Malleable environments composed of Signed Distance Fields

Einars Bruveris
BSc Computer Games Applications Development, 2023

School of Design and Informatics
Abertay University

Table of Contents

Table of Figures

Acknowledgments

I would like to thank Erin Hughes for all the guidance and support throughout the making of this project. Her expertise in the field has been invaluable.

I also would like to thank my partner who has been a great support and was very patient when I went on yet another tangent about non-polygonal geometry.

I would like to thank my friends and family with a special shout-out to the graphics team.

And finally Tibbles/Beans/Chonk the kitten for being a cute and playful and seeing me through the composition of this paper.

Abstract

Signed Distance Fields are an alternative representation that can be used for rendering 3D geometries, It allows to render merge and meld objects into one seamlessly. It is widely used in visual effect and in simulations of gases and fluids but is less common in real time game applications for geometry representations.

The goal of this study is to highlight the careful considerations that have to be made when using SDF for their merging properties to construct the geometries within a game environment in a way that aims to be computationally and memory conservative whilst maintaining accurate representation of the source functional geometry.

This project outlines the implementation of sparce texture and its indirection structure which was picked to be an octree. And a dense alternative texture that demonstrated how an indirection structure can impede the render performance. As well as explores a traditional way to rendering an implicit using marching cubes to perform fidelity testing.

Gathered results have shown the implications of using a dense structure against the sparce volume with input variables such as scene complexity, scene density and intricacy. Output render fidelity of discrete approaches have been compared against the polygonal mesh and ground truth fully functionally defined SDF scene.

The project concludes that the data structure design have to be heavily informed by the intended use of the geometry, universal approach is not advised and constructing a fully editable environment sparsely can be computationally efficient for revaluations but the resulting structure is less optimised for rendering.

Keywords

Signed Distance Fields, Sparce Voxel Octrees, Raymarching,

Abbreviations, Symbols, and Notation

SDF - Signed Distance Field

NURBS – Non-Uniform Rotational B-splines

SVO – Sparse Voxel Octree

CSG – Constructive Solid Geometry

Brick – 3D texture block that contains an SDF Isosurface within.

SRV - Shader Resource View

UAV- Unordered Access View

GSM- Group Shared Memory, memory accessible to all threads within a single group within a dispatch.

Brush – a piece of geometry represented with a functional SDF

Scene – Collection of Functionally defined brushes sampled as a single object.

Chapter 1 Introduction

Real-time surface deformations in games provide a great degree of immersion as there is an expectation for certain object types and surfaces to deform, blend, or otherwise leave indentations on the surface. It is also expected that different organic structures will be smooth, such as those made of clay or wax. If those structures are animating i.e. melting there is an expectation to not create any hard edges in the process. The triangle can be used for these applications but an increasingly larger amount of them is required to correctly represent soft surfaces that expand and shift. Additionally, it is difficult to transform explicit triangle geometry between different shapes. For instance, accurate simulation of Ditto Pokémon (Pokémon Corporation) would be difficult in real-time as morphing would require accurately mapping all vertices to all possible targets for interpolation to go smoothly.

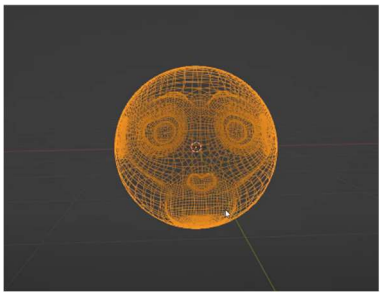

Figure 1Suzzane morpherd into a sphere

Figure 1. demonstrates an example of a morph which is very easily performed with triangles morphing into a sphere done via normalization of all the positions though can yield varying results depending on the position of the Pivot position. Although, the non-uniformity of spacing between the vertices can still be observed, which can cause visual artifacts due to over-stretching of the faces.

Implicit representations of geometry such as Signed Distance Fields have been widely used for Constructive Solid Geometry (CSG) in Computer-Aided Design (CAD) i.e., Solid modeling as they provide great scalability and eminent blending abilities. In Animation and VFX conversion to an

SDF is performed to allow for Volumetric effects otherwise not possible or more complicated to perform using triangle meshes as previously described.

These representations also had been widely used for different scientific visualizations to render objects that are otherwise too difficult to display using conservative amounts of triangles and traditional rasterization. Such as fractals or MRI scans etc.

In Games and more specifically game engines CSG is commonly used to construct geometry, such as Geometry Brush tools in Unreal Engine that is later stored in the form of polygons but there have been a few attempts that take this further and abandon the polygons altogether and construct worlds that are entirely made from SDF representations to utilize similar techniques that are common in offline rendering. This allows for soft, malleable environments, morphing shapes, and volumetric effects but is met with challenges of storing and handling large amounts of volumetric data, this way requiring careful consideration for underlying data structures purpose, construction, memory layout, and speed of traversal.

1.1 Research question

How do the sparse discretization approaches of signed distance fields impact the computational latency and the visual quality of the geometry when constructing and rendering arbitrarily editable environments?

1.2 Aims

This project aims to evaluate an Octree-based approach to rendering opaque SDF geometry
In terms of visual fidelity, and computational latency for construction and rendering.
An application will be developed that would allow to build and render discrete representations of SDF geometry entirely on the GPU as well as provide the option to render the original geometry using a purely

functional approach or polygon model that would be rendered using a raster pipeline.
This way providing us with the means to capture and evaluate the quantitative differences between pipelines.

1.3 Objectives

- Research the rendering pipelines of SDF-based geometry
- Implement the Marching cubes algorithm to create a polygonal alternative.
- Implement Functional Raymarching to create ground truth for fidelity testing
- Implement Texture Raymarching as a step towards Octree
- Implement Sparce Octree and provide the means tray trace it.
- Evaluate the computational latency of construction and rendering of the sparse approach to the dense approach.
- Compare the output of the discrete raymarching to the non-discrete representation as well as the polygonal alternative (marching cubes) for visual fidelity.

1.4 Hypothesis

The method of testing outlined in section 3.6 should demonstrate that fragmentation of space would allow for faster construction and volume re-evaluation of the discrete representation but would introduce additional computational latency during rendering. It would also enable us to save memory as we do not need to store just unreferenced by the octree.
Although Octree would negatively impact step complexity it should reduce the step count allowing some rays to terminate sooner which would demonstrate how valid is this application.
The Fidelity of the discrete representation should remain comparable to the highly tessellated polygonal alternative.

Chapter 2 Literature Review

Approaches of non-polygonal geometry for surface rendering include vector graphics, signed distance functions, contours, NURBS, etc. These approaches depart from traditional polygonal pipelines and require careful design consideration regarding memory throughput, and computational latency. In this chapter, we aim to

2.1 Signed Distance Functions

$$f(p) = \mathrm{v}$$

Figure 2 SDF function with input p and output distance v

Signed Distance Functions accept a world space position as an input, and return a value v, which is an orthogonal distance to the surface (the closest surface to the point). v is positive if the closest point is outside the geometry, and negative if it is inside. 0 is returned if the point lies on the surface of the geometry.

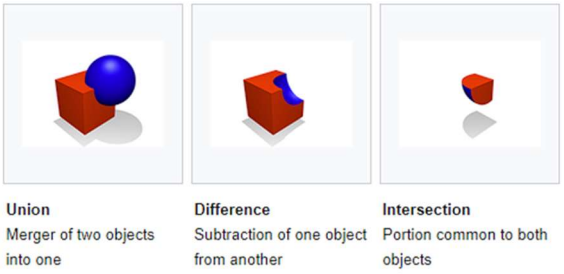

Figure 3 SDF Boolean Operations (Wikipedia)

To create more complex objects and scenes, signed distance functions can be combined using Boolean operations which determine how objects interact with each other. Figure 2 shows the possible operative(?) combinations. Smoothing can be applied to the functions to create smooth transitions between fields.

These operations are the basis of Constructive Solid Geometry and are usually organized into binary trees to form more complex shapes. Though, in-game applications of the technique, such as Dreams (Media Molecule, 2020) and Claybook (Second Order, 2018), these edits are stored as simple lists of brushes and Boolean operations combining them

2.1.1 Benefits and Trade-offs

Functional representations are nontrivial to define for arbitrary geometries. Although there is work in the field of Machine learning to fit a model to return an SDF of an object, those are unsuitable for real-time applications for now. When baking a field into a texture to create Discrete representations which will be further discussed in section 3, we need to make careful considerations for memory management and resolution as well as the format the SDF is stored, because it often does not directly benefit from the higher floating-point precision and at insufficient resolutions, the detail is lost. One of the unavoidable consequences of the discretization process is the loss of sharp edges, though, for this research, this is a minor setback. Fine and thin details are also easily lost in a discrete representation.

Figure 4 Discretisation loss of quality example (Epic Games)

But as Signed Distance Fields encode a gradient of distance values into a volume, it provides several benefits that make this representation useful. SDF can be used for computationally fast and fully dynamic lighting for Global Illumination approximation as fine detail might not be required, examples of this can be found in modern engines like Godot with SDFGI solution and Unreal with Lumen (the software raytracing aspect)
SDF allows producing a bent normal, false normal oriented along the gradient i.e. direction of least occlusion, which is used for ambient occlusion calculation that is robust.
SDF is also often used for various volumetric effects like fluids thanks to the computationally inexpensive way of merging fields.

Because of the extra information, SDF is also good at reconstructing an isosurface even at low resolutions this is commonly used in text rendering. Green, C (2007). In 3D this manifest in a lot higher level of detail for smooth surfaces while requiring comparable memory amounts and lower artistic involvement with LOD meshing considerations. Distance information accessible in the GPU allows for particle collision detection.
On the other hand, the storage of the SDF needs to consider other parameters that might be desirable such as color as SDF cannot be textured in the same way as polygon mesh, for that resorting to Bi-planar or Tri-planar mapping which demands more operations.

2.2 Rendering

2.2.1 Raymarching

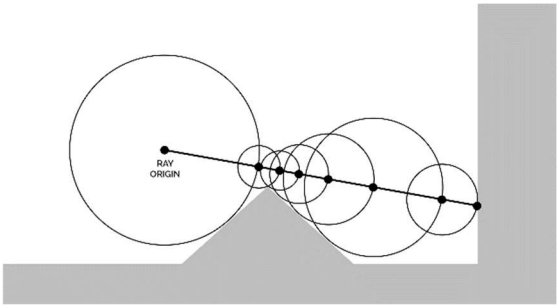

Figure 5 Raymartching step visualisation (Wikipedia)

Raymarching or Sphere tracing, a form of raytracing introduced by Hart, John C. (June 1995). This technique renders a 3D SDF to a screen buffer or texture.
Raymarching works by advancing the ray along a ray's direction with the step length or magnitude set to the distance sampled from an SDF, minimizing calculations in space. The raymarching terminates when the sampled distance becomes too small, or the ray reaches the maximum count of steps.
This technique is widely used to this for displaying implicit surfaces that can be functionally defined as signed distances such as various fractals.

Inigo Quilez (2013) defines plutonic solids as SDF implementations including additional surface information such as central difference normal evaluation and trilinear texturing.

Raymarching presents several disadvantages. Increasing sampling cost as the scene complexity increases can be resolved using space partitioning and bounding volumes, or baking SDF values into a texture/buffer. This will be further discussed in section 3.
Raymarching works even if the SDF is precalculated and stored in memory as a 3D texture as hardware Bilinear sampling, or Trilinear if the mipmapping is utilized, produces a close enough approximation of SDF Value. This on the other hand exchanges the computational cost of the functional representation to the high memory requirement of 3d textures and computational latency of frequent sampler invocations.
Parallax occlusion mapping technique, ray marches a 2D height map where Z depth is a function of x, y to simulate the depth on an otherwise flat surface, with the UV coordinate of the rasterized triangle as the entry point for raymarching and performing it in the fragment shader. Zinc. J (2013)

While raymarching with SDFs provides flexible surface representations, the SDF function sampling count increases considerably when the algorithm is close to the surface as it can only take very small steps. Acceleration of steps can be achieved by Approximating flat spans of a surface to achieve larger step lengths. Dynamically adjusting the resolution of the traces can also minimize render times. Speed up can also be achieved by performing several passes of Cone tracing, each smaller cone trace starting where the previous pass completed allows maintaining the final image fidelity and resolution while still performing the render in real-time (Balint C. et. al.), A similar approach is used in Claybook.

2.2.2 Marching Cubes

SDF surface representations are not coupled to raytracing or raymarching and can be easily converted into a polygon mesh.
Marching cubes, or polygonizing a scalar field, provide a conversion from a general surface into discrete polygon triangles for the traditional rasterization pipeline. Bourke, P. (1994)
The major downside of the marching cubes algorithm for visualizing implicit surfaces is the lossy conversion resulting in lower visual fidelity. Triangles excel at rendering flat surfaces but greatly suffer when displaying smooth, curvy, and otherwise organic-looking objects to retain fidelity and very high polygon density. (Evans. A, 2015).
With emerging virtual geometry applications such as Nanite (Epic Games, 2022) Polygon count becomes less of a problem as geometry can be loaded from SSD on demand for out-of-core rendering. The drawback of nanite for the time being is that it is not a structure that can be modified online.

2.2.2. Analytic Voxel Intersection

Due to the limitations of the raymarching alternatives have been proposed such as the work of Hansson-Söderlund et al. (2022) on the analytical evaluation of the ray - SDF grid intersections. This work expands on Parker et al. (1998). Here, the surface inside a voxel is derived as a cubic polynomial, which can be solved analytically by finding the closes root i.e. min root. This allows for producing results that are more expressive than Marching cubes while not sacrificing the smoothness of the curves while taking up as much space as little space as 2 32bit vertices. It is also important to note that this approach resolves the issues of raymarching whilst supporting a variety of data structures commonly used in signed distance field geometry. On the other hand, it concludes that the best results are achieved using the DXR BVH acceleration structure with the downside high memory requirement. While DirectX 12 Ultimate adoption is high the average consumer GPU does not yet reliably support raytracing (Steam hardware survey, April 2023)

3. Data Structures

Rendering functional representations of geometric plutonic as an SDF benefit from dynamic resolution matching output targets with relatively low latency intersection tests. However, complex organic shapes created from unions, subtractions, and intersections of a hierarchy of plutonics create a combinatorial complexity explosion. Beyond computational latency concerns, kernel–pixel/compute shader program needs to be recompiled at runtime each time a modification is made which is not suitable for real-time game applications where the mesh modifications can be dynamic. (Reiner, T et.al). This approach is suitable for offline applications like high-precision CSG where high scalability is crucial. For dynamic applications, a discrete representation is used, such as 3d textures, as it avoids the computational latency of computing through all plutonic of a scene. Additionally, space partitioning data structures like octrees are often utilized to reduce the need of storing sections that do not directly contain an isosurface.

3.1 Baking SDF to a Texture

The game engine bakes SDFs into 3D textures. This
Both Dreams' Bubblebath Engine (2014) and Claybook (2019) use 3D textures as fundamental blocks in surface representation but manage them differently. Claybook prioritizes players' ability to mold the environment and thus opts for a monolith-dense structure containing lots of space that can at any time in the game be filled by the player during gameplay. Dreams on the other hand try to minimize object size and avoid storing blocks containing no isosurface, this makes modifications more complex.
Additionally, As SDF is infinite, introducing modifications to a texture throughout its size becomes too expensive with high resolutions. There exist several approaches that address that that are discussed in this section.

3.1.1 Mip Mapped Dense Textures as Used in Claybook (2018)

In Claybook, the entirety of the play space, including the space is baked into a 3D texture that is persistent for a level and resides entirely in the GPU memory, 1024x1024x512 texture, the entire volume is only ever constructed once per level. But as the point of the game is to mold the surrounding environment this cannot be performed per frame. To avoid updating the entire volume, the SDF range is limited to 4 voxels to values between [1;-1] which allows them to be stored in 8-bit snorm integers. Mip maps are constructed for the volume reducing the band by 2 voxels each time, lower resolution maps containing larger values. This allows for larger steps during raymarching, minimizing redundant intersection tests. This also allows for efficient local modifications to the volume as only affected within a 4 voxel radius need to be recalculated. Mipmapped data leverages hardware trilinear sampling resulting in XYZ. The reduction in computational latency is met with the increased memory requirement for the full scene, often with scenes with large unoccupied areas, and mip maps of this scene.

3.2 Sparse Textures

A great deal of previous research into volumetric visualizations has focused on the efficient use of memory as only recently have consumer GPUs started getting increasing amounts of memory, storing entire volumes of data on the GPU was unfeasible To achieve that, A common approach in the previous works is the use of Octrees or N-Trees as Indirection Textures, which point to Node pools. This way only Bricks that contain isosurface need to be stored.

3.2.1 Octree approaches

In *Gigavoxels* (Crassin, C et al., 2008) Octree Subdivides space into Octants recursively until a desired resolution is reached, only octants that contain isosurfaces are subdivided. At the termination depth residing leaf node that points to a position within the node pool that is populated by nonempty bricks. A compact Octree also omits child nodes in cases where there is no need to subdivide space finer as the octant at a given

scale already contains only one child. Brick sizes in Gigavoxels are 16^3 and 32^3.

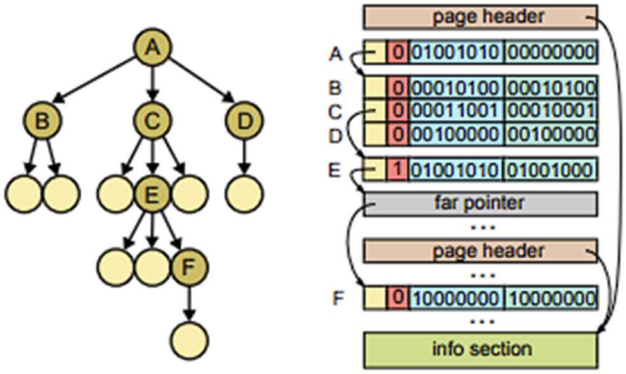

Figure 6 Sparce Efficient Voxxel Octree structure of a compact Tree (Laine S. et.al, 2010)

Octree traversal is performed using a stackless kd-restart algorithm meaning for each step the traversal of the indirection texture begins at the root, this method is due to a limitation of GPUs available at the time. *Gigavoxel* is notable in that it also provides an intricate out-of-core rendering mechanism, where only the data that is being rendered resides on the GPU. During render, GPU informs the CPU what blocks are missing. This approach was necessary as it was not feasible to store the entire volume on the GPU as the dedicated memory capacity of the GPUs in 2009 was on average limited to 2GB. This structure is also not suitable for online reconstruction as the out-of-core mechanism mirrors the octree structure on the CPU as well as GPU. The approach is notable though as it still could be used for Voxel maps far larger than even modern GPUs.

Space Efficient Voxel Octrees (Laine, S et. al. 2010) approach is different in that it puts greater priority onto the Octree for space skipping subdividing space down to individual voxels the approach differs in that it does not utilize raymarching but instead stores the contours, Great focus is placed on the compactness of the structure, so the entirety of the

scene could reside on the GPU removing the need for object updates. The Node Structure of the Octree is of importance here research question at hand.

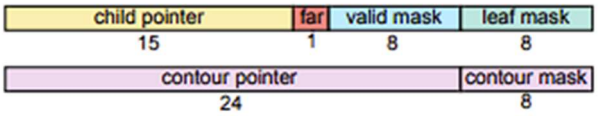

Figure 7 Node Structure (Laine S et.al, 2010)

All nodes are of the size of 64 bits and encode the position of the first child node and a description of what kind of child that node is, valid mask specifies both child's position and child count as the position of the child is calculated using a pointer and incrementing it based of the bits in the valid mask.
Leaf nodes do not require descriptors of their own as they are encoded in the parent.

This approach highlights how making the acceleration structure more compact also makes it faster to traverse. Because of the relationships between all the nodes, it is also possible to rearrange them on the fly, compact nodes make that process easier as we are less constrained by the bandwidth. The papers demonstrate that the performance for the construction is not real-time and is not performed on the GPU but the output structure represents a dataset way larger than the structure intended for this research. It is also important to note that the hardware has advanced since and today the performance might differ from the publishing date of the paper.

3.2.2 Non-Octree approach

Sparse Leap (M. Hadwiger et. al) Presents an alternative to Octree Space skipping with their novel Occupancy geometry. This has been developed with fine-detail scenes in mind as Octree's space-skipping approaches introduce a lot of space fragmentation which necessitates lots of lookups and smaller steps in addition to putting all of the space-skipping work on the ray casting shader. Their Solution is to utilize the

rasterization stage to perform space skipping. This is achieved by constructing per-pixel linked lists composed of records of positions where the ray enters occupied space i.e. bricks, and where it exits them and merging records that share the same occupancy class. This is achieved by rendering nested polygon boxes in the same way that would be performed for the Order Independent Transparency and utilizing the same API capabilities that allow OIT.
Resulting linked lists are then used for ray marching in a single pass avoiding the need to do any space skipping. This approach allows for fine control over the composition of the structures as the Bricks could be included or omitted from the rasterization pass. Unfortunately, although it allows for restructuring it makes

3.3 Alternative Approach to Deformations

One way of deforming the Scene is by changing the implicit defined in the texture. As it is previously described this can be done relatively efficiently with dense structures. With octrees some challenges are present dues to the compactness of data and on GPU inserting nodes is difficult but determining which nodes to merge to be conservative with edits is simple.

Alternatively, Seyb, D. et. Al (2019). Describes a method for nonlinear Sphere tracing. Instead of Editing the Implicit solid itself via changes to the underlying structure Authors propose instead to deform the space that the solid exists in thus achieving more familiar ways of animating the solid, those akin to those using polygonal meshes like stretching, squeezing, etc. The Algorithm comprises the following steps: Constructing the Solid, Creating a hull around it using marching cubes, Deforming the hull using chosen forward deformation, Rasterization then raymarching the surface within. Implicit surface exists within undeformed space so the ray has to be deformed to account for the deformation. For this, they describe a method of sphere tracing where the trace is incremented by a vector transformed by the approximate deformation applied to the undeformed point. The resulting algorithm still performs at

real-time speeds and allows for deformations that do not alter the underlying structure in which the implicit is contained.

Chapter 3 Methodology

3.1 Overview

In this chapter, we present the methodology used to evaluate different SDF (Signed Distance Field) geometry pipelines. We developed an application in C++ using the DirectX 11 API framework at feature level 12, utilizing shader model 5. The application allows for rendering a predetermined number of functionally defined objects on the screen. These objects can be discretized into a dense texture or a sparse texture along with the corresponding octree required for correct rendering.
The pipeline can be configured to output geometry from a functionally defined scene or the discrete volume scene. The implicit surface can also be polygonized using marching cubes and rendered through the rasterization pipeline.
The main objective of the application was to evaluate the construction and rendering of sparse discretization approaches in terms of computational latency compared to the dense approach. Additionally, we aimed to compare the visual fidelity of raymarching approaches with polygonal alternatives.
To conduct shape fidelity testing, a polygonal model with a high level of tessellation was necessary. To achieve this, we implemented the marching cubes algorithm in the compute shader to extract the polygonal model from the signed distance field using the same Scene configuration.
To evaluate the computational latency of the construction process, we describe the implementation of the octree construction stages. This includes selecting grid cells to be built and using their respective addresses to construct the octree. We discuss the GPU-based, parallel, octree construction approach employed to achieve this.
The testing section discusses the conditions of each test designed to reveal results related to the computation latency of the octree construction as well as the latency of raymarching. This section will also

outline fidelity testing. We outline the variable parameters used in the tests and state the expected outcomes.

3.2 Functional Raymarching

To establish the ground truth for our discretized geometry, our first step was to implement a renderer that solely relies on functional raymarching. This renderer is capable of producing scale-independent and pixel-perfect representations of the SDF (Signed Distance Field) plutonic.
Our objective here was to recreate the functionality of Shader Toy, the renderer by Inigo Q (2023). A WebGL renderer to allow for testing pixel shaders often used to display SDF functions and other implicit on the screen through the technique of raymarching. (is this needed?)

3.2.1 Implementation

The distance to the functionally defined scene is evaluated using a compute shader dispatch of one thread per pixel. Each thread performs Raymarching and terminates when the evaluated distance is below the specified epsilon condition. The Output to the Screen is normal that is calculated from the scene SDF gradient at the terminal position. The normal Is calculated by taking a central difference per each axis. The bias value for the normal is taken from the shader input constant buffer.

The results of this renderer have been used to test scenes for the next iterations of the pipeline. Because of its pixel-perfect nature independent of the resolution, this is used as a ground truth for Laplacian comparisons between the baked texture and marching cubes.

3.3 Marching cubes

Marching cubes were implemented following Paul Bourke's (1994) implementation adapted from the CPU to DX11 SM5 Compute Shader with fixed grid resolution but modifiable grid Stride allowing for varying the resolution of an output model, it is also possible to change the isolevel offset that model would represent.

The edge table and Triangle table are passed to the compute shader as 1D and 2D buffers respectively formatted as FLOAT32.
Compute shader evaluates the cell and if it is intersected by an isosurface the shader outputs a triangle configuration for that cell to the append buffer.
The triangle structure contains 3 vertices.
Each vertex contains position, texture coordinate, and normal.
True normal is calculated by taking the cross product of 2 edges of a

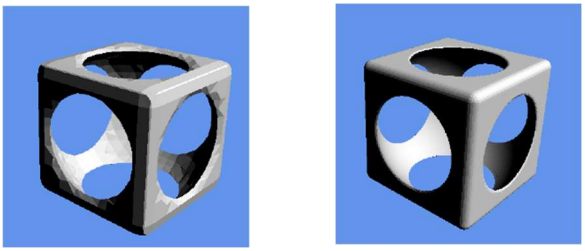

Figure 8 False normal at 0.01 and 0.05 strides respectively

triangle.

Alternatively, a false normal could be calculated from the source SDF

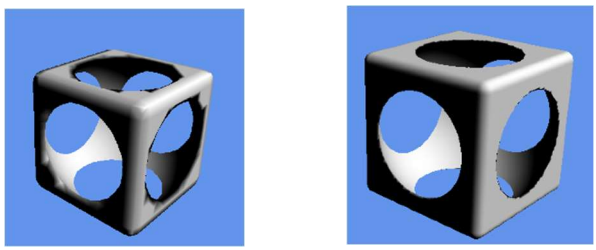

Figure 9 True normal at 0.01 and 0.05 strides respectively

For fidelity testing, a true normal has been used as it produced more consistent results at lower resolutions of the grid.
Marching cubes algorithms performance will not be evaluated directly as is not an optimal implementation and several implementations exist that produce better performance. It Should also be noted that the implementation presented only creates a triangle list which would unfavorably impact the memory statistics of the algorithm as there is no index buffer output and vertices are not reused. To note, Texture coordinate is stored but not used.

Marching cubes were implemented to give polygonal alternatives for fidelity testing.
It was also used in testing whether SDF was correctly written within the 3D texture, next section, and later individual sections of that texture, I.e., octree bricks section 3. in 5.4

3.4 SDF Discretisation

To achieve the goal of creating individual bricks for sparse volume, initially, the entire texture volume has been evaluated.
The volume of dimensions 256x256x256 is written in a dispatch. Each texel stores an SDF value sampled from the functional scene.
Texture is stored as 8bit SNORM which is chosen based of Hasson-Soderlund, Aaltonen, and Evans, as SDF does not benefit from higher precision, in addition when sampling the Scene brush floating point precision is highest in the range [-1;1]

In the Application, Texture Stride or the encoding of the 8-bit range is controllable from 1 to 32, with the value of 1 encoding full precision 1/256 a the d value of 32 encodings 1/8.

To test whether the Texture is written correctly and produces the expected shapes the marching cubes were used with positions sampled in normalized uv space instead. The resulting polygon model should not differ from the model sampled from the functional representation.

To Raymarch the dense texture, the scene sampling stage was replaced with Sample Level using Linear sampler state to reconstruct gradients between cell values using hardware accelerated texture sampling.
Texture Stride is also passed in to correctly decode the step size.

To prevent raymarching through clamped values of the texture the trace begins with an evaluation of the intersection between a ray and a bounding box. If the intersection occurs the ray origin is moved along the ray to the position of the intersection tmin and then translated to the local normalized coordinates of the bounding box to allow for correct sampling to occur. Raymarching terminates when the tmax value is reached.

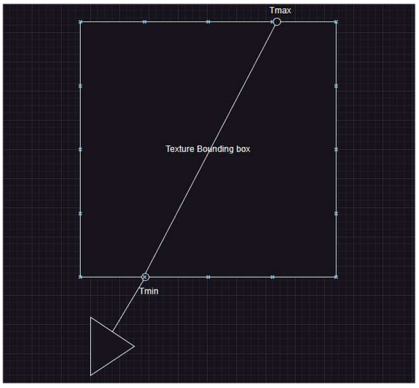

Figure 10 Dense texture Raymarching should only occur in the Bounding Box

3.4.1 Dense Volume Construction Performance Issues in the Artifact

During testing, the Construction of the dense volume is disregarded as its implementation is suboptimal. Dispatch of 256^3 groups of 1 thread although build the texture performs roughly 130% worse than a Dispatch of 8^3 groups of 32^2 threads. This implementation will be further discussed in section 3.5 as part of octree construction.
Instead, the Octree Brick construction stage at the maximum density of 512 bricks will be taken in the discussion.

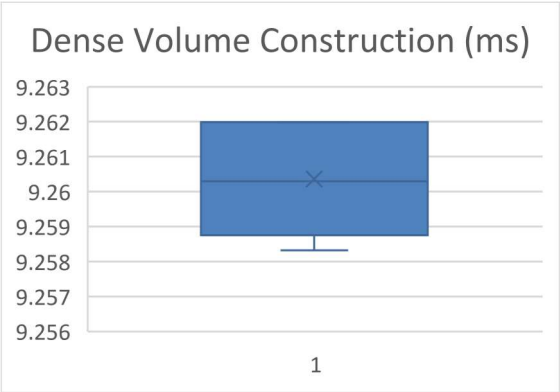

Figure 11 Dense Volume Construction Unoptimised Shader

3.5 Octree

To create an Octree structure that will be used to render the sparse volume that is otherwise impossible to render due to the lack of information to inform raymarching correctly we propose a structure that is constructed in 3 Stages on the GPU with the node structure inspired by work od S Laine et al. 2009 on Sparce efficient Voxel octrees and Crassin et al. 2008. The structure is implemented for the volume texture of the resolution 256^3 where each block stores 32^3 values giving us the maximum resolution of 512 bricks. Careful consideration has been taken to make sure that the algorithm maintains high occupancy of the GPU at each stage and accesses to the global memory are kept to a minimum.

3.5.1 Identifying relevant Bricks

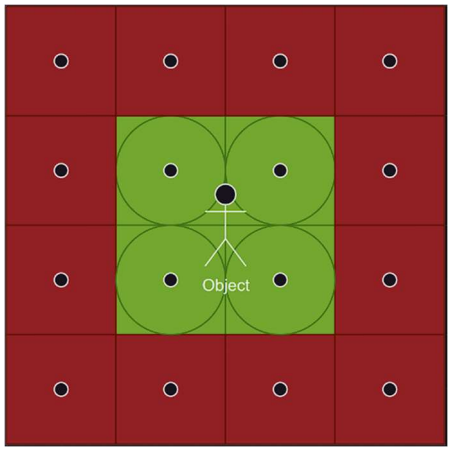

Figure 12 Valid Brick evaluation expected outcome exemplified in 2d for clarity. Circle indicates the

Figure 11 illustrates the approach that was taken when considering which bricks to keep during construction. Following Aaltonen's approach for chunks to be built, it needs to contain the sdf within the desired stride. That is done by taking the centre point which is calculated using formula in the Figure 12.

$$((Dispatch\ threadID\ *\ BRCK\_SIZE)\ +\ 0.5\ *\ BRICK\ \ \_SIZE * Volume\ Resolution\ -\ 0.5) * TextureStride$$

Figure 13 Center point calculation

if the Distance to the sdf at this point is below 1. The chunks index is stored in the Group Shared Memory

3.5.2 Morton Codes

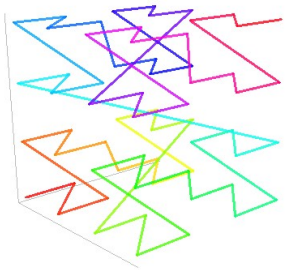

Figure 14 Z-Order Curve (Wikipedia)

To encode the 3D position as a 1D index and preserve the spatial coherence of the nodes which will be used during octree construction we utilize the Z-order curve, so the index is encoded as a Morton number which is done by interleaving the bits of the XYZ 3D address. Implementation to compute Morton code follows the “Magic Number” approach described by Terro Karras (2012)

Our octree construction approach realizes that the nodes are in the ascending order of the Morton code. So before writing them to the global memory, i.e., the structured buffer allocated to store our tree structure, we sort the codes using enumeration sort which in this case has O(n) complexity as we are working with a thread count larger than the number of elements in the list. Then the Sorted list is written to the global memory and the total node count is written to the argument buffer to inform the brick construction shader how many groups are needed to be dispatched.

3.5.3 Texture bricks

Texture Bricks are constructed, at position decoded from the Morton codes written in the previous dispatch within the texture. Each group of 32x32 groups writes 32^3 cells within the texture, this brick size was used following the work of Crassin et al. Each thread performs 32 steps to fill out a brick. This was done as it gives the full occupancy having 1024 threads in a block and aligns exactly with the 32 threads in a warp.

3.5.4 Octree Construction

Nodes encode information following S Laine implementation, except they do not compress the masks and child pointers into single 32 bits. This was done to simplify validation.

To construct the sparse octree the Morton codes are copied to the GSM. The construction happens Bottom-Up starting with the highest resolution of the grid and reducing till we reach the root node.

Here we exploit the Z-order curve to determine the common local group of the nodes by looking at Morton codes in 6-bit intervals because we are working in 3 dimensions, the highest 3 bits indicate the parent group whereas the lowest set of 3 bits indicates the position within that parent

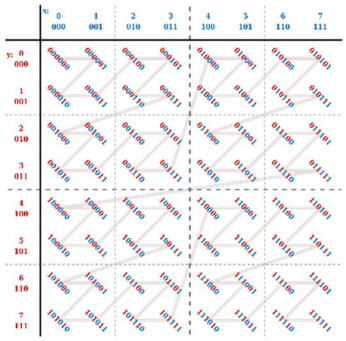

Figure 15 2D Z-order curve with bits interleaving to illustrating the grouping

group.

Working Node per thread, each node considers its parent group and the parent group of the node that is located before it in the GSM

If the parent is different Node Thread writes the position of the node to a separate array in the GSM at the parent node location as shown in the figure below. Then the Valid mask is written to the parent node by performing interlocked bitwise OR operation of the bit shifted by the number describing position within a group.

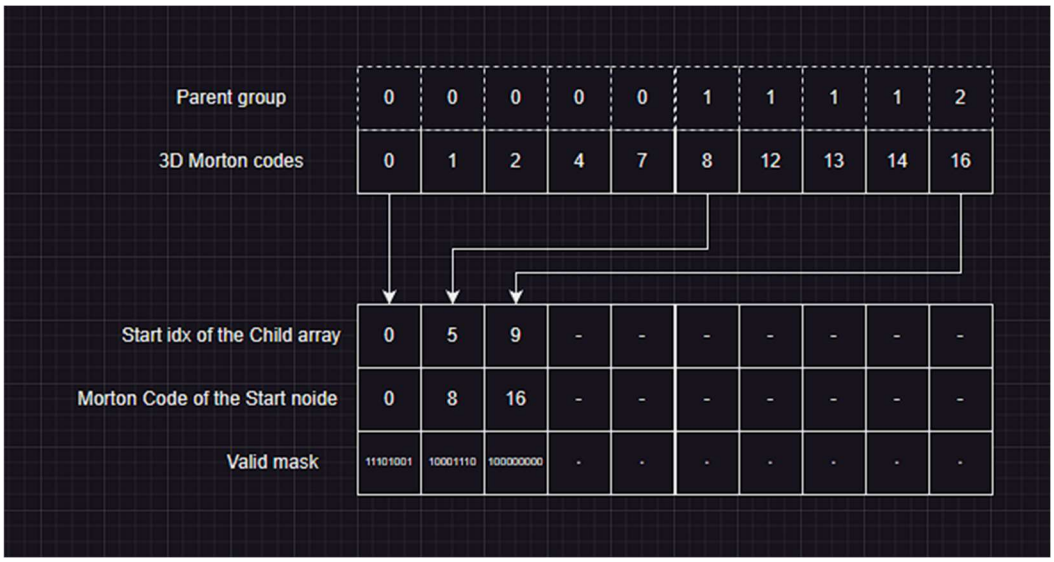

Figure 16 Parent-level construction using the Morton codes

This approach produces a similar Child-Parent relationship between nodes as described by S laine et al.

To compress the nodes in case some parents have no children which is indicated by the valid mask, the Parallel Prefix Scan algorithm is used. compression is meant to maximize the chance of cache hits.
The compressed Layer is written to the Global memory. In the following layout. The layout was picked as it would make it easier to update the relevant nodes in the hierarchy if insertions were to occur later on in the development

Figure 17 Octree Level node layout in memory

Construction of the next layer begins by copying the current layer back into the GSM. As we only construct a total of 3 layers the process is repeated twice. As the tree is not compact, ie it stores a fixed number of layers instead of omitting layers in cases where there's only one leaf node per parent, a leaf mask is written but remains unused.

Verification of the node relationship has been done manually by using renderdoc and inspecting the output buffer. The result of the node of

interest should follow the convention index = Level Offset + Node Offset + Child Offset.

The level offset indicates the sum of the size of the levels prior, the Node offset is the start position of the node encoded in the parent node, and the child offset is the number of bits that are set in the child mask in the position before the local position of interest.

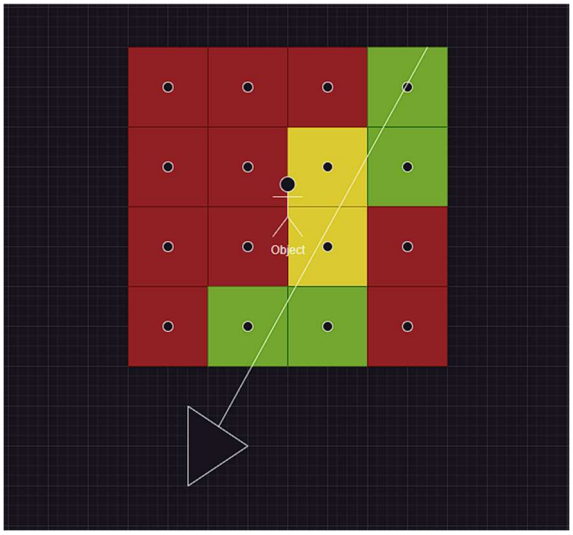

Figure 18 Octree Raytracing

3.5.6 Octree Raytracing

To traverse the tree and identify all the leaf nodes that need to be ray marched we follow Crassin's approach. Meaning we perform a lookup into the tree each time we perform a step in the structure. Similarly to the dense structure we first perform the

3.5.6.1DDA

A Branchless Digital Differential analysis algorithm is used following Inigo. Q implementation to visit every relevant node seen in Figure 17 nodes in yellow and green. DDA is performed on the grid that corresponds to the lowest level of the octree. Coordinates that are used to look up in the octree are the central position of the cells as indicated in Figure 17. Border values cause visual artifacts due to floating point precision floor() and ceil() errors.

Validation was done using the structure that contained only one node comparing the expected result to the output. If the ray hit that node brick was colored red.

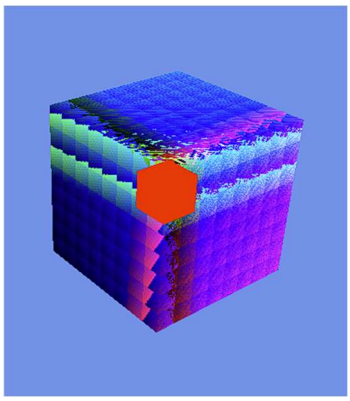

Figure 19 Octree with only 8 nodes in the center populated, DDA returns nodes within the cube at the expected position

3.5.6.2 Brick Raymarching

When the node is identified as a valid node the raymarching can occur. Raymarching of the node is the same as the raymarching of a dense texture described in Chapter 3. Except the beginning position is acquired from the dda algorithms distance accumulation. The minimal accumulated distance along the ray serves as the start and end position of the next min accumulated distance.

Raymarching octree in our implementation should produce the same results as the dense texture raymarching as the nodes do not change the position within the texture. The main focus of the research is speeding up the evaluation and re-evaluation of the volume.

3.6 Testing

The goal of the experiment was to evaluate how sparse voxel octree can be applied to accelerate the discretization of a 3 Dimensional object composed of SDF while maintaining fast construction and rendering time for the discrete representation. While maintaining the Discretisation process have also been evaluated for image fidelity compared to the polygonal alternative.

To facilitate this experiment 3.5 SDF render pipeline configurations have been created Functional Raymarching, Polygonization and rasterization, and Discrete Raymarching. Discrete raymarching has been split into a sparse and dense approaches.

Computational latency testing has been performed for rendering and construction of Dense volume although the dense volume against sparse volume octree approach.

Fidelity testing has been performed to compare polygonal representation again the discrete ray marched.

3.6.1Testing Scenarios

Testing has been performed by reconfiguring the source scene to facilitate the most insightful results for each of the tests, scenes vary from dense to sparse, porous to solid, containing spans of flat surfaces or lots of curved surfaces, this way uncovering the downsides of on rendering approach against the other and presenting a fair approximation of real-world application. Each testing scenario will outline what scene has been used.

3.6.2.1 Construction performance

Construction has been evaluated for the Dense Volume and Sparce Voxel octree. Evaluations have been performed over 4 runs of construction with varying input sizes. As our implementations' octrees' maximum resolution is fixed, we vary the input by configuring the scene to occupy the set number of bricks. To minimize the impact of the selected brush used for the testing a sphere that was centered for each brick and then repeated using domain repetition taken from Inigo Quilez

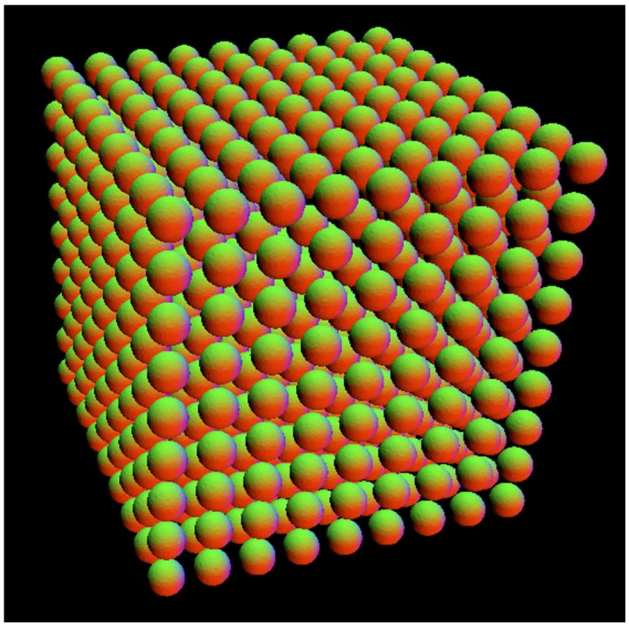

Figure 20 Fully occupied scene with 512 spheres

Dense Volumes have been evaluated only once as the full texture is rewritten.

Sparse Volume was constructed by varying the input size for 4 runs and the results have been averaged between runs for the respective input sizes. Each stage of the octree has been evaluated.

3.6.2.2 Render Performance

Render performance has been evaluated in various scenes by capturing each scene through the span of 250 frames while moving the camera around it. Scenes vary from dense to sparse to show the advantages and disadvantages of the additional step complexity introduced by octree traversal.

These Results have been paired with the Heatmap of the results for each scene to better facilitate the discussion around the Step Complexity vs Step Counts.
To not affect the other results this step has been taken out and is performed via an identical shader that specifically renders the heat maps. All raytracing shaders are clamped to perform a maximum of 256 steps. This way Heat map colors encode values 0 to 256 to colors blue ->green-> yellow-> red.

Fidelity testing

Fidelity testing has been performed for a single scene object, with matched scale at a fixed camera angle. The teapot scene Created by Sebastien Durand (2014) has been used as it provides a variety of test cases to be mindful of and renders lots of smooth surfaces as well as sharper edges.
To quantify the results a 5x5 Laplacian has been applied to the output images for ground truth - functionally evaluated teapot, the output of raster pipeline with mesh produced using marching cubes, and the discrete raymarching pipeline.
This allows us to evaluate and quantify the difference between the images by taking the difference for each pixel of the Laplacians between ground truth and the respective test cases. Images were combined using Adobe Photoshop.

Chapter 4 Results

This chapter will present the result of the testing that has been er

All Performance evaluations have been performed on a PC running Windows 11 equipped with 32GB RAM, AMD Ryzen 5800X, and AMD Radeon RX 6800XT with 16GB of GDDR6.

Results have been captured using GPUPerf API. Results for fidelity evaluation have been captured using Renderdoc

4.1 Construction Performance

Octree Construction

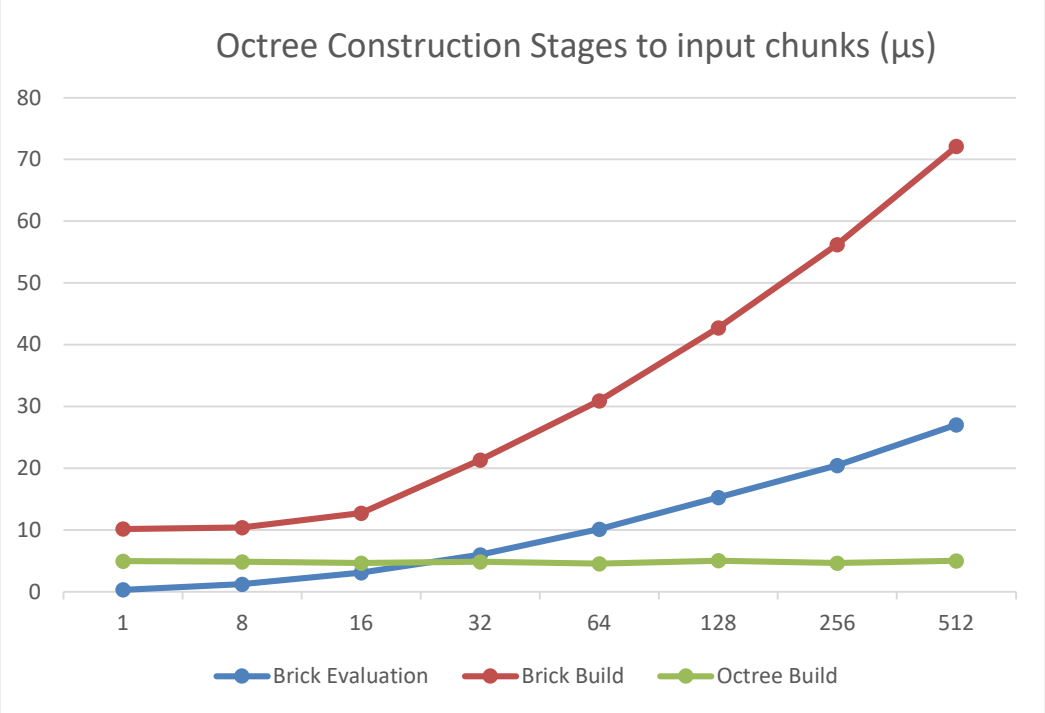

Figure 21 Octree Construction stages time complexity concerning the input brick count as a line graph to demonstrate

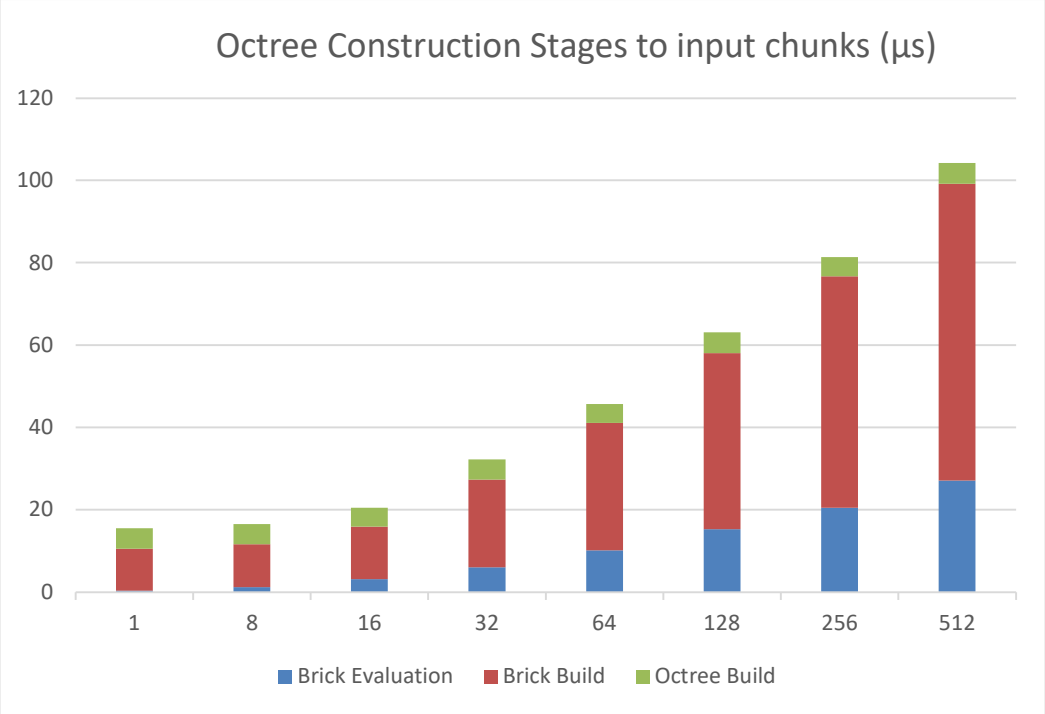

Figure 22 Octree total time complexity about input size as a stacked bar to better demonstrate the potential savings.

4.2 Traversal Performance for Raymarching

For reference the most complex scene object that can be rendered in the artifact is a teapot, to render it the average time taken is 2405.88 microseconds on average. (2.405 ms).

4.2.1 Octree Traversal

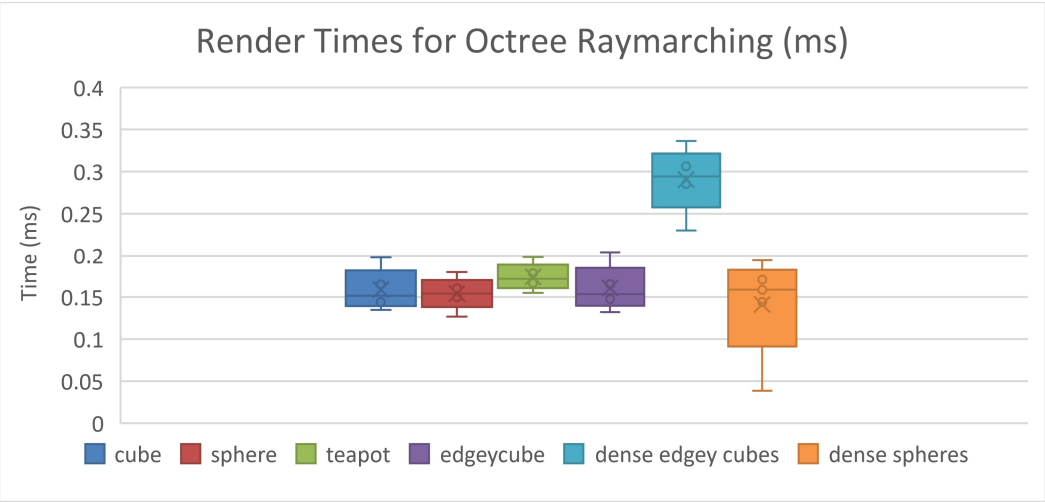

Figure 23 Render times for Octree

4.2.2 Dense Texture Traversal

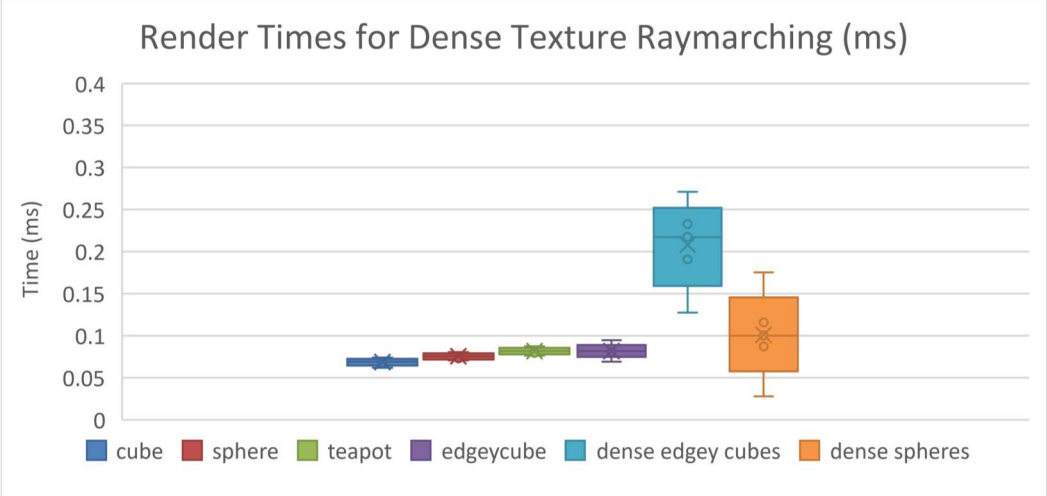

Figure 24 Render times for Dense textures

4.2.3 Heatmap Results in Step Count

Step count was captured for each scene (brick, teapot, edgy cube, dense spheres, and dense edgy cubes).

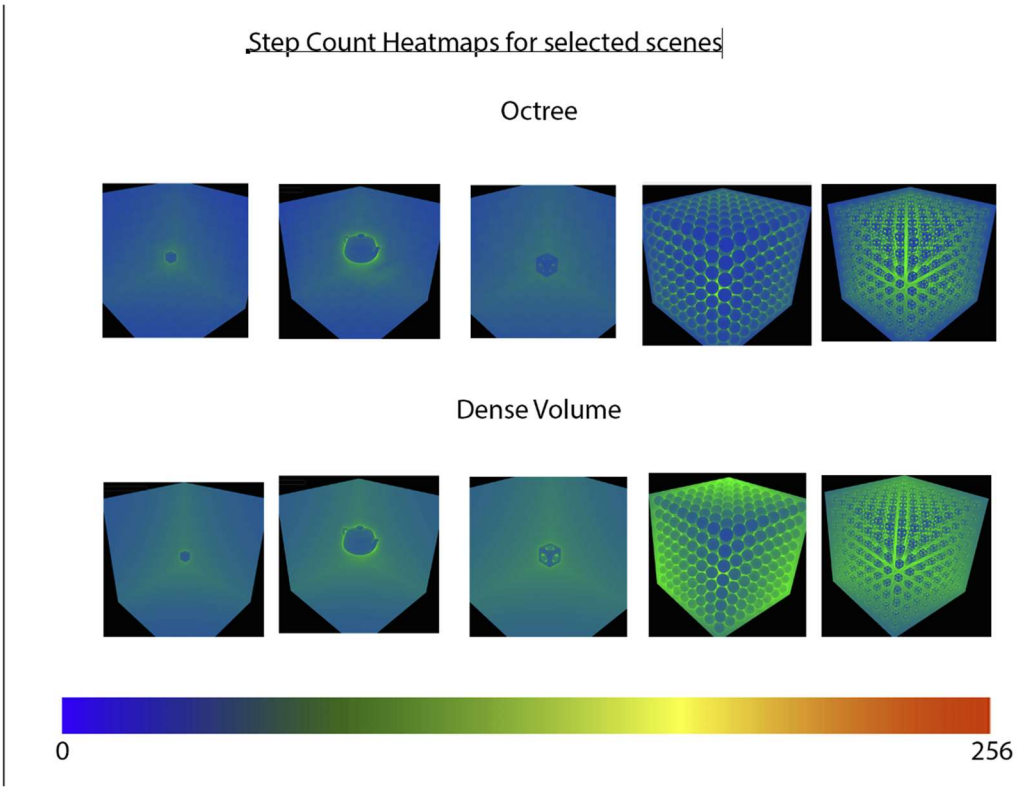

Figure 25 Step count heatmap

4.3 Fidelity Comparisons

Ground Truth Functional Ray marched Teapot

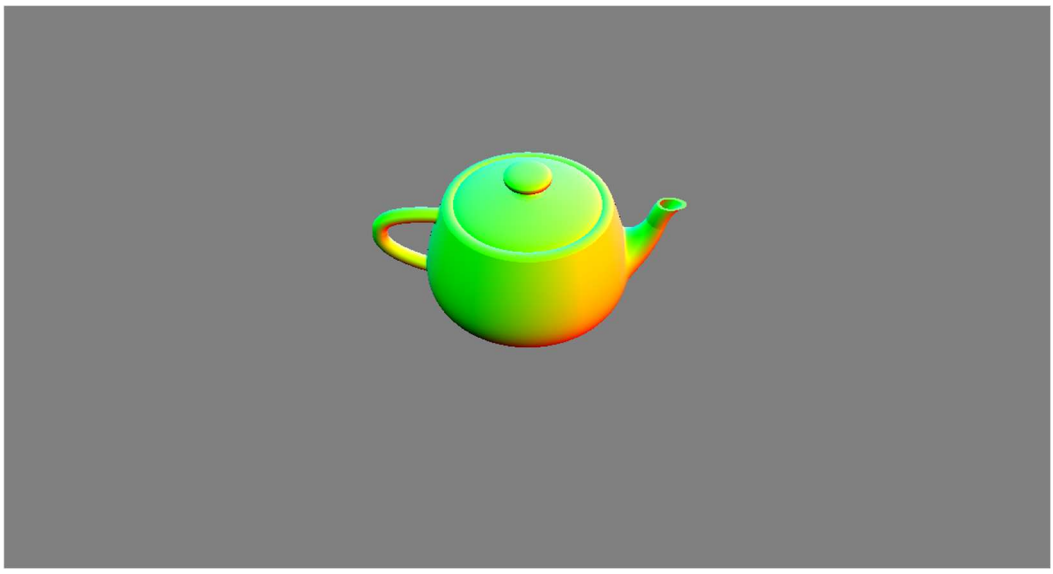

Figure 26 Teapot rendered using functional representation

Polygon Teapot

Using teapot at grid stride 0.05 with grid size 100x100x100

Polygon count = 68440 with 205 320 vertices

Render Time 25.20 µs

Construction Time 17.137 ms

Numbers are provided for reference.

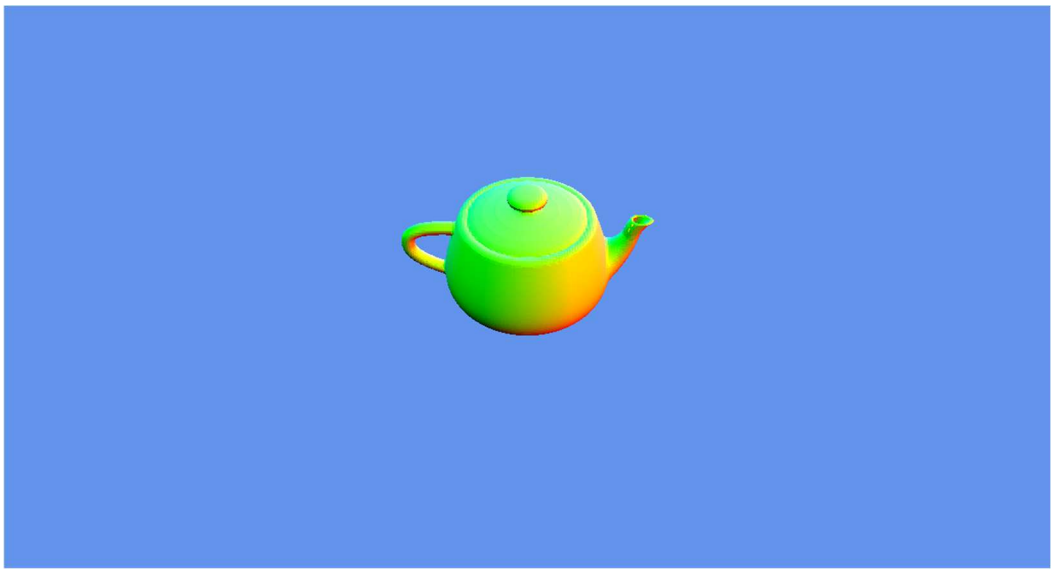

Figure 27 Teapot, Polygonised

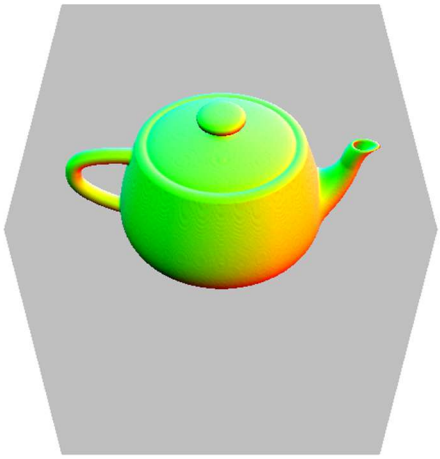

Figure 28 Teapot Discretised and then Raymarched

Discrete Ray marched teapot

Figure 29 Ground Truth Laplacian difference with blank white canvas

Figure 30 Figure Polygonal and Raymarched Texture difference between laplacians to functionally raymarched scenes respectively.

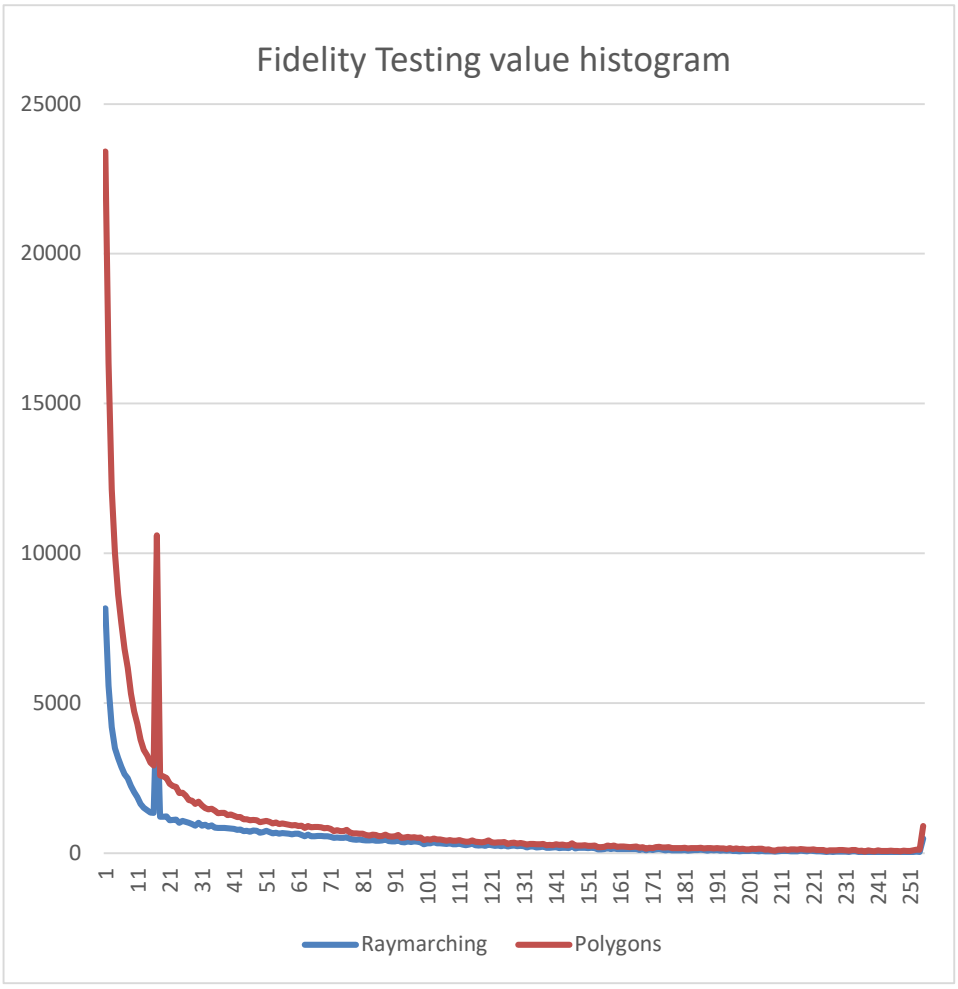

Figure 31 Figure 5 is a Histogram of the red channel

To give us a more valuable insight into the distribution of error pixel values of 0 were removed from the graphs, these pixel counts are as follows.

1769333 for raymarching and 1768520 pixels for polygonization.

Chapter 5 Discussion

This study has explored how Signed Distance Fields could be discretized and rendered in real-time to allow for real-time modifiable objects. In this section, we will go over the results displayed in Section 4 to answer the research question: How do the sparse discretization approaches of signed distance fields impact the computational latency and the visual quality of the geometry when constructing and rendering arbitrarily editable environments?

It was hypothesized in section 1.4 that the fragmentation of space through the use of an octree would allow us to discretize the SDF faster while maintaining real-time render times. This relies on the assumption that the scene sampled contains a lot of space that would be omitted from the construction. In cases where the scene is dense with detail, it was expected that the rendering performance would be most impacted as the raymarching is burdened by the octree lookups which are unnecessary when all the neighboring bricks are populated, However, the evaluation of the dense scene render times and the step heatmap have revealed this to be somewhat inconclusive might need further examination. The following sections will discuss these results in detail.
Fidelity evaluation is also performed by comparing the outputs of the histograms.

5.1 Construction performance

5.1.1 Dense Approach

As mentioned in section 3.4.1 Dense volume construction by itself results have been dismissed to the poorly optimized shader. Taking as a baseline for dense volume construction the results of the construction of 512 individual chunks were used in the further discussion about our sparse approach. It takes approximately 72 microseconds. Technically on our hardware, this remains real-time framerate but if the volume were to be expanded to the nearest real implementation, Aaltonen’s Claybook, a volume that is of resolution 1024x1024x512, or 32 times bigger our data

can be extrapolated and a volume of that size would approximately take 2.3 ms which although still is under 16ms mark is missing any lighting or texturing calculations and also is not evaluating a single functional scene, for which there would be more as it was discussed Literature review that texturing an implicit volume is more challenging than a polygonal mesh.

5.1.2 Sparce Approach

Figures 22 and 23 demonstrate When only the relative chunks are constructed which demonstrates great benefits to be gained over the dense approach as it allows us to reduce the amount of workload to only include the necessary areas of the object brush. In line with the expectations the octree construction remains constant, and brick evaluation grows with the input size due to the sorting algorithm implemented. And brick construction grows with the input size. It is hard to approximate the real memory savings due to the potential need for brick adjacency information to be required but theoretical savings can be quite significant. As octree size, when Nodes are compressed further removing information that was used for the convenience of debugging would only take up the size of a few bricks.

Extrapolating the information gathered we could estimate the construction time for the octree that would support a volume as large as one in Claybook for real-time revaluation but would demand an alternative approach to current memory management which potentially would introduce further latency due to the additional layers in the memory structure, making it less of a viable strategy. On the other hand, Our structure can be used for smaller objects albeit sacrificing some degree of detail.

5.2 Render performance

At first glance, the results show that the Octree look-up greatly impedes our render times. This is in line with the expected outcome as octree rendering ultimately just adds extra complexity compared to the dense volume raymarching. This is in addition to making the render times less

consistent per object as render times produce a larger broader range of values.

On the Other hand, it is important to note that we have successfully reduced the step count with some scenes as can be seen in Figure 26. Although that result is suboptimal and could be further improved with a different space-skipping approach, rather than utilizing the DDA and lots of descents. This is in line with the literature review as Crassin's et al. (2007) approach, which is what our descent implementation is based is dated and perhaps less fit for the capabilities of modern hardware.

5.2.1 Step complexity vs Step count.

Upon closer inspection of Figure 26, it is relatively safe to conclude that on average both methods terminate at around 128 - 134 steps at a maximum which means that no rays produce extra steps beyond what is expected. This further suggests that improvement is required in the octree descent to improve step performance.
Interestingly, with dense scenes, the additional partitioning of the scene has some benefit on the step count when rendering outside of the volume.
Testing results per scene also outline that our approach loses to the Grid Sphere tracing described by Aaltonen (2018) as we take far more steps to converge the scene. The mind map structure cuts down this count in half compared to our approach. The step latency of our approach is also higher as it does not benefit from the hardware-accelerated texture sampling.

5.3 Fidelity comparison

Figures 31 and 32 demonstrate that generally, the polygonal output produces results that are closer to the ground truth due to the larger count of black pixels, but as expected through visual inspection it struggles more with fine details encoded by the volume, like the rim of the lid of the teapot for example. This highlights requirement for polygonal meshed to be more tessellated to better represent those details which is

in line with A.Evans.(2018) However, it is hard to ignore that in this particular comparison the ray marched version loses, primarily due to the artifacts of the normal calculation, this is also indicated by the distribution of error. Polygonal model suffers at the specific parts of the object where's raymarching is uniformly wonky.

5.4 Test Limitations

While the findings reflect the hypothesised outcome there are more opportunities to make the results more definitive.

Performance testing could be improved by having a closer look at what types of steps are having a larger effect on the rendering performance, a more exact measure of the cost of octree descents against raymarching. Lack of control over the resolution of the bricks and the breadth and depth of the octree make it hard to confidently estimate what produces a bigger impact on the render performance.

Due to the omission of the fast and reliable marching cubes implementation is hard to truly evaluate fidelity and render the performance of a consistent polygonal alternative.

Fidelity testing is performed on a single object which ultimately is not very representative of the possible scenarios and the overall behaviour of the marching cubes with the other scenes. Chosen Fidelity evaluation is also affected by the fact that normal calculation severely affects it's results and just by simply changing the normal bias value a smoother result can be produced but that can be inconsistent between objects with different brushes.

Testing has been mostly performed from outside of the volume with the camera as an observer which depending on the use case would produce varying results.

More testing is required that would investigate the specific performance of the octree for render performance as it was constructed for speed of construction and ease of editing which poorly impacts the render performance. One such test could be looking at the L0 cache hit rate.

In addition, it might be worth noting that testing on different hardware might uncover different inconsistencies in the implementation of shaders themselves.

5.5 Overview

The collected results have definitively helped to highlight the importance of space partitioning when constructing and rendering discrete geometry composed of signed distance fields. On the other hand it has demonstrated that the sparce approach to rendering should take great focus on whether to allows for easy editability or fast reconstruction, as both could not be easily maintained at the same time, unless the objects are not very intricate. To maintain desired fidelity a consideration have to be made on the chink size and resolution which makes it more difficult to keep a good balance between reliance on raymarching and octree raytracing, highlighting the need for optimisation I the area that affects the performance more.

The performance metric have showed that our approach is not particularly suitable to be rendered as individual objects due to the slow render performance but also not quite suitable for large dataset rendering due to the very limited maximum resolution.

Chapter 6 Conclusion and Future Work

In conclusion, the method that we have implemented for lookup shows inefficiencies that make it hard to recommend but it demonstrates pitfalls of rendering sparce approaches to the sdf raytracing. Resorting to building and storing only the bricks that contain a surface necessitates a use of an indirection structure like an octree, that although can be extremely fast even allowing entire octree to be rebuilt every frame for multiple objects can negatively affect the render performance so optimisations for space skipping using octree need to be applied carefully as they would impact the how the object can be modified, perhaps removing the ability to add bricks. For some game applications this might be fine as a dent in a surface doesn't require a node restructuring, but in the pursuit of bigger edits should inform the data structure design and layout. Same approach goes to fidelity. Lower precision might produce desirable results with no need to store chunks bigger than 8x8x8.

Through our sparce octree implementation we have been able to demonstrate that the careful consideration for the particular piece of geometry have to be made when trying to benefit from the malleable opportunities the sdf representation provides otherwise this will limit the expansiveness of use of such objects due to large GPU memory requirement or excessively expensive render times or make them rigid and not as flexible as construction on the fly is not achievable.

Future Work

From this base implementation it would be valuable to explore the alternative space skipping approaches like Sparce Leap but it would require understanding limitations of order independent transparency and how such approach might be incompatible with games context. Following improvement to space skipping it would be valuable to explore alternative approaches to raymarching such as Analytical voxel intersections especially in at lower resolutions to see whether those methods could

compete with raymarching and a small scale in terms of expressiveness and computational latency.

List of References

Aaltonen, S., 2018. *GPU-based clay simulation and ray-tracing tech in Claybook (GDC 2018)*. [online] https://www.secondorder.com/publications. Available at: <https://www.dropbox.com/s/s9tzmyj0wqkymmz/Claybook_Simulation_Raytracing_GDC18.pptx?dl=0> [Accessed 18 October 2022].

Braithwaite, W. and Museth, K., 2022. *Accelerating OpenVDB on GPUs with NanoVDB | NVIDIA Technical Blog*. [online] NVIDIA Technical Blog. Available at: <https://developer.nvidia.com/blog/accelerating-openvdb-on-gpus-with-nanovdb/> [Accessed 18 October 2022].

En.wikipedia.org. 2022. *Constructive solid geometry - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Constructive_solid_geometry#:~:text=%20Constructive%20solid%20geometry%20%28%20CSG%3B%20formerly%20called,geometry%29%20is%20a%20technique%20used%20in%20solid%20modeling.> [Accessed 18 October 2022].

Evans, A., 2015. *Learning from failure: A Survey of Promising, Unconventional and Mostly Abandoned Renderers for 'Dreams PS4', a Geometrically Dense, Painterly UGC Game*. [online] Media.lolrus.mediamolecule.com. Available at: <http://media.lolrus.mediamolecule.com/AlexEvans_SIGGRAPH-2015.pdf> [Accessed 18 October 2022].

S. Laine and T. Karras, "Efficient Sparse Voxel Octrees," [online] *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 8, pp. 1048-1059, Aug. 2011, doi: 10.1109/TVCG.2010.240.

M. Hadwiger, A. K. Al-Awami, J. Beyer, M. Agus, and H. Pfister, *"SparseLeap: Efficient Empty Space Skipping for Large-Scale Volume Rendering*," [online] IEEE Transactions on Visualization and Computer Graphics, vol. 24, no. 1, pp. 974-983, Jan. 2018, doi: 10.1109/TVCG.2017.2744238.

Quilez, I., 2022. *Inigo Quilez*. [online] Iquilezles.org. Available at: <https://iquilezles.org/articles/distfunctions/> [Accessed 18 October 2022].

Reiner, T., Mückl, G. and Dachsbacher, C., 2011. Interactive modeling of implicit surfaces using a direct visualization approach with signed distance functions. *Computers & Graphics*, 35(3), pp.596-603.

Hart, John. (1995). Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces. The Visual Computer. 12. 10.1007/s003710050084.

Hansson-Söderlund, H., Evans, A. and Akenine-Möller, T. (2022). *Ray Tracing of Signed Distance Function Grids*. [online] Jcgt.org. Available at: https://jcgt.org/published/0011/03/06/ [Accessed 25 Nov. 2022].

Crassin, C., Neyret, F., Lefebvre, S. and Eisemann, E. (2009). *GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering*. [online] *Inria Maverick*. ACM Press. Available at: https://maverick.inria.fr/Publications/2009/CNLE09/index.php [Accessed 22 Nov. 2022].

Zink, J. (2013). *A Closer Look At Parallax Occlusion Mapping*. [online] GameDev.net. Available at: https://www.gamedev.net/articles/programming/graphics/a-closer-look-at-parallax-occlusion-mapping-r3262/#:~:text=Parallax%20occlusion%20mapping%20is%20a%20technique%20that%20reduces [Accessed 27 Nov. 2022].

Seyb, D., Jacobson, A., Nowrouzezahrai, D. and Jarosz, W. (2019). Non-linear sphere tracing for rendering deformed signed distance fields. *ACM Transactions on Graphics*, 38(6), pp.1–12. doi:10.1145/3355089.3356502.

J. F. Blinn, "How to solve a cubic equation. Part 1. The shape of the discriminant," in *IEEE Computer Graphics and Applications*, vol. 26, no. 3, pp. 84-93, May-June 2006, doi: 10.1109/MCG.2006.60.

Lorensen, WE and Cline, HE (1987), Marching cubes: A high-resolution 3D surface construction algorithm, ACM SIGGRAPH Computer Graphics, Association for Computing Machinery (ACM), pp. 163–169.

Bálint, C. and Valasek, G. (2018). Accelerating Sphere Tracing. *EG 2018 - Short Papers*. doi:10.2312/egs.20181037.

Green, C (2007), Improved alpha-tested magnification for vector textures and special effects, ACM SIGGRAPH 2007 courses, ACM.

Bourke, P. (1994) *Polygonising a scalar field*, *Polygonising a scalar field (Marching Cubes)*. Available at: http://www.paulbourke.net/geometry/polygonise/ (Accessed: 16 May 2023).

Microsoft (2009) DirectX 11 API Specification, Microsoft, Available at DirectX-Specs | Engineering specs for DirectX features. (microsoft.github.io)

Valve Software (2023) Steam Hardware Survey, Valve, Available at **Steam Hardware & Software Survey (steampowered.com)**

Terro Karras (2012) Thinking Parallel, Part III: Tree Construction on the GPU, Nvidia, Available at **Thinking Parallel, Part III: Tree Construction on the GPU | NVIDIA Technical Blog**