a more viable algorithm in terms of our application is its ability to alter its step distance based on the surrounding geometry.

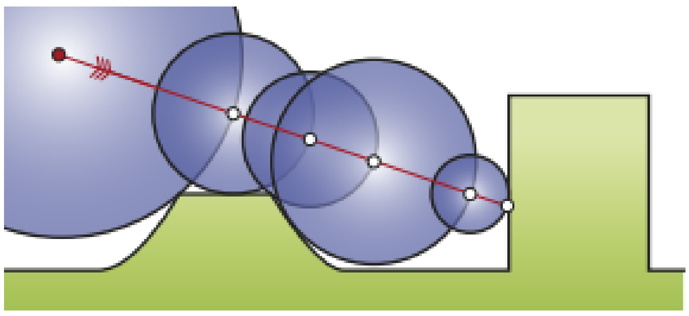

Figure 5: *Step procedure of the sphere tracing algorithm used by our application [5]*

As seen in figure 5 compared to that of figure 5 the step procedure size is different in terms of length. Since our application requires the user to develop complex geometry sphere tracing ensures that the primitives in the scene will be respected by the rendering and will produce accurate results.

However, sphere tracing contains limitations as well and thus problems do arise in our 3D modelling application.

Although sphere tracing respects the primitives in the scene and removes the risk of passing thin geometry it unfortunately has limitations when it comes to tracing edges. As sphere tracing approaches edges it reduces in steps. However due to the increase in step sizes it results in high performance cost. Furthermore, since the max step size needs to be controlled to prevent performance issues it causes the trace results to be inaccurate.

Despite this limitation sphere tracing still holds as a better technique for our surface rendering. Sphere tracing produces better performance in dealing with complex shapes. Since our modelling application expects users to create complex models we found that this method of rendering was the more optimal choice.

4 Parameterised Modelling Using Signed Distance Fields

The parametrised modelling format works by using what is known as a node editor. By creating a series of nodes, the user can call different segments of code to construct a 3D scene. For example,

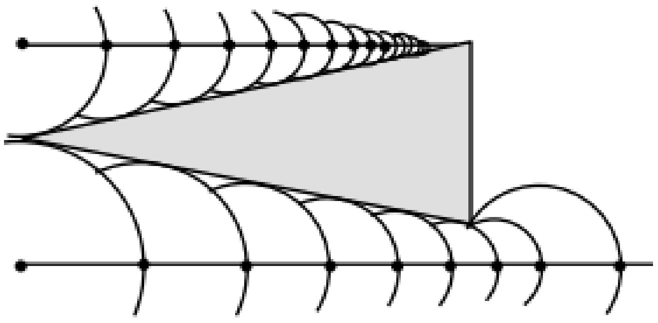

Figure 6: *Sphere tracing limitation [16]*

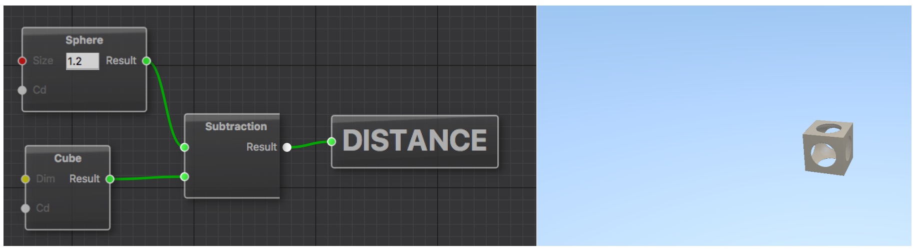

Figure 7: *The result of a sphere being subtracted from a cube.*

in our set up the user may wish to subtract a sphere from a cube to produce a newly formed SDF primitive. Therefore, the user would create the nodes labelled cube and sphere then proceed to call create a subtraction node. By using the output sections of the primitive nodes, the user can connect these to an SDF operation node (subtract) and finally to the "Distance"-node from which the scene will be traversed to generate the shader code resulting in a newly generated primitive as demonstrated in Figure 7.

The example described above is a really simple way to demonstrate the how SDFs can be used for 3D modelling. By using different combinations of these distance primitives and their operations, it allows for the creation of almost arbitrarily complex objects.

The real power of parameterisation can be utilised by chaining and reusing nodes. With the use of parameterisation, the models can contain a procedural element, where big changes can be made by simply adjusting a scalar value or changing an SDF operation. For example, by modelling and parameterising a simple cogwheel the user can then change the number of teeth it has as well as the scale of the cogwheel or its teeth by modifying only a few parameters. Once a model has been created, the node tree can be collapsed to a single where its inputs and outputs are controlled by the user (collapsing nodes is explained more in section 5.2). This can then be used later in the scene resulting in more and more complex scenes and objects. Another benefit of constructing scenes using parameterisation is the reverse engineering element that comes with it. Like that of Houdini, the node based structure of the application allows for multiple node trees to be saved and reloaded. This allows the user overlook each command that is being initiated and see the full breakdown of the scene, thus making the learning curve easier.

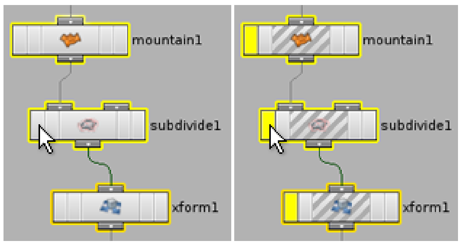

Figure 8: *Example of the node editor Houdini uses. Here it uses a displacement effect then proceeds to subdivide the polygons and further translate them*

Since the approach of our node editor was to have a similar format to that of Houdini [4]. We had to research potential elements the software contained that could be useful to our application. Not only did this require analysing nodes which would link well to our SDF but also ones that were achievable to develop in such a short time. By doing this, we would be building around the idea of an already successful format that contained similarities but in our own unique style. By using a simple node-based editor, the users with previous experience from similar software could pick up and navigate our application, while new users could quickly learn it with the help of visual cues (explained more in section 5.2) and almost instantaneous feedback. Due to the nature of implicit surfaces and node-based systems, even users with lack of experience in 3D modelling could get up to speed quickly as normally some amount of knowledge is required to produce models of decent quality.

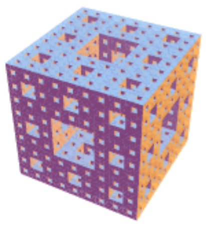

Figure 9: *Menger Sponge [18]*

To make everything parameterisable in the application we wanted to minimise the number of nodes without possible input while still allowing most of the nodes to work without the inputs being connected by exposing the default values inside a node. This meant that almost an arbitrary amount of inputs could be connected to a single output/value. Thus by modifying this single output everything connected to it would change accordingly. A simple example of this is given in Figure 10. The nodes that we considered did not need an input were nodes such as "Scalar" and "Time" that were considered to be pure output nodes. To harness the power of the parameterisation, even more, we implemented a so-called Copy-node, which allows a part of the node-tree to be iterated over a specified number of times. With copy-node the user can also use the number of the current iteration as an input to nodes connected to the copy-node, thus making recursive shapes such as Menger Sponge (Figure 9) easier to create.

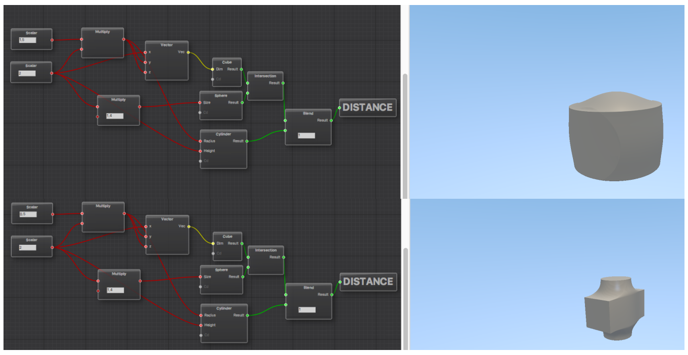

Figure 10: *By changing a single value the whole nature of the generated object changes*

An example of the cogwheel can be seen in Figure 11, where a single cogwheel is modelled with the combination of cylinders and cubes. The base of the cogwheel is modelled by blending and subtracting cylinders from each other till the desired result is achieved, then the base is blended with cubes (or teeth) that are revolved around the edge of the base. The distance of the teeth from the centre is controlled by the radius of the base, which is defined by the user. The amount of pegs is also controlled by the user which also affects the angle by which each tooth is rotated around the centre. Finally, the cogwheel is duplicated and translated each both copies are rotated around their Y-axis over time to create a simple animation. For a video of the animated cogwheels, refer to the examples provided with the submission.

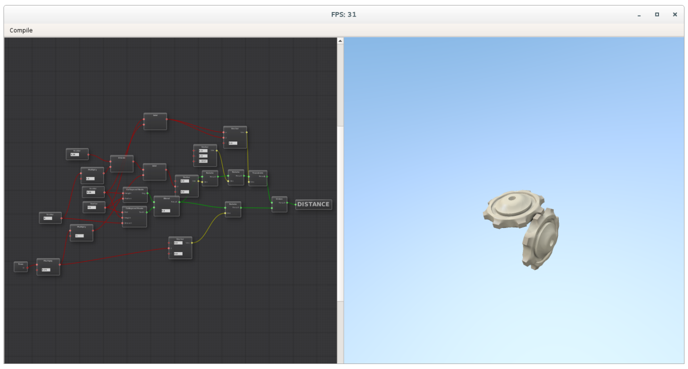

Figure 11: *Cogwheel modelled and duplicated using our application*

5 Implementation

For our 3D parameterised modelling application we found it would be best to develop the software in C++ using Qt5 and OpenGL.

The basic outline of the application's workflow

- The user generates the scene inside the node editor using the nodes provided and connects the final result to a static "Distance"-node.
- When the user wants to see updated feedback of their scene, they will press "Compile" which starts the traversal of the node tree.
- The node tree traversal begins from the final output node and anything connected to this tree will be traversed.
- The traversal is done recursively and depending on the type of the node, different operations are performed. A transformation matrix is also carried through the recursion thus passing the correct transformations to each primitive.
- When reaching a leaf node, the corresponding shader code is returned back to the top of the tree.
- Finally, the generated shader code is compiled to a fragment shader that is then used to render out the scene.

To display the generated scene, the application uses OpenGL and GLSL to render a screen sized quad to which the scene is then rendered inside the fragment shader. As the scenes are built using a set of predefined distance primitives and operations, the distance functions for them and the rendering are prepended and appended to the final shader code. This also meant that the shader code generated by the node-tree traversal only needed to contain the corresponding shader function calls and transformation multiplications. Most of the distance functions used were provided gathered from the article by Íñigo Quílez [13].

5.1 Rendering the scene

The basic rendering algorithm is outlined as pseudocode in Algorithm 1. It follows the simple sphere tracing algorithm [6], the modification that we made to the distance estimation functions was that they would also return the colour of the primitive that was traced with its distance estimation. The reason we chose sphere tracing was its ability to trace even thin surfaces, which is explained more in section 3.4.

When casting a ray, the shader code generated from the node editor scene is evaluated and the trace result is returned. Trace result consists of the distance estimation and colour but could be extended to include more material properties like the surface type etc. that could then be used to shade the pixel differently. If the distance estimation is less than a predefined trace precision a

Algorithm 1 Rendering algorithm

```
 1: foreach pixel do
 2:     Create a ray
 3:     Calculate the background colour (sky)
 4:     Trace the distance and the colour of the scene
 5:     if distance is less than trace precision then
 6:         Calculate the position, normal and reflection ray of the intersection
 7:         foreach light in the scene do
 8:             Accumulate the diffuse, ambient occlusion, specular highlights and shadows
 9:         end foreach
10:     Apply fog, vignetting and other effects.
11:     Apply gamma correction
12:     Return final colour
13: end foreach
```

hit was found, thus we can proceed with the lighting calculations. The trace step and maximum distance from a surface were limited to reduce the computational cost of the tracing.

The normal for the point is calculated by offsetting the ray slightly and calculating the distance estimations around the point, thus giving an estimation of which direction the normal of the surface is pointing. Then the ray is reflected around the normal to be used for specular calculations. Using this information we perform basic lighting calculations like diffuse intensity, specular highlights etc. to get the colour of the pixel. Due to time limitations with the implementation, the lights are predefined and not exposed to user control. At the end of the rendering loop, additional details such as fog and vignetting were applied to increase the aesthetics of the scene.

Other effects such as ambient occlusion and soft shadows were also implemented with little to no effect on the real-time results. For shadows, we ray march along a vector from the point being shaded to a light and if an intersection is found, the point is in shadow. To get soft shadows, we can use the just calculated distance estimation by applying a darker shadow when we are closer to the surface, thus creating the penumbra and shadow falloff. As the same calculation was required to for normal shadows, we get soft shadows with basically no increased computational cost.

5.1.1 Colour blending

As one of the aspects we wanted to bring in to the application was material blending (only implemented as colour blending due to time limitations) we used the method of transfinite interpolation as explained in [2]. To get the time variable required to do the attribute interpolation, we use a polynomial smooth minimum function for blending explained in [19] which tackles the discontinuity of min() that is used for union. From the polynomial smooth minimum we can get the time variable using

$$\text{clamp}(\ 0.5 + 0.5 * (b - a)/k,\ 0.0,\ 1.0\ )$$

Where a and b are the distance estimations of the objects being blended and k is a user-defined blend factor. This gives us the time variable that can be used for the material interpolation. A more detailed explanation can be found in [19]. The colour (and shape) blending is demonstrated in Figure 12

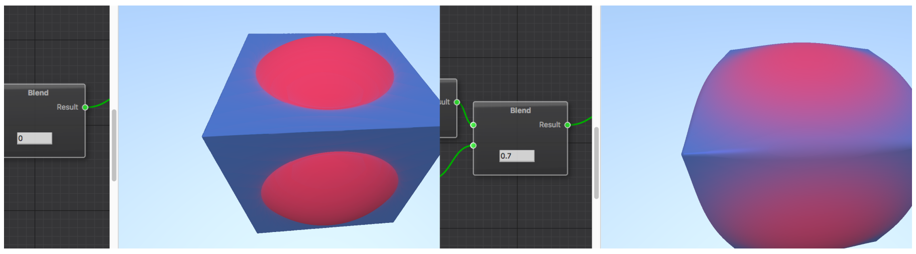

Figure 12: *Left: Blend factor set to 0, Right: Blend factor set to 0.7*

5.2 Node Editor

To reduce the amount of time spent in the designing stage, we decided to use an existing node editor developed for Qt. This would let us focus more on building the interface between the node editor and the shader code and not invest unnecessary resources into developing a node editor from scratch. We ended up using a node editor that is currently being developed by Dmitry Pinaev [].

During the debug phase, we used the hard-coded values for the attributes provided in [13]. These values were later exposed to the user inside the node editor to for proper parameterisation. Due to our main focus revolving around creating an effective 3D modelling application with SDF we felt this method of user control was necessary.

5.2.1 Creating a scene

When using the program the user is given a node scene with a single static node called "Distance", this is the final node that will determine what is shown in the scene and what is not. Anything connected to the final distance-node will have an effect on the shader code generation. When the user wants to update the scene, they will need to press "Compile" as this tells the application to traverse the node tree and generate the required shader code. Initially, we had the scene update whenever a change was made to a node in the node tree, but this proved to be inconvenient as the shader would have to be recompiled after every minor change.

The node connections in the editor are color-coded based on the data type that is being propagated, thus giving the user a quick indication of what can be connected where. The nodes are also categorised based on their function and this information is used the perform different operations when the node tree is traversed.

5.2.2 Modifications

The node editor was working well for general purposes like data propagation but modifications were needed for the node graph to meet our requirements. The first obvious drawback with the node editor was that an output could not be connected to multiple inputs, as this was one of our requirements for the parameterisation. The developer of the node editor had this functionality listed in his todo-list, but there was no indication of when this would be implemented so we ended up implementing this feature to the node editor ourselves. We showed the developer our modifications and he later informed us that he took ideas from our development to help improve the functionality of his software.

To make the editor more user-friendly when dealing with larger scenes, we decided to give the user ability to collapse bigger node trees to single nodes or "functions" for which the user can specify inputs and outputs. When collapsing the nodes the new node is just a reference to its selected inputs and outputs, thus when traversing a collapsed node the traversal is redirected to the selected input and/or output. While this works as expected when dealing with the scene, it posed new problems when using the save/load functionality of the original node editor. As the collapsed node contained references to the nodes inside it, when loading a saved scene, the nodes referenced might not exist when the collapsed node was loaded from the save file. This forced us to rewrite a lot of the saving/loading functionality to allow for the usage of these custom functionality nodes.

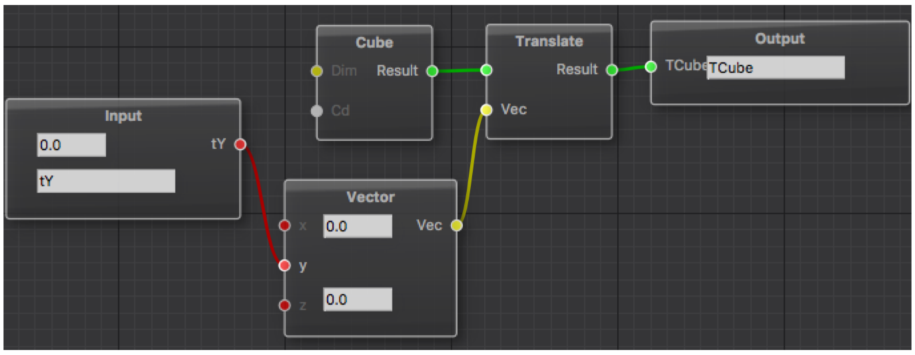

Figure 13: *Simple node tree*

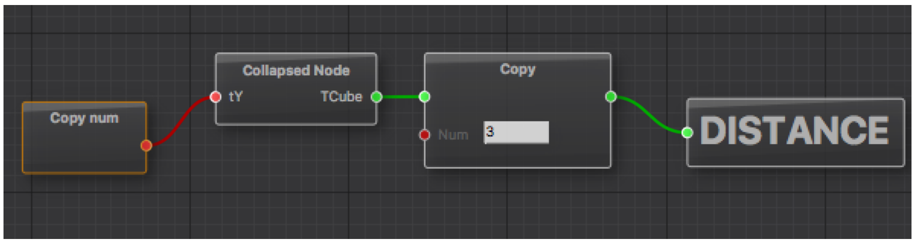

Figure 14: *Node tree shown in figure 13 collapsed*

Other minor modifications to the node editor were made to have it match our needs. These modifications include, but are not limited to; improved the readability of the graph (e.g. coloured connections), bigger scene area by default, custom menu categories and sorting of elements under these categories, connection replacement and so on.

5.2.3 Traversal

When traversing the node tree, we begin from the static "Distance"-node and recursively iterate through the connected nodes. A transformation matrix is passed along the recursion to pass all the required transformations to the primitives. As distance primitive transformations inside the shader are matrices, we offloaded as much of the matrix calculations to the CPU as possible. This was done to reduce the number of calculations needed to be performed on the GPU and thus improving the performance of the visual feedback.

Each node in the tree also contains information on its type. The node types in the current implementation are: Primitive, Transform, Mix, Vector, Scalar, Color, Io(Input/Output), Collapsed and Copy. The type is then used to tell the program how to handle any given node. For example, a transform node always contributes to the current transformation matrix, mix-node requires two distance functions as its inputs, vector and scalar nodes are "ignored" as they pass their data upwards to the node that they are connected to and a primitive node returns its corresponding distance function call. A simple outline of the tree traversal can be seen in Algorithm 2.

Algorithm 2 Node tree traversal

```
 1: for current node do
 2:     if node type = primitive then
 3:         concatenate distance function to shader code
 4:     else if node type = transform then
 5:         apply the node transform to current transform
 6:     else if node type = mix then
 7:         get the distance function inputs for the mix
 8:         concatenate distance function to shader code
 9:     else if node type = copy then
10:         loop through the node tree in input for x times
11:     else
12:         ignore node
13:     foreach connection to node do
14:         recurse the nodes and pass transformation down
15:     end foreach
16:     return shader code
```

6 Results

In result, we found that there are many benefits from using implicit primitives to model a 3D scene. Although limitations do arise from our application we find that it contains the functionality to produce and alter complex models which would often be difficult to accomplish using an explicit modelling application. In this section, we will demonstrate examples we have developed using our application and the approaches we took to build said example.

One of the examples suggested demonstrating our application was a cog see /refcog. With this, we could take multiple SDF operations as well as primitives and demonstrate how the user could create a generic cog and procedurally control the different aspects it contained. E.g. How many teeth, teeth size and overall scale of the cog. Furthermore, the user could choose different design features such as subtraction to create a hole or blend a cylinder to create a welded pole to go through the centre of the object.

Furthermore, with the addition of the time node we could rotate our cog in the chosen x, y, z orientation meaning you could have an animated mechanism of cogs rotating. Due to the multiple steps and the procedural nature of the object it allowed us to demonstrate many aspects in the CSG modelling format.

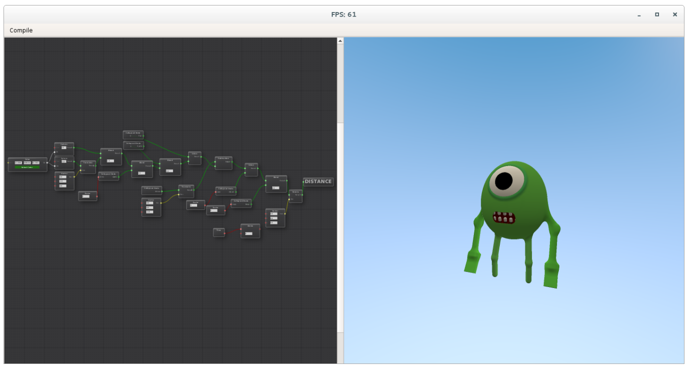

Figure 15: *Blocked out model of Disney Pixar's Mike Wazowski.*

6.1 Limitations

Despite our application fulfilling the requirements and possessing the ability to construct 3D models in a parameterised format. We found there were a few limitations in terms of performance as well as functionality that could have been improved upon or was simply a limitation that could not be prevented due to the scope of this project.

One of the limitations is that the user is currently restricted to the node based format that we supply. This makes it somewhat unintuitive to simply move and object to a specific location as the node process would have to call a vector node, then a transform node and further proceed to connect the correct connections. Compared to that of Houdini it may be beneficial to later include something similar to the edit node where the user does not require mathematical precision [15]. Here the user can take control of the scene using a transform tool like the ones seen in other 3D applications. However, this is merely speculation and possibly might not be within the scope of our application.

Another limitation is the real-time performance as the complexity of the shader code grows. By zooming in close to the object it proceeds to drastically affect the performance of the scene. Thus, the user must favour between being close to the object with lesser performance or further away with the better performance. In addition to this limitation with the edge rendering issue of the sphere tracer when the user zooms into the object the geometry begins to produce disfigured geometry. This is due to the step threshold of the sphere tracer reaching its limit and therefore producing inaccurate results. However, this limitation is not noticed and a formidable distance from the geometry.

7 Conclusion

We produced an application which demonstrates how SDF primitives can be manipulated by the user to construct 3D scenes through a visual node base editor. Furthermore, our application helped remove the requirement of intermediate modelling knowledge that would be needed in other modelling applications. With the addition of SDF operations as well as specific operations it made the modelling process more efficient. Personal operations include the addition of colour selection where the user can alter the primitives colour attributes. This was then further enhanced with the adaptation of the blending node, by providing control over the blending threshold it allowed more visually appealing and accurate construction of geometry.

Furthermore, the adaptation of the node editor provides the application with a more visual modelling format. Thus, the user can see the breakdown of their scene which provides a reverse engineering approach to their creation. Other examples is the ability to revert to a previous state to make alterations to an already developed model.

In conclusion despite the limitations and areas for improvement, we feel satisfied with the outcome of this application. By building complementary operations for both SDF and our own personal additions it helped create an efficient application for artists. It's hard to say whether signed distance modelling can stand up to other explicit 3D modelling applications. However, from the results produced by our application, as well as the continuous development of better functioning hardware, there is room for a suggestion that signed distance modelling could potentially see its way into an industry format. We found it enjoyable to implement this research topic with the result not only being a functioning application where complex models could be demonstrated. But it also possesses the opportunity for further development.

7.1 Future Work

For future work one of the areas that was discussed amongst us was the viewport. Compared to other modelling applications the user has more control over the screen space in terms of which camera angle they can view their work from. For example in software such as Houdini, the user can split the window into four separate views for the orthographic, side, top and front view. Furthermore, the user can choose which camera to snap to granting the user ability to construct their model more accurately. For an application, which relies on mathematical accuracy having this feature would allow the artist to not only view their model from multiple angles at once but also specify which angle they wished to view their model separately.

Enhanced sphere tracing was an option we considered implementing for generating our SDF primitives to the scene. As implemented in the paper [5] enhanced sphere tracing has proved to be a more efficient method compared to that of normal sphere tracing. This is due to sphere tracing having a substantial limitation when dealing with edges which were apparent in our application. With enhanced sphere tracing it represents an over-relaxation technique to deal with the limitation recently discussed. With the implementation of this to replace our current method, it could result in increasing the performance of our application. Therefore, this could potentially be left for open discussion for future work.

Another area for improvement was the implementation of our own shader pseudocode language that on compile would be fed back to the user. Although our application wishes to respect that the user doesn't need access to the backend of the application we did consider that there may be some users who will gain a benefit of having a breakdown of the shader. With the use of the BNF code format, the user could compile their scene and proceed to open a provided text editor which would display the pseudocode. However, due to time constraints, we were unable to test or begin implementing this method. However, it was an interesting element of discussion and one we would potentially like to consider further for future development.

Other future work included providing the user more control over the materials displayed in the viewport. We began providing this through the implementation of our colour shader and the use of blending colours. However, we felt that more control could be provided to increase the ascetic look of their work. Parameters such as specular intensity, Specular roughness, and subsurface scattering can be added to the shader and furthermore be given access to the user for control. With this, it could allow artists to create not only complex shapes but to have them contain realistic reflective parameters. We found this to be a particularly interesting idea for future work as it could potentially provide the opportunity to the artist to create realistic materials.

References

[1] Tim Reiner, Gregor Mückl and Carsten Dachsbacher, 'Interactive Modeling Of Implicit Surfaces Using A Direct Visualization Approach With Signed Distance Functions' (2011) 35 Computers and Graphics <https://cg.ivd.kit.edu/downloads/IntModelingSDF.pdf> accessed 22 January 2017.

[2] Mathieu Sanchez and others, 'Space-Time Transfinite Interpolation Of Volumetric Material Properties' (2015) 21 IEEE Transactions on Visualization and Computer Graphics <http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6894217> accessed 22 January 2017.

[3] Tuomo Rinne, 'Efficient Evaluation Of Functionally Represented Volumetric Objects' <http://eprints.bournemouth.ac.uk/22562/1/Rinne,Tuomo_MRes_2014.pdf> accessed 22 January 2017.

[4] 'Home | Sidefx' (Sidefx.com, 2017) <https://www.sidefx.com/> accessed 22 January 2017.

[5] Benjamin Keinert1 and others, 'Enhanced Sphere Tracing' [2017] STAG: Smart Tools and Apps for Graphics (2014) <http://erleuchtet.org/ cupe/permanent/enhanced_sphere_tracing.pdf> accessed 22 January 2017.

[6] J. C. Hart, D. J. Sandin and L. H. Kauffman, 'Ray Tracing Deterministic 3-D Fractals' (1989) 23 ACM SIGGRAPH Computer Graphics <http://graphics.cs.illinois.edu/sites/default/files/rtqjs.pdf> accessed 22 January 2017.

[7] Stanley Osher and Ronald P. Fedkiw, 'Level Set Methods: An Overview And Some Recent Results' (2001) 169 Journal of Computational Physics <http://physbam.stanford.edu/ fedkiw/papers/cam2000-08.pdf> accessed 22 January 2017.

[8] Brian Wyvill, Andrew Guy and Eric Galin, 'Extending The CSG Tree. Warping, Blending And Boolean Operations In An Implicit Surface Modeling System' (1999) 18 Computer Graphics Forum <http://pages.cpsc.ucalgary.ca/ blob/ps/blobtree.pdf> accessed 22 January 2017.

[9] S Osher, R Fedkiw and K Piechor, 'Level Set Methods And Dynamic Implicit Surfaces' (2004) 57 Applied Mechanics Reviews <http://link.springer.com/book/10.1007/b98879#page-1> accessed 22 January 2017.

[10] Jaemin Shin, 'Mesh Generation With A Signed Distance Function' (2017) 1 OFFICE OF GRADUATE STUDIES <http://elie.korea.ac.kr/ cfdkim/papers/Jeamin.pdf> accessed 22 January 2017.

[11] Chandrajit Bajaj and others, 'Introduction To Implicit Surfaces' <https://books.google.co.uk/books?hl=en&lr=&id=T3SSqIVnS4YC&oi=fnd &pg=PR10&dq=Introduction +to+Implicit+Surfaces&ots=BvD3Kh-L-R&sig=o_7WNdKNYXOYX1KJKwRl0ZO6x_8#v =onepage&q&f=false> accessed 22 January 2017.

[12] Íñigo Quílez, 'Ray Marching Distance Fields'

<http://iquilezles.org/www/articles/raymarchingdf/raymarchingdf.htm> accessed 22 January 2017.

[13] Inigo Quilez, 'Fractals, Computer Graphics, Mathematics, Demoscene And More' (Iquilezles.org, 2017) <http://iquilezles.org/www/articles/distfunctions/distfunctions.htm> accessed 22 January 2017.

[14] 'Shadertoy' (Shadertoy.com, 2017) <https://www.shadertoy.com/view/Xds3zN> accessed 22 January 2017.

[15] 'Edit - Houdini Online Help' (Sidefx.com, 2017) <https://www.sidefx.com/docs/houdini9.5/nodes/sop/edit> accessed 22 January 2017.

[16] Gabor Liktor, 'Ray Tracing Implicit Surfaces On The GPU' [2017] Department of Control Engineering and Information Technology <http://www.cescg.org/CESCG-2008/papers/TUBudapest-Liktor-Gabor.pdf> accessed 22 January 2017.

[17] Unity3d Hackweek 2016: Signed Distance Field Rendering With Node Based Editor (jfryer3D 2017). <https://www.youtube.com/watch?v=M3-TdTkScOU> accessed 22 January 2017.

[18] Weisstein, Eric W. "Menger Sponge." From MathWorld–A Wolfram Web Resource. <http://mathworld.wolfram.com/MengerSponge.html> accessed 22 January 2017.

[19] Íñigo Quílez, 'Smooth Minimum' <http://iquilezles.org/www/articles/smin/smin.htm> accessed 15 November 2016.