Real-Time Rendering and Construction of Signed Distance Fields

Jamie Buttenshaw

BSc (Hons) Computer Games Technology, 2024

School of Design and Informatics
Abertay University

Table of Contents

Table of Contents i
Table of Figures iii
Table of Tables v
Acknowledgements vi
Abstract vii
Abbreviations, Symbols and Notation viii
Chapter 1 Introduction 1
Chapter 2 Literature Review 4
2.1 Acceleration Structures 4
2.2 SDF Representation 8
2.3 SDF Reconstruction in Real-Time 11
Chapter 3 Methodology 13
3.1 Structure 13
3.2 Rendering 14
3.3 Construction 15
3.3.1 Top-Down Construction 16
3.3.2 Edit Dependencies 17
3.3.3 Brick Counting 19
3.3.4 Scanning 20
3.3.5 Brick Building 21
3.3.6 Edit Culling 22
3.3.7 Brick Evaluation 23
3.5 Testing and Evaluation 24
3.5.1 Method and Tools 25
3.5.2 Metrics 27
3.5.3 Test Scenarios 28
Chapter 4 Results 30

4.1 Rendering 31
4.2 Construction 34
4.3 Memory 40
4.4. Visual Fidelity 42
Chapter 5 Discussion 45
5.1 Results Analysis 45
5.1.1. Rendering Performance 45
5.1.2. Construction Performance 47
5.1.3. Memory Usage 51
5.1.4. Visual Fidelity 52
5.2 Critical Evaluation 54
Chapter 6 Conclusion and Future Work 57
6.1 Overview 57
6.2 Future Work 58
References 60
Appendices 62
Appendix 1 – Rendering Complete Data 62
Appendix 2 – Construction Complete Data 65

Table of Figures

Figure 1: A comparison between the minimum (left) and smooth minimum (right), as defined by (Quilez, no date), of two functions. 9
Figure 2: Two iterations of brick building. 17
Figure 3: An example of identifying edit dependencies and selecting relevant edits for a brick. Edits 1, 2, and 3 would be selected and edit 4 culled. 19
Figure 4: Spatial coherence of the bricks. Hue increases with index into the buffer. 21
Figure 5: Edit count per voxel for 'Drops' for a blending radius of 0.2 and 0.5 respectively, where green signifies 1 edit and red 16 edits or greater. 23
Figure 6: A cross-section from 'Drops' brick pool, with a magnified excerpt on the right. 24
Figure 7: The function used to obtain the memory consumption of the brick pool. 25
Figure 8: The ground-truth render of the ‘Spheres’ scene for evaluating fidelity. 27
Figure 9: The test scenes (in row-major order): Drops, Cubes, Rain, and Fractal. 29
Figure 10: Average time taken to raytrace each scene for each brick size. 31
Figure 11: Average time taken to raytrace each scene as brick count increases. 32
Figure 12: Average Raytracing Core Throughput for each brick size for each scene 32
Figure 13: Average L2 cache hit rate for each scene for each brick size. 33
Figure 14: Average raytracing time for each scene with a brick size of 0.0625 with edit culling enabled and disabled. 33
Figure 15: Construction time for each scene for each brick size with edit culling enabled. 35

Figure 16: Construction time for each scene for each brick size with edit culling disabled. 36
Figure 17: A direct comparison between construction times with edit culling enabled and disabled. 36
Figure 18: The time taken by each construction stage, as a percentage of total construction time. 37
Figure 19: SM throughput as brick size increases for each scene, with edit culling enabled. 37
Figure 20: SM Throughput for each scene and each brick size, with edit culling disabled. 38
Figure 21: GPU Occupancy for each construction stage as brick size increases. 38
Figure 22: Unit throughput during brick evaluation for the ALU and LSU for each scene and brick size. 39
Figure 23: Warp Launch Stall Reason for each scene and brick size. 39
Figure 24: Memory usage for each resource for each scene in MB. 40
Figure 25: Resource memory usage, displayed as a percentage of the object's total memory usage. 41
Figure 26: Memory usage in MB by brick count. 41
Figure 27: The fidelity test scene rendered at brick sizes (left-to-right) 0.5, 0.25, 0.125, 0.0625. 42
Figure 28: The difference between each fidelity test and the ground truth for brick sizes (left-to-right) 0.5, 0.25, 0.125, 0.0625. 43
Figure 29: Figure 27 with highly exaggerated colours for clarity. 43
Figure 30: Noise can be seen in the surface normals. A series of magnifications are displayed on the right. 43
Figure 31: The Drops scene rendered with edit culling (left) and without edit culling (right). 43
Figure 32: Error in the edit culling algorithm. The differences are imperceptible when not exaggerated artificially. 44
Figure 33: Error in the edit culling algorithm, greatly exaggerated and magnified to be perceptible. 44

Table of Tables

Table 1: Metrics collected for rendering. 27
Table 2: Metrics collected for construction. 28
Table 3: The number of edits composing each scene. 30
Table 4: The number of bricks composing each scene when edit culling is enabled, for each brick size. 30
Table 5: The number of bricks composing each scene when edit culling is disabled, for each brick size. 31
Table 6: Memory usage in MB of each resource for each scene. 40

Acknowledgements

I would like to sincerely thank Erin Hughes, whom I am incredibly grateful to have had as my supervisor, and whose wisdom and enthusiasm inspires me to produce my best work.

I am also exceptionally thankful to my partner Viktora, for her infinite support and for putting a smile on my face every day.

Finally, I would like to thank my parents. I am incredibly proud to be your son. Thank you for teaching me to be hardworking, to believe in myself, and to aim for the stars.

Abstract

Signed distance fields (SDFs) are an implicit representation of geometry with a collection of useful properties, e.g., affinity for constructive solid geometry, and an intrinsic definition of interior versus exterior. They are useful for sculpting tools, deformable objects, fluids, and volumetric effects. These techniques can be challenging to perform with polygons. Until recently, the use of SDFs in real-time interactive applications has been limited due to performance and memory constraints. Several studies have documented how discrete SDFs can be rendered in real-time. However, the study of SDFs that are also modifiable in real-time has not been treated in depth.

This study aims to implement a memory-efficient representation of SDF geometry that can be rendered and modified in real-time and evaluate the feasibility of using SDFs as a rendering primitive within games.

A sparse implementation of SDFs was designed, and an application was developed with C++ and DirectX 12 to render SDFs using hardware-accelerated raytracing and software sphere-tracing. A top-down construction algorithm was developed that hierarchically refines space and uses culling solutions to accelerate distance field evaluation.

Results showed that scenes of hundreds of thousands of spatial primitives could be rendered in 2-10ms, and that this is scalable with the number of primitives. It also finds that efficiently culling primitive shapes is key for construction performance, where culling improved construction times from multiple seconds to under 100ms, often under 20ms.

The study concludes that it is feasible to make use of modifiable SDFs in a real-time interactive application. With further work to reduce error in surface normals, this technology could be applied in a game context. Future research could also further improve culling solutions and integrate shading attributes into the SDF representation.

Abbreviations, Symbols and Notation

AABB	Axis-aligned Bounding Box
ALU	Arithmetic and Logic Unit
BLAS	Bottom Level Acceleration Structure
BVH	Bounding Volume Hierarchy
CPU	Central Processing Unit
CSG	Constructive Solid Geometry
CSV	Comma Separated Values
CTA	Co-operative Thread Array
DXR	DirectX Raytracing
GPU	Graphics Processing Unit
GSM	Group-Shared Memory
LOD	Level-of-detail
LSU	Load-Store Unit
RAM	Random Access Memory
RT	Raytracing
SDF	Signed Distance Field
SM	Streaming Multiprocessor
SVO	Sparse Voxel Octree
VRAM	Video Random Access Memory

Chapter 1 Introduction

Polygons have long been the staple representation of 3D geometry for graphical applications. They provide an efficient explicit description of planar surfaces, and graphics hardware has evolved to rasterize them very efficiently. However, they are not the only choice, and the more powerful general computation capabilities of modern hardware is allowing for the use of alternative representations of geometry to be explored. As one such alternative, signed distance functions provide a simple implicit representation of geometry and show interesting properties such as support for constructive solid geometry (CSG) operations, shape morphing, and an intrinsic definition of interior versus exterior, all of which are typically challenging to achieve with triangle geometry. Furthermore, unlike a discrete representation of geometry, signed distance functions are continuous and resolution independent. This representation is well suited for deformable surfaces, volumetric effects, and smooth organic shapes.
As they can be rendered using a simple sphere-tracing algorithm (Hart 1995), signed distance functions have seen extensive use through the computer graphics demo-scene for decades, where unique graphical effects not seen in games of the time were created with software such as *Shadertoy* (Quilez, no date). However, adoption in real-time graphics applications has been limited until recently. This is due to issues of scale. Dense scenes with many objects require many signed distance function evaluations, making real-time rendering of signed distance function-based scenes infeasible.

To decouple rendering time from the number of primitives, an offline process can be performed where the signed-distance functions can be sampled regularly and stored in a grid – forming what is known as a signed distance field (SDF). Therefore, distance values from the distance field can be acquired in constant time with respect to the number of primitive signed distance functions.

However, there are drawbacks and challenges with this volumetric representation of SDFs. Firstly, what before was a compact functional representation is now a 3D volume of samples that involves a significant

memory overhead for large objects. Secondly, the resolution-independent continuous functional representation is sacrificed for the limited precision of discrete samples, where volumes of high resolution are required to create visually continuous geometry. Finally, functional representations allow the shape and resolution to be fully dynamic, as the functions must be re-evaluated every time the scene is rendered regardless. Indeed, this is what made signed distance functions so popular within the graphics demo-scene. Modification of the static volume data associated with an SDF is significantly more challenging.

Many approaches to efficiently store a distance field in a discrete volume have been investigated. There have been various motivations and applications for this, e.g., interactively rendering medical scans (Crassin, 2009), where a volume of data may contain billions of samples, and to create a unified representation of 3D geometry using voxels, to replace the requirement for high resolution displacement maps to bring detail to coarse triangle geometry (Laine & Karras, 2010). *Dreams* (Evans, 2015) is a groundbreaking example of using sparse volumes of distance fields in games. Using SDFs as its representation of geometry allows intuitive in-game sculpting tools, which enhance the user-generated content experience.

Notably, *Claybook* (Aaltonen, 2018) shows how a fully deformable world can be constructed from SDFs, and it is a strong example of how a game can benefit from using SDFs as a dynamic representation of geometry. However, *Claybook* does not make use of sparse volumes, and the entire world is effectively one single object which limits how its approach can be used for games in general.

So far, there has been little discussion about how a modifiable and dynamic distance field can make use of a sparse and memory-efficient representation within the context of a game.

This study researches the feasibility of real-time construction and rendering of signed distance field geometry and evaluate how it can be used as a real-time rendering primitive in games in general.

It aims to efficiently represent SDF geometry with a sparse volume which can be rendered using raytracing. A construction algorithm is developed

that will allow the SDF geometry to be modifiable and interactive in real-time. It is hoped that this representation of geometry can demonstrate the unique properties of SDFs, such that they could become more commonplace in games in the future.

Chapter 2 will review relevant literature surrounding the development of sparse representations of SDFs that can be rendered efficiently and existing methods of constructing or modifying SDF geometry on the GPU. Chapter 3 will cover the implementation details of the technology developed in this research to represent, render, and reconstruct SDF geometry. The method used to evaluate the developed technology will also be discussed. The results obtained through testing are presented in Chapter 4, followed by a thorough discussion and analysis of these results in Chapter 5. Finally, conclusions on this research will be drawn in Chapter 6.

Chapter 2 Literature Review

A variety of techniques have been investigated to approach rendering large SDFs in real time. The challenge associated with rendering volumetric data arises due to the vast quantity of data involved. The naive approach of visiting every sample along a ray, while precise, would be far too slow to be useful in a real-time application. An acceleration structure that can partition space is required, which allows for as few samples as possible to be visited.

2.1 Acceleration Structures

An example of such an acceleration structure is employed in *Gigavoxels* (Crassin, 2009), which proposes a novel out-of-core method to render extremely large volumetric data sets in real time. It implements a sparse tree structure where each node points to either a 'brick', an N^3 block of distance data, or a constant value in the case of a homogenous region of space. Sibling nodes are stored adjacently so that children can be accessed through the pointer to the first child to reduce the amount of data to be stored per node.

Bricks are stored separately from the nodes in a 'brick pool'. Non-leaf nodes also own bricks, which allows for mip-mapping when sampling the volume. This produces an inherently filtered and anti-aliased image when the volume is rendered, which is a very desirable attribute in the context of graphical applications such as games.

The volume is rendered using raytracing and can optionally be encased in proxy geometry that is first rasterized. The node tree is descended using a kd-restart algorithm, to avoid the need for a stack, until the desired level of detail is reached. At this point, if the node contains a homogenous value, it is integrated along the length of the ray. Otherwise, the associated brick is fetched from the brick pool and is ray-marched to accumulate density along the ray. In the case of opaque geometry, traversal can be discontinued upon the first surface intersection.

Gigavoxels also demonstrates a novel and advanced ray-guided streaming method, where the GPU efficiently feeds back to the CPU what

bricks were accessed, and which missing bricks are required to be loaded into the brick pool for the next frame.

While out-of-core rendering is an essential concept for rendering vast voxel-based worlds, such as would likely be encountered in a game, it is outside the scope of this research.

Using this method, volumes of size greater than the available video memory can be rendered at real-time framerates. This allows the size of the brick pool to be adjusted to match the available hardware capabilities – a smaller brick pool will consume less video memory at the expense of being able to store less bricks at a time. This sort of dynamic configuration is important in a game context, where a technology may need to adapt to various hardware specifications. Nevertheless, *Gigavoxels* demonstrates how large amounts of voxel data can be arranged and dynamically updated in a tree structure, which is an important idea for developing a structure that can support the modification of SDF objects.

Efficient Sparse Voxel Octrees (Laine & Karras, 2010) is motivated by the idea that an efficient voxel-based representation of geometry can unify coarse geometry and fine detail. It investigates a memory-efficient representation of voxel data that is also fast to render using a raytracing algorithm. A compact data structure is presented, where each voxel is represented by a node in a sparse octree. Each node contains bitmasks from which the state of its children can be determined, and therefore the leaf voxels do not need to be stored directly. Nodes are also stored compactly, as the bitmasks also describe the spatial relation between a parent and each of its children.

The concept of contours is also applied to aid rendering performance. Contours are two parallel planes which bound the surface within a voxel. Contours can be stored per-voxel and allow raytracing to be accelerated by providing a tighter-fitting bounding volume compared to simply using the voxel's cubic bounding box. This allows a ray to visit less voxels when searching for an intersection.

Efficient SVO's also tackle out-of-core rendering through splitting the tree structure into 'blocks' and using relative pointers within each block. Entire blocks can then be streamed in and out of memory as required. Traversing through the octree can be done efficiently, as the index of the next node to visit can be found by flipping bits of the current node index, depending on the ray direction. A stack of indices is maintained that fully describes the path from the root of the tree to the current node.

Laine & Karras demonstrate that an Efficient Sparse Voxel Octree is an effective method of compactly storing geometry within a game context and state that their implementation is compact and efficient. However, this is under the assumption of static geometry. Therefore, Efficient SVO's present useful ideas for the compact storage of voxel data but do not provide a method to efficient construct or modify the structure that takes advantage of the massive parallelism offered by the GPU. The additional overhead of calculating contours for each voxel is additional sacrifice to construction speed in favour of rendering speed, again making this representation better suited to static geometry.

Traditionally, as with Efficient Sparse Voxel Octrees and *Gigavoxels*, tree structures would be constructed iteratively one level at a time. Maintaining a compact and memory-efficient structure is trivial as each iteration can simply subdivide the previous level as appropriate. However, this significantly reduces the extent to which parallelism can be exploited. This is acknowledged by (Karras, 2012), who demonstrates this can be solved using an efficient parallel tree construction algorithm for spatial data that is well suited for the GPU by utilizing the spatially coherent properties of Morton codes. Morton codes describe the path to take through an octree to reach a leaf. This property is utilized such that each interior node can calculate the split position of all its children in-place, based on the depth of the first difference in the bit representations of the Morton codes. By allowing all interior nodes of the tree to be processed independently, all nodes in the tree can be constructed simultaneously. This is clearly superior to constructing each level of the tree iteratively. The performance

gains are particularly pronounced for large workloads that can sufficiently fill the GPU to take advantage of all its available resources.

However, to make use of this algorithm, a complete set of the leaf nodes are required to be in sorted order of their Morton codes. Therefore, this algorithm is well suited to applications that involve building an acceleration structure around existing spatial data - for example, to construct a bounding volume hierarchy to accelerate collision detection or raytracing. This algorithm is not applicable in the case of constructing new spatial data, where the next level in the tree depends on the properties of the preceding level. In such a case the tree must be constructed one level at a time, as interior nodes can no longer be evaluated independently.

(Evans, 2022) perform a thorough comparison of SDF traversal methods and intersection methods, resulting in a proposed novel analytic voxel intersection method. This method works by representing the iso-surface within a voxel (in this case, a 2^3 grid of distance field samples) with a cubic polynomial. The cubic polynomial defining the iso-surface can be differentiated to give a quadratic polynomial, from which the solutions detail any ray segments containing intersections with the surface. Newton-Raphson iteration can then be performed to quickly converge on the point of intersection. This method is compared to other analytical methods of intersection, such as repeated linear interpolation and analytical cubic polynomial solvers, as well as sphere tracing as an example of an iterative method. It found that the analytic methods of calculating intersection with the iso-surface performed significantly better than sphere-tracing. While this is a useful result for the rendering of SDF geometry, methods of calculating intersection have no bearing on construction performance.

A comparison of grid traversal algorithms is also performed. It compares grid sphere-tracing – the method used in *Claybook* (Aaltonen, 2018), a sparse voxel set – where every voxel is uniquely associated with a raytracing bounding box, sparse brick set – like the method used in *Dreams* (Evans, 2015), and sparse voxel octree – as described by (Laine & Karras, 2010). It was found that grid sphere tracing performed best in open and simple scenes, whereas sparse brick set and sparse voxel octree

performed well compared to other techniques in complex scenes. It is speculated that this is due to improved cache coherency as the data is stored more compactly. However, sparse voxel set outperformed all other methods in general by leveraging DXR hardware therefore requiring less work in the custom software intersection shader.

While the rendering performance of various SDF traversal and intersection methods is thoroughly analysed, it does not touch on which methods are best suited to fast reconstruction. Nevertheless, as grid sphere-tracing is very similar to *Claybook* (Aaltonen, 2018), it is known that it is a suitable structure for modification. Sparse voxel set is a simpler data structure that relies on raytracing hardware, and consequently could be simpler to construct. However, it does have a much more significant memory overhead compared to the other methods. Sparse brick set is a good compromise between grid-sphere tracing, which has been shown to work with real-time modification, and sparse voxel set, which outperformed the other methods in Evans study.

2.2 SDF Representation

Dreams (Evans, 2015) is a strong example how SDF-based geometry can be applied within a commercial game. SDFs have a natural affinity for constructive solid geometry (CSG), where solid 3D shapes can be constructed through the unions and subtractions between primitive shapes. This is used to create simple and intuitive sculpting tools – which is favourable within a user-generated content game. Objects in *Dreams*, called ‘Things’, are constructed from lists of ‘edits’, where an edit is a primitive shape described by an analytical signed distance function. Edits can only be union-ed with or subtracted from other edits, forming an entirely right-leaning CSG tree. This simplifies to a linear list of edits to be applied in order. As well as supporting binary union and subtraction operations, *Dreams* allows for smoothed addition and subtraction, where the shapes blend into each other over a specified radius. These smooth operations are implemented using a ‘smooth minimum’ function, an example of which is shown in Figure 1. However, as this function has an infinite range – it

reaches zero at infinity – *Dreams* uses a formula for smooth operations that will blend over a fixed radius (Evans, 2015, pp. 30).

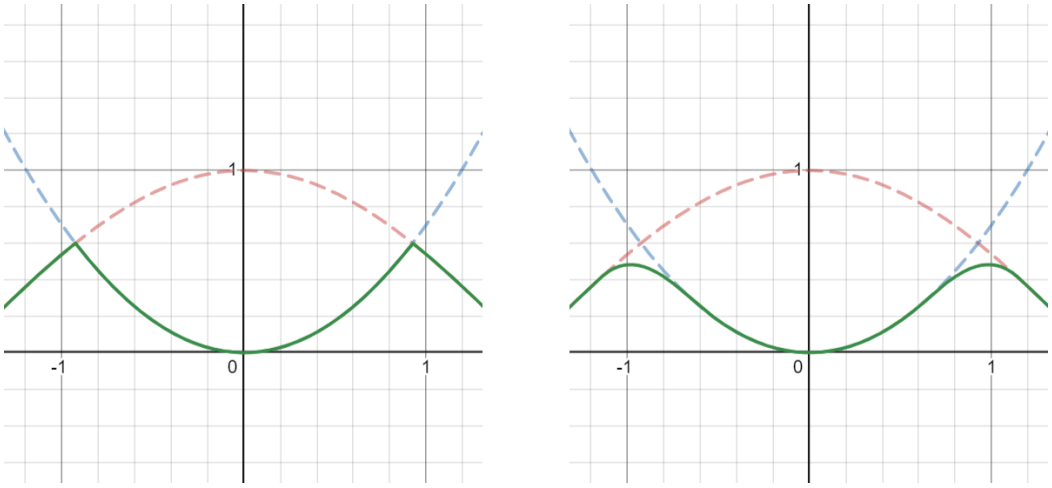

Figure 1: A comparison between the minimum (left) and smooth minimum (right), as defined by (Quilez, no date), of two functions.

Dreams does not directly render the list of analytical signed distance functions in their functional form, but instead uses a series of compute shaders (the 'evaluator') to discretize these functions into a sparse distance field. The resulting voxels in this distance field are then grouped into 'bricks' of 8^3 voxels.

For the evaluator to be useable in a real-time application, edit culling is performed as aggressively as possible. This is achieved through hierarchically subdividing around the edit surfaces until a precise list of edits per voxel is acquired. The balance between fidelity and speed is complex. For example, not subdividing enough results in cracks in the surface, but subdividing more than necessary results in additional unnecessary work and greater latency. Through its advanced culling solutions, *Dreams* can evaluate objects with many thousands of edits.

It is interesting to note that *Dreams* uses max distance norms (Evans 2015, pp. 29). Instead of a Euclidean distance, the primitive distance functions calculate the greatest component of the absolute value of the distance, i.e., either the distance along the x, y, or z axis, whichever is largest. Consequently, distance functions can be simpler and faster to compute as generally less square root operations are required. Furthermore, some primitives, such as ellipsoids, do not have exact Euclidean distance functions but do have an exact max norm distance

function. Finally, instead of distances implying a bounding sphere, they imply a bounding box, which simplifies evaluation schemes that perform culling using a grid. However, in a raytracing context, max norms can cause issues for secondary rays which will originate from an objects surface. This is not an issue for *Dreams* as it does not use raytracing.

This approach shares similarities with the approach taken by *Gigavoxels* (Crassin, 2009), with key differences to make the technology useful for a game. In *Gigavoxels*, the entire world is represented by one single volume, and this is impractical for use in games. *Dreams* overcomes this by, instead of tracing rays from the eye into a hierarchy of bricks, choosing a view-dependant cut through the tree of bricks in advance and rasterizing their bounding boxes. This also removes the need for any indirection within the inner ray-marching loop, as each rasterized cube directly corresponds to a single brick.

Once rasterized, parallax occlusion mapping (Tatarchuk, 2005) is performed to reveal the SDF iso-surface within each brick. The dimensions of the bounding boxes are adjusted to maintain a constant screen-space size – which provides appropriate filtering of the distance field data to produce an anti-aliased image.

While interactive CSG with SDF's is a core gameplay mechanic for *Dreams*, it does not allow for the edits themselves to be modified once evaluated. This means that while Things in *Dreams* benefit from the CSG properties of SDFs, the edits within an object are static. With newer and more powerful hardware becoming available, an investigation into dynamism within objects through on-the-fly animation of the edits is worthwhile.

In all studies seen thus far, real-time uses of signed distance functions rely on initially sampling the functions into a discrete grid of distance data. However, an alternative approach is presented by (Aeva, 2022) that avoids discretization altogether. The analytical signed distance functions are sphere-traced as the scene is rendered, as often seen in *Shadertoy* (Quilez, no date).

To make this feasible for real-time rendering, an offline step is performed that analyses the CSG tree to compile a large set of shaders that can each correctly render a portion of the scene while performing the minimum number of signed distance function evaluations per ray. While an interesting and unique approach, unfortunately it is not suited to real-time modification as continuously recompiling shaders at runtime is not feasible, and not an intended nor effective use of the GPU hardware.

2.3 SDF Reconstruction in Real-Time

There are much fewer examples of how continuous reconstruction or modification of SDF geometry can be performed in real-time. *Claybook* (Aaltonen, 2018) is one key example of a fully real-time deformable SDF-based world in a game. This world is formed of one global SDF of resolution 1024x1024x512, from which a series of five mip levels are constructed. This single SDF volume is rendered using a sphere-tracing algorithm. Each subsequent mip level encodes distance values twice as great as the previous level, and this is used to accelerate sphere-tracing. By sampling coarser mip levels, greater distances can be travelled in each sphere-tracing iteration while still travelling conservatively. The magnitude of the distance value sampled at the current location informs the algorithm when it is appropriate to ascend or descend mip levels.

Modification is performed through allowing pre-defined ‘brushes’ to be applied to the world SDF. *Claybook* stores the world SDF as a dense volume; empty and non-empty regions of space are stored in memory alike. This allows the algorithm to make use of the locality of the brushes; a brush has a deterministic range and only the nearby affected areas of the world SDF need to be re-evaluated when a brush is applied. This would not be possible in the case of a compacted data structure, as location in the data structure cannot be inferred from location in space without maintaining an indirection table of some form. Even with such a structure, the number of surface-intersecting voxels is not static, and the structure would require insertions, which is a slow and difficult operation to implement. The conflict of compaction and dynamism would require the structure to be rebuilt from scratch per change. Therefore, it would no longer possible to take

advantage of the locality of the brushes if the volume were stored in a compact form like the methods detailed in Efficient Sparse Voxel Octrees (Laine & Karras, 2010) or *Gigavoxels* (Crassin, 2009). This demonstrates a general trade-off between the memory efficiency of a data structure and its ability to be easily modified.

AMD's *Brixelizer* (Kramer, 2023) builds a global SDF for a scene of triangle geometry in real-time, with support for dynamic geometry. This SDF can then be used for fast calculation of ray-scene intersections for purposes such as global illumination.

While it shares the goal of the present research of constructing an SDF every frame, it is in a slightly different context – where one SDF is created for an entire scene around existing triangle geometry. As only a single SDF is created per scene, it can be generated in a view-dependent manner using several cascades of decreasing resolution to improve construction performance. *Brixelizer*, like other technologies that have been discussed, constructs one global distance field that is constructed from a collection of local distance fields called 'Bricks'. Bricks are constructed around the surface of geometry in the scene. Once all bricks have been constructed, a tree of the axis aligned bounding boxes (AABBs) of the bricks is constructed in a bottom-up fashion. This could make use of a parallel tree construction algorithm such as described by (Karras, 2012). Instead of evaluating a distance per sample independently, a jump-flooding algorithm is used to populate samples that do not directly intersect with the surface. First, all samples that intersect with the surface are set to the minimum representable value, and then distance values are extrapolated for all remaining samples in the volume.

This is useful where calculating a distance to the surface is significantly more complex than a binary intersection test, as is the case with voxelizing triangle geometry. However, in the case of evaluating analytical signed distance functions, where the product of an intersection test is the distance value itself, this method offers no benefit.

Chapter 3 Methodology

An application was developed using DirectX 12 and DirectX Raytracing to construct and render SDF-based objects, with the focus on optimization for objects that are re-constructed continuously. As there is a more significant wealth of literature based on rendering sparse distance fields, this research focused on how they can be constructed quickly by making effective use of the parallelism of a modern GPU.

The same terminology as used in *Dreams* (Evans, 2015) is utilized – where an 'edit' is a primitive analytical signed distance function, and a 'brick' is a small cubic volume of discrete distance field data. In this case, a brick is defined as a single raytracing AABB that contains 6^3 distance samples, with an additional one-voxel neighbourhood giving 8^3 samples in total.

To allow for simultaneous construction and rendering, each object will possess two full sets of resources, such that one full set of resources can be written to without waiting for frames in flight to complete. This doubles the memory usage of an object, but crucially reduces stalls on both the CPU and GPU timelines and allows for asynchronous construction to be implemented with ease.

3.1 Structure

The aim for the artefact was to design an SDF geometry representation that is applicable to games applications in general. Therefore, no assumptions can be made about the style of the geometry that will be represented.

The work of (Evans, 2022) stated that a sparse brick set is likely the best to use in a memory-constrained scenario as it is faster than a sparse voxel octree and uses the least memory. The success of *Dreams* (Evans, 2015) also showed that a brick-based approach is applicable to real-time games applications. *Gigavoxels* (Crassin, 2009) also demonstrates how out-of-core rendering and mipmapping-based LOD systems can be achieved using bricks.

An additional attraction to a brick-based structure is that work for evaluating the distance field can naturally be divided in a GPU-friendly manner. An 8^3 brick of data can be processed by a thread group of 8^3 = 512 threads. (Evans, 2015) also uses 8^3 samples per brick, as that matched the wavefront size of the targeted GPU hardware.

Even though each brick contains 8^3 samples, only the inner 6^3 samples are sphere-traced. As sampling with linear filtering will gather the 8 surrounding values and perform interpolation to achieve a single filtered sample, a neighbourhood of voxels is duplicated for each brick to avoid the sampler from crossing brick boundaries. This does result in 58% of each brick being solely dedicated to adjacency data but enables the use of texture sampling hardware to perform the interpolation. Additionally, the performance implications of divergent reads at voxel boundaries are avoided.

For these reasons, bricks were decided upon to be the core of idea of the structure used in this application.

3.2 Rendering

The implemented rendering method is similar to the ‘sparse brick set’ method described by (Evans, 2022) as it also makes use of hardware accelerated raytracing. A raytracing AABB is constructed for every brick. These bounding boxes are placed into a bounding volume hierarchy (BVH) to allow for hardware-accelerated raytracing. Only the leaf bricks are required to build the BVH. All other levels of the hierarchy are discarded after construction. If, in future, a level-of-detail or mip-mapping scheme was introduced, similar to the one present in *Gigavoxels*, the full hierarchy of nodes would need to be retained.

Once the hardware raytracing detects a potential intersection between a ray and an AABB, a software intersection shader is invoked. The t_{min} and t_{max} of intersection with the AABB are calculated, and this gives a ray interval over which to check for intersection with the isosurface. The point of intersection is transformed into the bounding box’s local coordinate system, ranging from [0,1].

An intersection between the ray and the isosurface described by the distance field within the brick is found iteratively using sphere-tracing (Hart, 1995). Despite (Evans, 2022)'s findings that analytical intersections performed faster, sphere-tracing is utilized because it is a simpler algorithm, and the focus of this research is on construction methods. Even with sphere-tracing, rendering is generally much faster than construction, and consequently optimization efforts were focused on construction instead. Nevertheless, it is likely that rendering performance could be increased by implementing an analytical voxel intersection test as per (Evans, 2022).

Distance values are sampled from the volume using trilinear sampling. Distances need to be transformed from the formatted form stored in the volume texture to distances in the space encoded within the AABB – where the sphere tracing is being performed. The formatting of distance values is discussed in section 3.3.7.

As all geometry within a brick is opaque, sphere-tracing can be terminated upon the first intersection with the surface. The intersection point is transformed back into object-space, so that the t-value of intersection can be calculated as the distance between the object-space ray origin and the object-space position of intersection.

Surface normals are calculated using central differencing to find the gradient in the distance field at the point of intersection. The tetrahedron technique (Quilez, no date) would be an improved method, as it only requires 4 samples as opposed to 6.

3.3 Construction

Constructing an SDF object can be divided into 3 stages. First, the edit list is analysed to identify dependencies between edits. Secondly, hierarchical brick construction is performed which consists of sub-stages described in sections 3.3.3 through 3.3.6. This set of sub-stages can be performed iteratively until the desired brick size is reached. Finally, the edits for each brick can be evaluated to populate the brick pool with the distance field data. Bounding boxes for the raytracing acceleration structure are also constructed in this stage.

Initially it may seem like a structure that could be partially reconstructed would be ideal; for example, if an edit is applied that only affects one octant of the object, then only that octant should be reconstructed. This idea is applied in *Claybook* (Aaltonen, 2018). However, as discussed, this is not practical with a compacted data structure. Therefore, the structure implemented is rebuilt from scratch upon every modification.

3.3.1 Top-Down Construction

A top-down approach to construction is taken, where the algorithm begins at the coarsest level of bricks and iteratively refines until the desired brick size is reached.

Performing construction this way has several advantages. For example, larger regions of space can be culled from further evaluation. If a brick is culled in an early iteration, then all space within that brick is never required to be visited again. This allows for large areas of space to be culled quickly and much larger and sparser objects to be feasible to construct. An example of how an object is hierarchically refined is shown in Figure 2.

Edit index buffers are also refined at each stage of construction, which reduces the number of edits that must be evaluated in the brick counting stage in the next iteration. As edit evaluation is generally the bottleneck of construction, evaluating as few edits as possible is critical to real-time performance. Edit index buffers are described further in section 3.3.2.

It also simplifies the process of ensuring that all bricks are sorted in a spatially coherent order, which helps cache-coherence. If each brick sorts its sub-bricks in a spatially coherent order in each iteration, then that is enough to guarantee the entire buffer is sorted when brick building completes. This allows all sorting to be performed within each compute shader group –no global sorting stage is required.

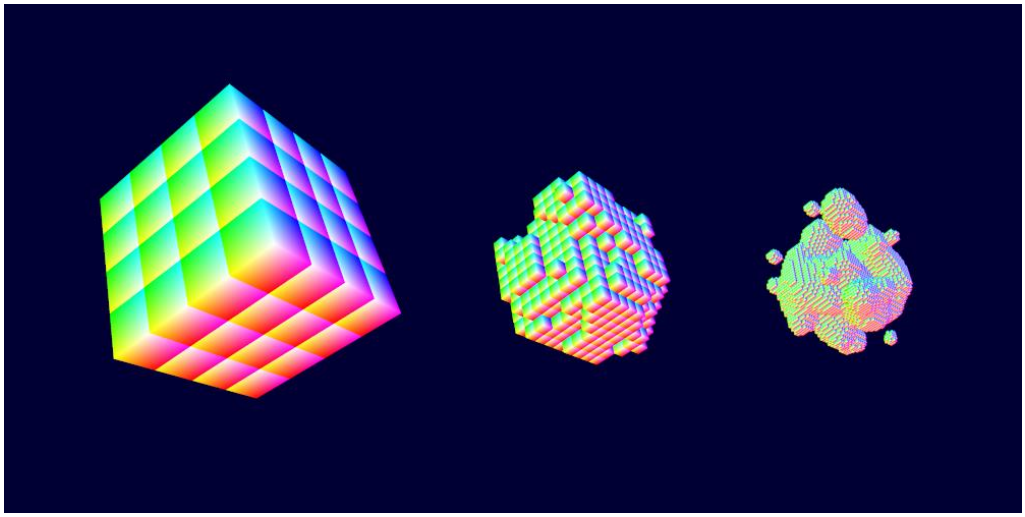

Figure 2: Two iterations of brick building.

However, there are also disadvantages to top-down construction. These issues are overviewed by (Karras, 2012) and discussed in Chapter 2. As the highest levels of the hierarchy contain few bricks, parallelism is severely limited in the first two iterations of construction. The first iteration will only contain 64 bricks – which is far from enough to fill the GPU with work. The second iteration can contain up to 4096 bricks, which is still not enough to fully occupy the GPU. However, as the tree is wide, the number of bricks can increase by up to a factor of 64 each iteration. By the third iteration this issue dissipates. This issue would be more pronounced with a narrower tree, such as an octree, where the number of bricks grows slower with each iteration.

3.3.2 Edit Dependencies

Generally, not every distance sample within the object will be affected by every edit. To allow for efficient evaluation of the distance field, the minimum number of edits should be evaluated for every sample. However, maintaining edit relevancy data on a per-sample basis would require significantly greater memory overhead and increase thread divergence. Therefore, edits are culled at a per-brick granularity. This allows all threads in a group to co-operate to build relevancy data for each brick and reduces divergence within a group during evaluation, as every thread will evaluate the same edits in lockstep.

The indices of all edits relevant to a brick are stored. Only one index buffer is allocated for each SDF object, and each brick sub-allocates from this buffer. In the artefact, it was defined that edits lists may contain up to a maximum of 1024 edits. This allows for 16 bits to be used per index. The full construction of index buffers is described in section 3.3.6.

To ensure index buffers are constructed correctly, dependencies between edits must be identified. As the edit list is the same for all bricks, edit dependencies can be computed a single time prior to brick construction, and re-used throughout each iteration of brick building.

An edit dependency occurs if any edit in the list will influence the effect of a subsequent edit. This is only relevant when using smooth edits – where an edit could produce different geometry depending on the preceding edits and amount of blending. To be able to correctly cull smooth edits, a precise analysis of edit dependencies is required.

A smooth edit is dependent on a preceding edit if their ranges of influence overlap – where the range of influence is the boundary of the edit inflated by its smooth blending radius. To make this calculable, the smooth-minimum function with a fixed radius as used by (Evans, 2015) was utilized. Determining the intersection of two signed distance function primitives is not trivial, so a simplification where conservative bounding spheres were used for all primitives in this artefact. This is a poor approximation in many cases, but correct geometry will still be produced at the expense of more edits being evaluated than necessary. A more thorough implementation of signed distance function intersection detection would improve the efficiency of edit culling.

An example is shown in Figure 3, where the boundary of one brick is displayed, as well as 3 edits and their ranges of influence. The edits are enumerated in the order they appear in the edit list. In this example, it is obvious that the red edit is relevant to the brick. The purple edit is also relevant, as its range of influence extends within the brick and therefore can affect the geometry within. While neither the green edit nor its range of influence is within the brick, as its range of influence overlaps with the red edit’s range of influence, it is determined to be dependent on red and therefore is also relevant to the brick. Yellow is not involved in any

dependencies nor intersects the brick in any way and can therefore be culled.

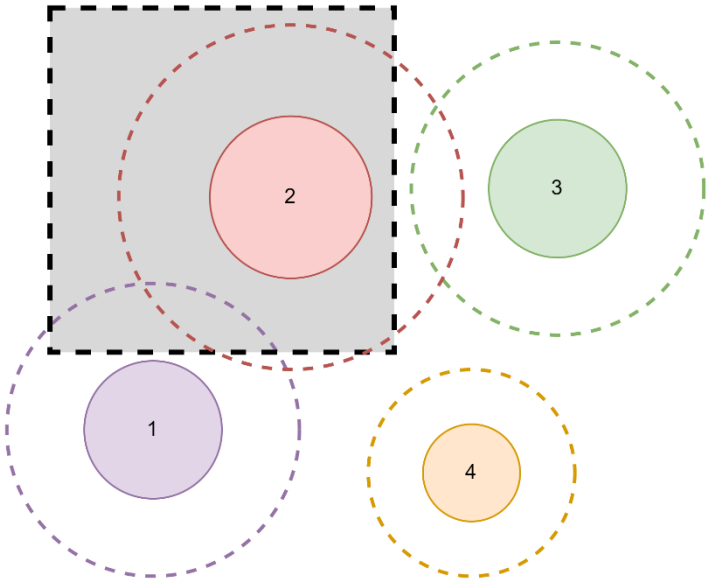

Figure 3: An example of identifying edit dependencies and selecting relevant edits for a brick. Edits 1, 2, and 3 would be selected and edit 4 culled.

Each thread in a compute shader dispatch is assigned a pair of edits to process. By dispatching a linear array of threads, and by using the largest triangular number less than the thread's index, a unique edit pair for each thread can be identified. In this scenario, work is balanced between threads helping to maximise occupancy.

If edit A is a smooth edit, the thread will check if it overlaps with edit B. If so, the index of edit B is inserted into edit A's dependency list. If edit B is also a smooth edit, then the index of edit A is also insert into edit B's dependency list. Due to the implementation of the edit culling algorithm, dependencies can be inserted in any order.

The dependency buffer could be compacted once it has been populated. It could be investigated if the time spent compacting the buffer results in a net reduction in construction time due to improved data locality in the dependency buffer.

3.3.3 Brick Counting

A compact buffer of bricks is maintained throughout the construction process. This is done to avoid the need for excessively large intermediate buffers. To achieve this, the first step in each iteration is to determine the

quantity of sub-bricks each brick will be divided into. This is done by evaluating the edit list for each potential sub-brick to determine if it will be intersected by the surface. Only edits referenced by the parent brick's index buffer need be evaluated. Every sub-brick intersected by the surface is marked as such using a bitmask within the parent brick. This allows the sub-bricks that require construction in the brick building stage (section 3.3.5) to be recovered without re-evaluating the edit list. The sub-bricks cannot be constructed immediately as the indices at which they will be placed into the buffer need to be calculated first.

As each group processes a single brick, and a brick can be split into 64 sub-bricks, each group consists of 4^3 threads. Consequently, bricks will quarter in size with every iteration. The total number of sub-bricks that will be produced from a parent brick is recorded using a group-shared atomic counter. Once the sub-bricks have been counted, the final count is stored in global memory to allow the final indices of each sub-brick to be calculated in the next stage.

3.3.4 Scanning

The indices of all sub-bricks to be constructed can be calculated by performing a prefix sum, or scan, of the sub-brick counts for all bricks.
Many implementations of prefix sum suitable for a GPU have been developed, and the implementation described by (Harada & Howes, no date) was followed.

Multiple stages of scanning are required. First, each group of 64 threads will perform a scan. Then, the output from the first stage will also be scanned by a subsequent dispatch of 64 threads. Then, the result from each stage is summed to produce a final index offset per brick. In cases of more than 262,144 bricks, two stages of scanning will be insufficient for the second scan to fit within a single group. In this instance multiple sums are performed in the summing stage, resulting in one additional load from the scan buffer for every 262,144 bricks constructed. A better solution would have been to allow for each thread to locally scan more than one value.

3.3.5 Brick Building

With the final location for each brick within the output buffer now determined, the sub-bricks can be constructed and placed into the buffer. The location of the sub-brick can be determined from its index within its parent. Before the sub-bricks are placed in the output buffer, they are sorted by their Morton codes to ensure that the brick buffer remains spatially coherent. As construction is performed one level at a time, the entire buffer of bricks can be guaranteed to be sorted simply through sorting within each group in each iteration.

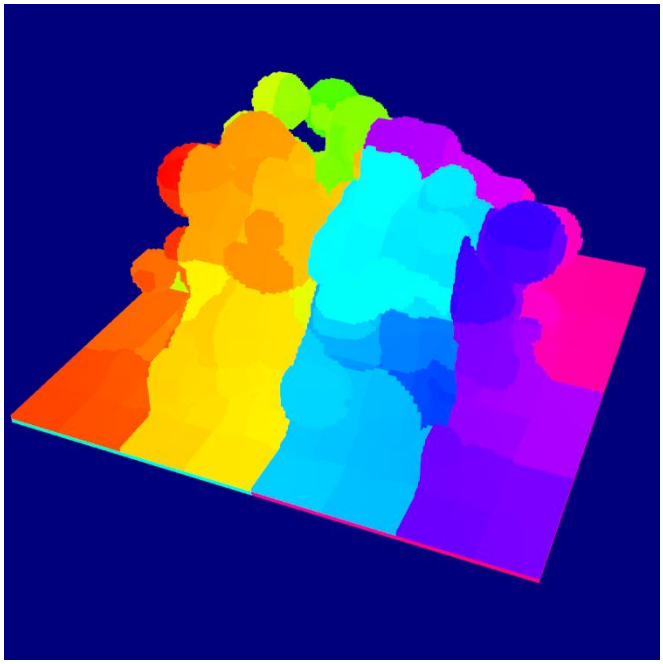

Figure 4: Spatial coherence of the bricks. Hue increases with index into the buffer.

An enumeration sort is utilized, as there can only be a maximum of 64 sub-bricks. This again takes advantage of the 1:1 mapping between threads and sub-bricks, where each thread calculates the sorted index of one brick. Once sub-bricks are sorted within a group, they are placed into the global buffer of bricks with respect to the group-local order, allowing the global buffer to remain both compact and spatially coherent. The visualization in Figure 4 shows that nearby bricks have similar hues and are therefore located nearby in the brick buffer. This is advantageous for making effective use of the cache during rendering – as nearby threads will likely access adjacent bricks, or a thread is likely to access a neighbouring brick when exiting a brick if an intersection was not found. Furthermore, it

is likely that bricks nearby in space will access similar edits, which is beneficial for cache coherency during evaluation.

3.3.6 Edit Culling

Once new sub-bricks have been constructed, a new index buffer must be constructed for each. This can be done by refining its parents index buffer. As every sub-brick is entirely encapsulated by its parent, there will never be an edit that can affect a sub-brick but not its parent.

There are two important considerations when constructing the index buffer – firstly, indices must be sorted to produce the correct geometry. A union followed by a subtraction will produce different geometry than the subtraction followed by the union. Secondly, an index should not occur more than once to avoid unnecessary work.

Both operations can be solved simultaneously with the use of a bitfield. Every edit is assigned to a corresponding bit in the bitfield. For a maximum of 1024 edits, this requires a bitfield 1024 bits wide, represented by a 32-element array of 32-bit unsigned integers.

To determine if an edit should be culled or not, edits are evaluated at the midpoint of the brick. If the evaluated distance is greater than the brick size the edit can be culled. All relevant edits set their corresponding bit in the bitmask. If an edit is smooth, then all its dependencies are retrieved from the dependency buffer, and the corresponding bit of each dependency is also set.

The index buffer will then be the indices of all the set bits within the bitfield. A prefix sum can be performed to compact the bitfield, and then the indices of the set bits can be written to the global index buffer. Indices are guaranteed to be both unique and in ascending order.

The effect of edit culling is displayed in Figure 5. It can also be seen how smooth blending can dramatically increase the number of edit evaluations per sample.

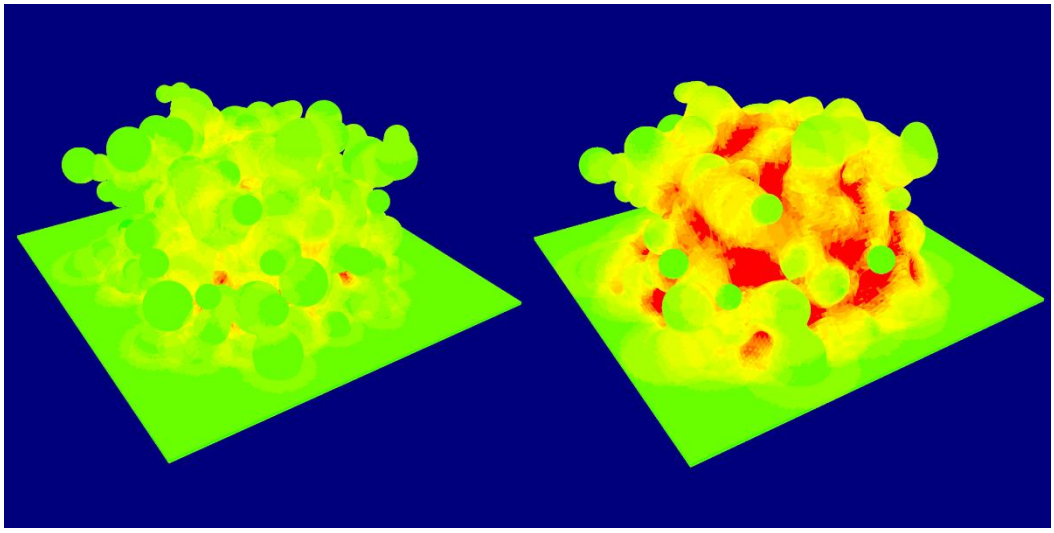

Figure 5: Edit count per voxel for 'Drops' for a blending radius of 0.2 and 0.5 respectively, where green signifies 1 edit and red 16 edits or greater.

3.3.7 Brick Evaluation

The CPU must wait until hierarchical brick construction has completed prior to commencing brick evaluation. This is because the final brick and index counts are required to allocate brick pools and index buffers. This stall could be avoided by allocating a brick pool large enough to contain the worst-case scenario number of bricks. However, accommodating for the worst-case scenario defeats the purpose of attempting to store the brick compactly in a pool in the first place.

As resizing the brick pool is an infrequent occurrence, the stall could also be avoided by simply continuing with brick evaluation and afterwards it could be determined if the brick pool was large enough for brick evaluation to be successful. If not, then a new brick pool can be allocated, and construction restarted. Therefore, the stall will not be present in nearly all constructions, however construction will be executed twice in cases where the brick pool requires resizing.

Once the brick count has been determined and the pool has been allocated, the distance field can be evaluated. To evaluate the distance field, each brick is processed by one thread group. Each thread group consist of 8^3 threads, and each thread will evaluate the edit list once to obtain a single sample.

Evaluating simply involves iterating through the index buffer of the brick and evaluating each edit at the thread's corresponding location in

space. The results from each edit are combined according to the CSG operation of each edit, and the final distance value is stored in the brick pool. The brick pool is formatted to contain 8-bit signed normalized values, which gives a precision of 256 possible values per voxel. Distances stored within the pool are mapped such that the maximum representable magnitude of 1 corresponds to 4 voxels, allowing each voxel to encode a range of [-4,4] voxels. Therefore, surfaces can be represented with a precision of 1/32 of the brick size. A slice from the brick pool of the ‘Drops’ scene is displayed in Figure 6.

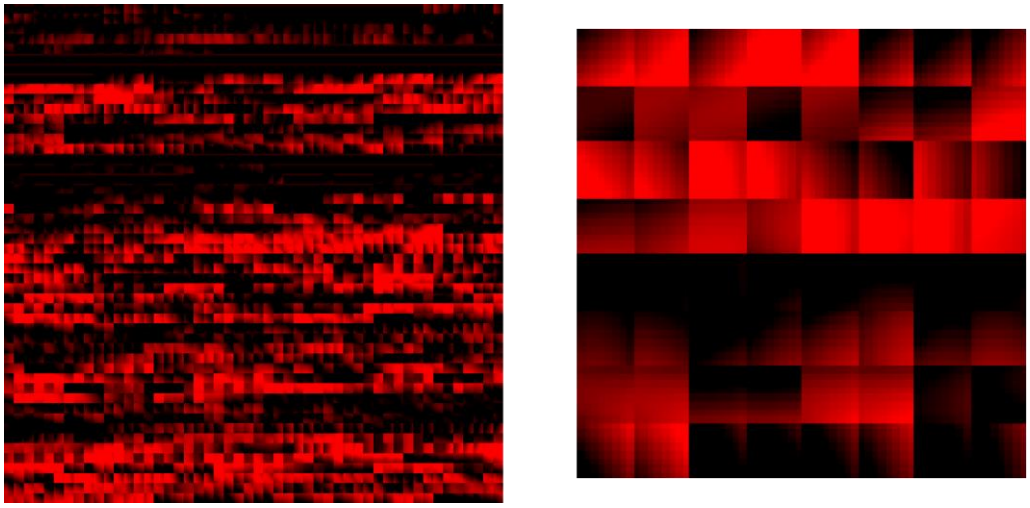

Figure 6: A cross-section from 'Drops' brick pool, with a magnified excerpt on the right.

To speed up evaluation, edits are first loaded into group-shared memory. As the maximum number of 1024 edits would not fit in group-shared memory at once, and since edits are evaluated independently, edits are loaded into group-shared memory in chunks of 256 at a time. Using larger chunks could limit occupancy due to the quantity of group-shared memory required per group, and using smaller chunks could limit the amount of parallelism achieved and result in more threads waiting at barriers for sibling threads. Potential optimization of this value is discussed but thorough evaluation can be considered for future work.

3.5 Testing and Evaluation

Testing was carried out to determine the feasibility of modifiable SDFs in a real-time application. The focus of the testing is to investigate if the implemented data structure and construction algorithm made effective use

of the GPU to allow for an SDF-based representation of geometry that is modifiable in real-time and scalable for the purpose of a game.

3.5.1 Method and Tools

NVIDIA Nsight Perk SDK was used to gather performance metrics. The ranges of GPU work to profile were specified in the application source code, and the application was launched in a profiling configuration to collect the specified metrics from the GPU. The collected data was formatted and output to a CSV file for further processing and analysis in Microsoft Excel. Nsight Perf mitigates the overhead of metric collection by profiling nested in ranges in isolation over multiple passes, such that no profiling operations will be executed within a range that is being actively profiled.

The memory usage of each resource of the SDF object was measured, for a brick size of 0.0625 world-space units. This includes the brick pool, the brick buffer, the index buffer, the raytracing AABB buffer, and the raytracing bottom-level acceleration structure (BLAS). Although the AABB buffer is a transient resource and can be discarded once the acceleration structure is constructed, in the case where objects are reconstructed each frame the AABB buffer will instead be retained as a persistent resource. As such, its memory usage is collected alongside that of the other resources.

Memory usage was measured separately from the performance metrics for rendering and construction. The memory usage was measured upon the first construction of the object, at timestamp t = 0s. This was done as the brick pool will be allocated optimally to provide the tightest fit for the number of bricks in the object, which may not remain true as the object animates over time. Objects were constructed with edit culling enabled.
For texture resources, e.g. the brick pool, memory usage was measured as the total size reported by the function listed in Figure 7.

```
ID3D12Device::GetCopyableFootprints(...);
```

Figure 7: The function used to obtain the memory consumption of the brick pool.

This provides a GPU-agnostic measure of the footprint of a resource. For buffer resources, the memory usage was calculated directly

as the number of elements within the buffer multiplied by the byte stride of each element.

In the context of a game, visual artefacts are unacceptable. Aside from any artistic considerations, the geometry must be technologically sound to produce the desired geometry with no artefacts.

With any discretized data, the resolution should be befitting the rate at which it will be sampled. In this case, this means that the bricks composing an object should be small enough such that even with the limited precision of the brick pool, the distance field produces a visibly continuous surface when sampled at the resolution of the display. It may also be the case that even small brick sizes do not correctly represent the intended geometry accurately. Visual fidelity will be evaluated as how closely the geometry constructed using bricks matches the same geometry rendered analytically.

To evaluate visual fidelity, a simple scene was constructed containing two spheres smooth blending together, referred to simply as the ‘Spheres’ scene henceforth. A ground-truth image of this scene was obtained by rendering this geometry analytically, with no discretization of the distance field. The ground-truth image is shown in Figure 8. The same scene was then constructed from a discretized distance field built from bricks 0.5, 0.25, 0.125, and 0.0625 world-space units in size. Through comparing each brick size to the ground truth, any error in the surface shape and surface normals can be identified.

By constructing the ‘Drops’ scene with and without edit culling, the correct-ness of the edit culling algorithm can also be validated visually. The surface shape and surface normals of the produced geometry should be identical in both scenarios. The scene was rendered at identical timestamps and identical viewpoints with and without edit culling to produce two images. These images are compared to discern inaccuracies in the edit culling algorithm.

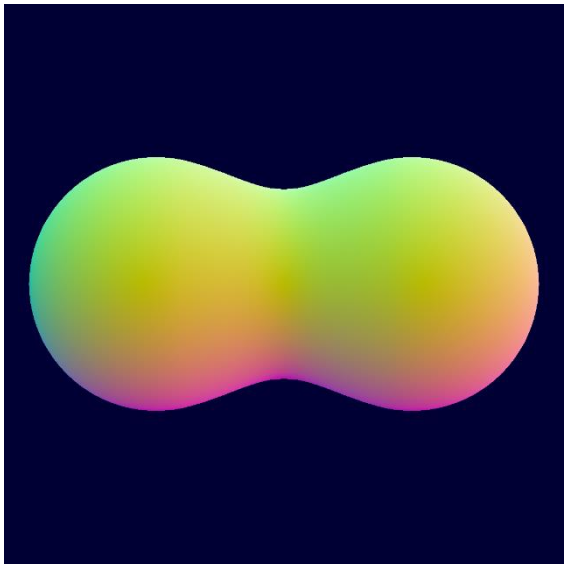

Figure 8: The ground-truth render of the 'Spheres' scene for evaluating fidelity.

3.5.2 Metrics

Different sets of metrics were collected for rendering and for construction so that only relevant data is collected for each. The metrics collected, along with justifications for the relevance of each metric, are displayed in Table 1 and Table 2.

Throughput is defined as the percentage of the maximum achievable rate that data is processed for a specific hardware unit or collection of units. Only metrics for the ALU and LSU hardware units are included in the data as it was found that other the hardware units had negligible usage by comparison and were not the bottleneck in any circumstance.

Metric	**Reasoning**
Duration (ns)	To determine the latency of an operation or workload.
Raytracing Core Throughput (%)	To determine to what degree the raytracing hardware is exploited.
SM Throughput (%)	To give a general sense of how close performance is to the theoretical maximum.
L2 Cache Hit Rate (%)	To determine the application is using cache coherent data access patterns.

Table 1: Metrics collected for rendering.

Metric	Reasoning
Duration (ns)	To determine the latency of an operation or workload.
CS Warp Activity (%)	To measure occupancy – how well the algorithm is managing to maximise the use of GPU resources.
SM Throughput (%)	To give a general sense of how close performance is to the theoretical maximum.
L2 Cache Hit Rate (%)	To determine the application is using cache coherent data access patterns.
ALU & LSU Throughput (%)	To determine which hardware unit is seeing the most significant use.
CS Warp Launch Stall Reason (%)	To determine what hardware resources are in high contention.

Table 2: Metrics collected for construction.

3.5.3 Test Scenarios

Both construction and rendering were profiled on 4 different scenes named Drops, Cubes, Rain, and Fractal. Screenshots of these scenes are displayed in Figure 9. Each scene contained one object which had 1024, 216, 512, and 64 edits respectively. These scenes were designed to exhibit a variety of edit counts, edit density, and edit complexity. For example, Fractal only contains 64 edits – but these edits are individually expensive to evaluate. Drops contains 1024 edits that frequently intersect with a moderate amount of blending. Rain contains a variety of edit sizes and densities. Finally, Cubes contains sparse structures, which is traditionally a poorly performing scenario for sphere tracing.

Tests were performed on each scene for brick sizes of 0.5, 0.25, 0.125, and 0.0625 world-space units in size – where a smaller brick size results in a larger number of bricks produced. Each scene was tested with edit culling enabled and disabled. Therefore, the application was tested under 32 different scenarios in total to investigate the effectiveness of the data structure and construction algorithm.

50 captures were taken for each brick size for each scene, and mean values were calculated. For profiling rendering, one capture is defined as a single time the scene was rendered via raytracing. For profiling construction, one capture is defined as a single time an object was constructed.

The scenes were rendered at a resolution of 3840px by 2160px. An orbiting viewpoint centred on the object was used to remove any bias each object may have for being rendered from a certain direction. Each scene was rendered at a timestamp of t = 5 seconds for every capture, allowing some warm-up time for each scene.

The final image simply displayed the world-space normal calculated by each ray, and therefore rendering time does not include time spent performing lighting calculations and shading. Only primary rays were used, with a single ray cast per pixel.

Rendering and construction were profiled in separate sessions as they are performed by separate GPU queues. Synchronous compute was used in all tests to ensure the GPU was not splitting its resources between work items.

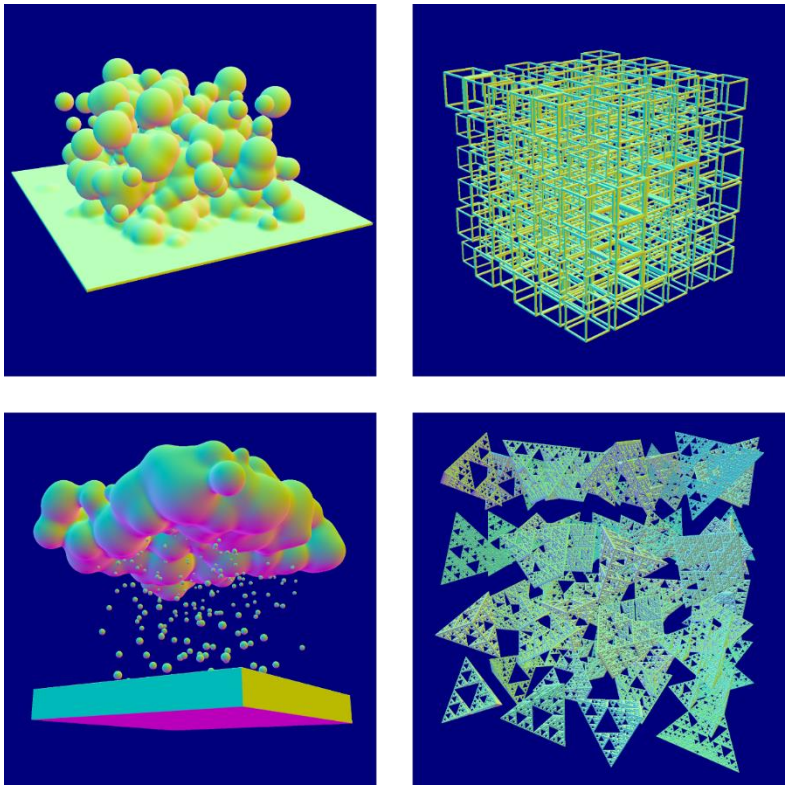

Figure 9: The test scenes (in row-major order): Drops, Cubes, Rain, and Fractal.

Chapter 4 Results

All tests were performed on an Intel 12th Gen Core i7-12700 CPU with 64GB of system RAM, with an NVIDIA GeForce RTX 3070 GPU with 8GB GDDR6 dedicated VRAM using driver version 551.86.

Table 3 displays the number of edits composing each scene to provide context to the statistics displayed in the figures below. When the scenes were constructed with and without edit culling, different brick counts were obtained. These sets of brick counts displayed in Table 4 and Table 5 for edit culling enabled and disabled respectively.

In all figures, both brick sizes and brick counts are plotted with logarithmic axes for clarity as they increase exponentially.

Demo	**Edit Count**
Cubes	*216*
Drops	*1024*
Fractal	*64*
Rain	*512*

Table 3: The number of edits composing each scene.

Demo	**Brick Size** (World-space units)			
	0.0625	0.125	0.25	0.5
Cubes	*176,422*	*53,620*	*12,491*	*2,693*
Drops	*362,731*	*79,248*	*19,083*	*4,043*
Fractal	*444,899*	*107,653*	*27,107*	*7,111*
Rain	*234,863*	*59,877*	*15,808*	*4,144*

Table 4: The number of bricks composing each scene when edit culling is enabled, for each brick size.

Demo	**Brick Size** (World-space units)			
	0.0625	0.125	0.25	0.5
Cubes	*183,491*	*55,765*	*12,406*	*2,693*
Drops	*376,973*	*82,439*	*19,547*	*4,102*
Fractal	*444,894*	*107,750*	*27,107*	*7,201*
Rain	*238,107*	*60,666*	*15,934*	*4,160*

Table 5: The number of bricks composing each scene when edit culling is disabled, for each brick size.

4.1 Rendering

The mean time taken to perform raytracing of each scene for each brick size and each brick count are presented in Figure 10 and Figure 11 respectively. Each scene was constructed with edit culling enabled.

Figure 12 presents the mean Raytracing Core throughput for each scene and brick size. Constructed with edit culling enabled. Raytracing Core refers to the hardware-accelerated raytracing units on the GPU.

The L2 Cache hit rate during rendering each scene at each brick size is displayed in Figure 13. This is the percentage of VRAM accesses that resulted in a cache hit from the L2 cache out of all VRAM accesses.

A comparison between raytracing times of each scene constructed with edit culling enabled and disabled at a brick size of 0.0625 world-space units is displayed in Figure 14.

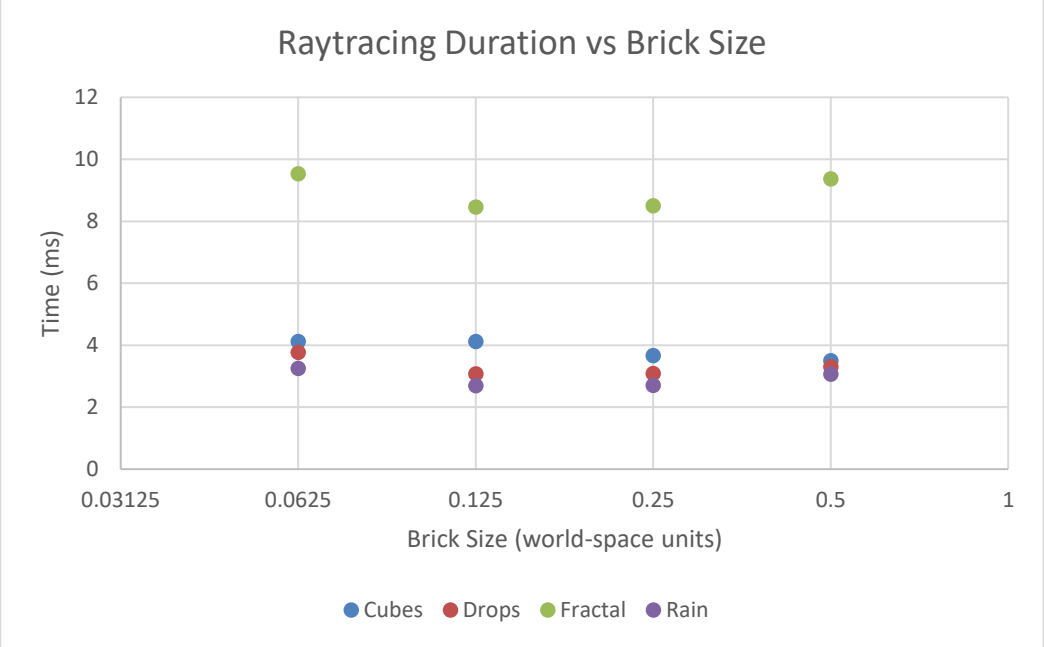

Figure 10: Average time taken to raytrace each scene for each brick size.

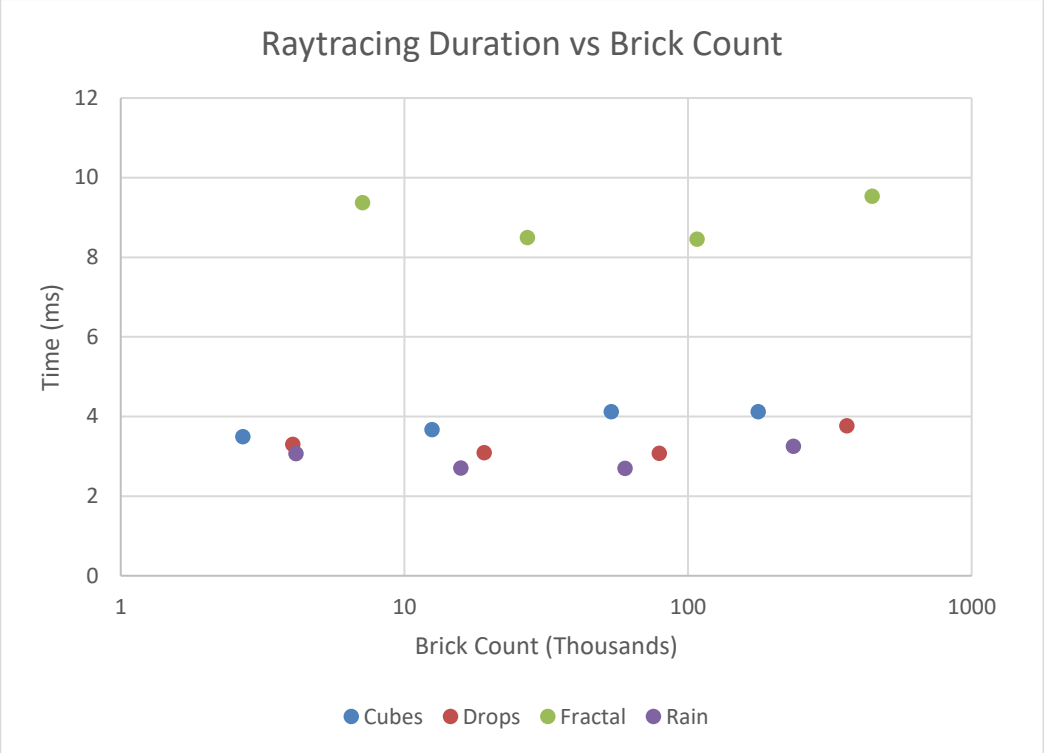

Figure 11: Average time taken to raytrace each scene as brick count increases.

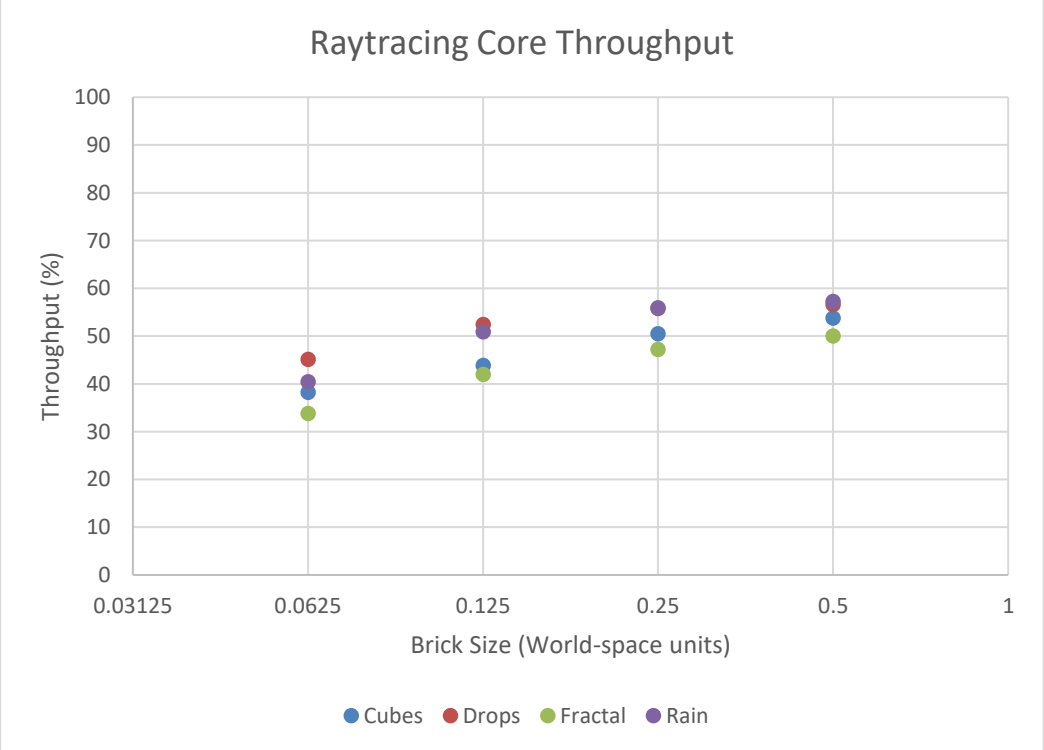

Figure 12: Average Raytracing Core Throughput for each brick size for each scene.

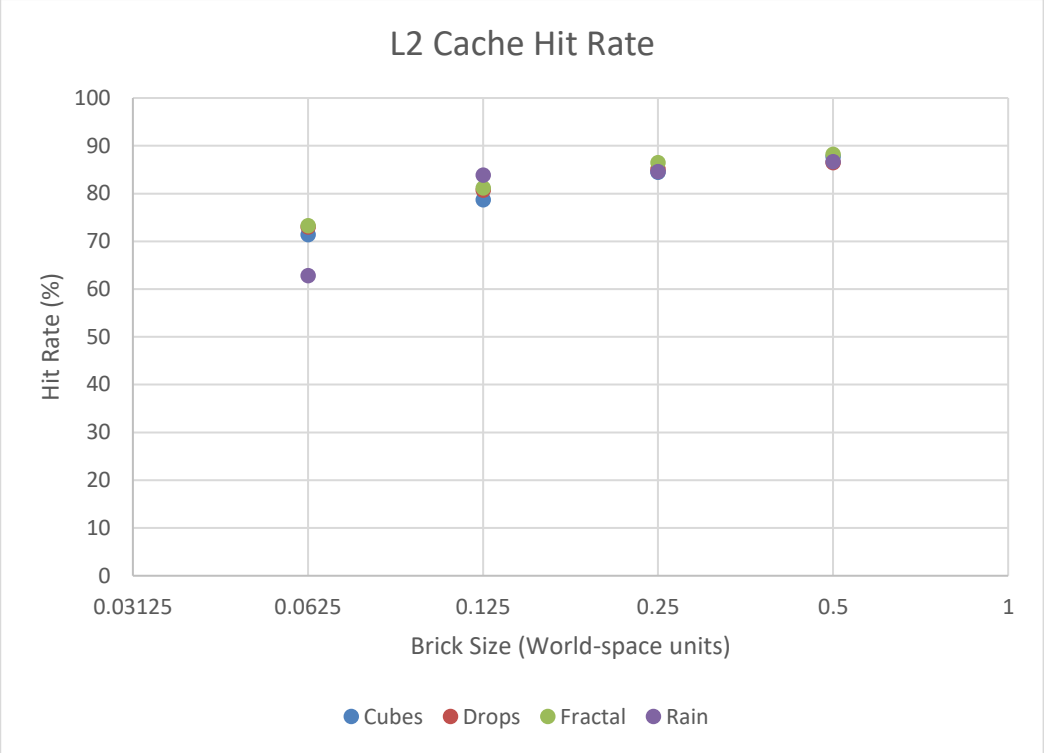

Figure 13: Average L2 cache hit rate for each scene for each brick size.

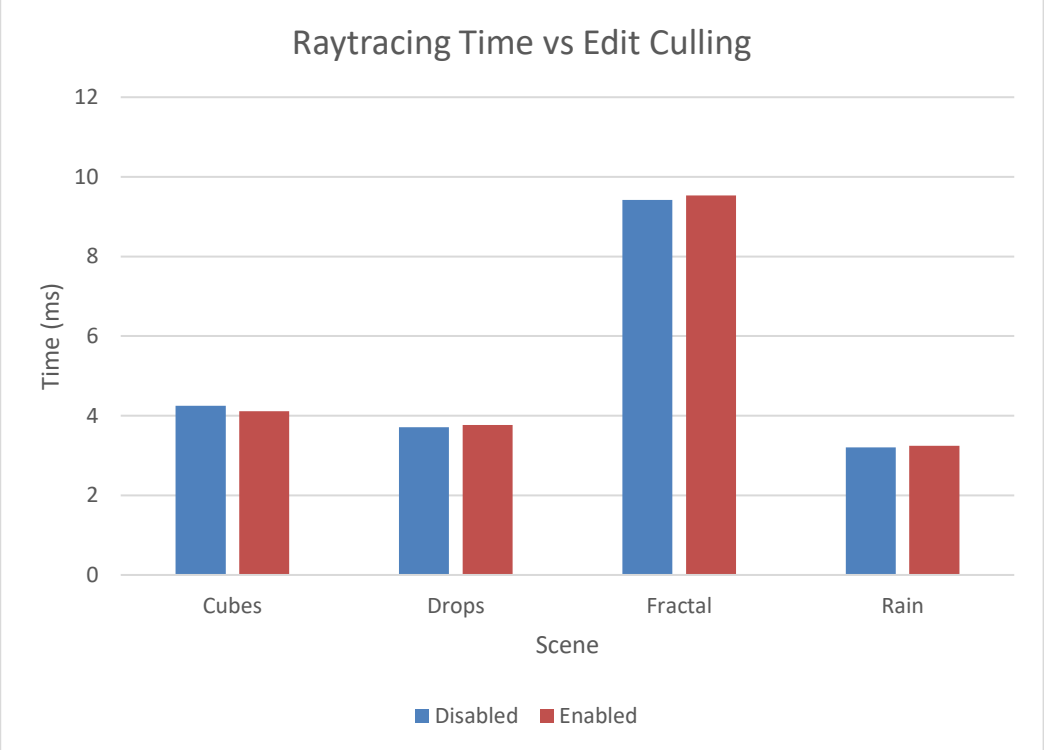

Figure 14: Average raytracing time for each scene with a brick size of 0.0625 with edit culling enabled and disabled.

4.2 Construction

The mean construction time for each scene for each brick size, with edit culling enabled, is displayed in Figure 15. The mean construction time of the same scenarios, but with edit culling disabled, is displayed in Figure 16. Power trendlines are displayed on both figures to show the trend as the brick size increases. A direct comparison between construction times with edit culling enabled and disabled is shown in Figure 17.

Figure 18 displays the proportion of total construction time consumed by each stage, to establish an idea of the dominant stages in construction. The contribution from evaluating edit dependencies and AABB building is negligible in proportion and consequently these stages are not easily discernible in the figure. The data shown in this figure was taken from the Drops scene but found to be similar across the other scenes.

The mean Streaming Multiprocessor (SM) throughput for each scene for each brick size with edit culling enabled and disabled is displayed in Figure 19 and Figure 20 respectively. A streaming multiprocessor refers to a group of computational units encapsulating all individual hardware units; therefore, SM throughput is a general measure of instruction throughput.

Figure 21 displays the mean occupancy for each stage in construction, with edit culling enabled. Occupancy is defined to be the percentage of warp slots across all SM's actively executing instructions out of all total warp slots available on the GPU. This data was taken from the Drops scene.

Mean hardware unit usage during brick evaluation for each scene for each brick size with edit culling enabled is displayed in Figure 22, to illustrate the effect that the properties of the edits can have on the hardware usage during construction of a scene. The graph is divided in two halves, with the left-hand side displaying LSU throughput, and the right-hand side displaying ALU throughput.

Figure 23 summarizes the reasons behind stalls in warp launches, where a stall occurs when the schedular must wait for some resource to become available before launching new warps. In general, the faster warps

can be launched the greater the overall throughput of the GPU will be. Warp launches may be stalled due to limited registers, limited CTA (Co-operative Thread Array, NVIDIA terminology for a group) slots, limited warp slots, limited group-shared memory, or limited barriers. Limited barriers were not an issue at any point and therefore not shown in the figure. The data plotted was taken from constructions with edit culling enabled.

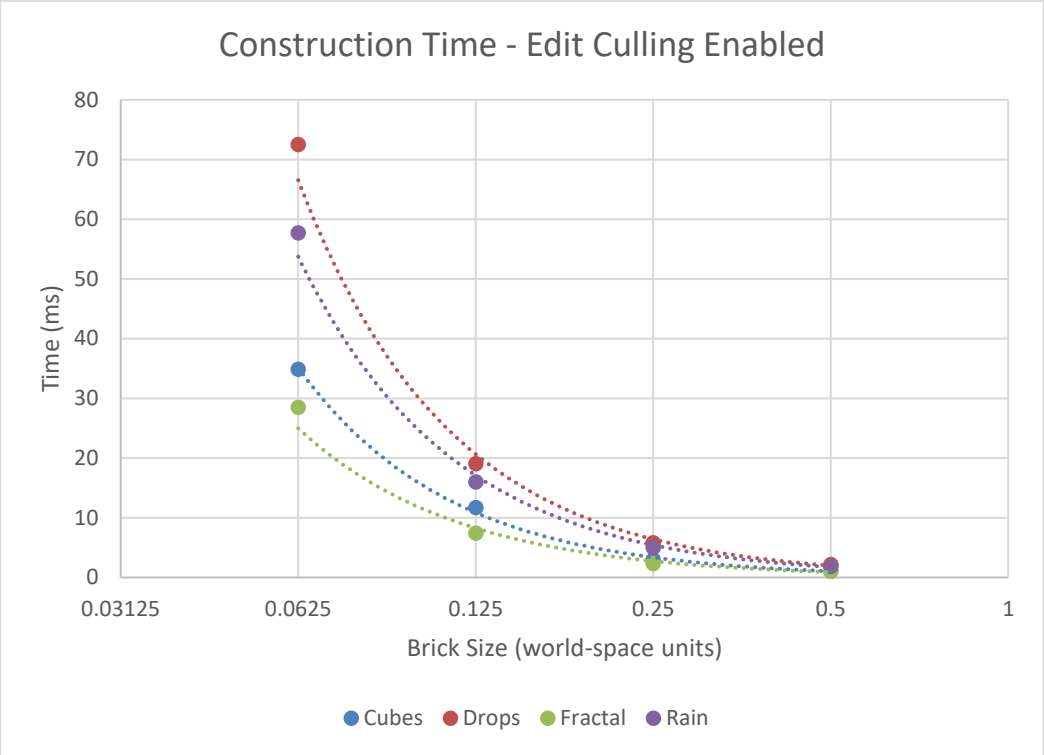

Figure 15: Construction time for each scene for each brick size with edit culling enabled.

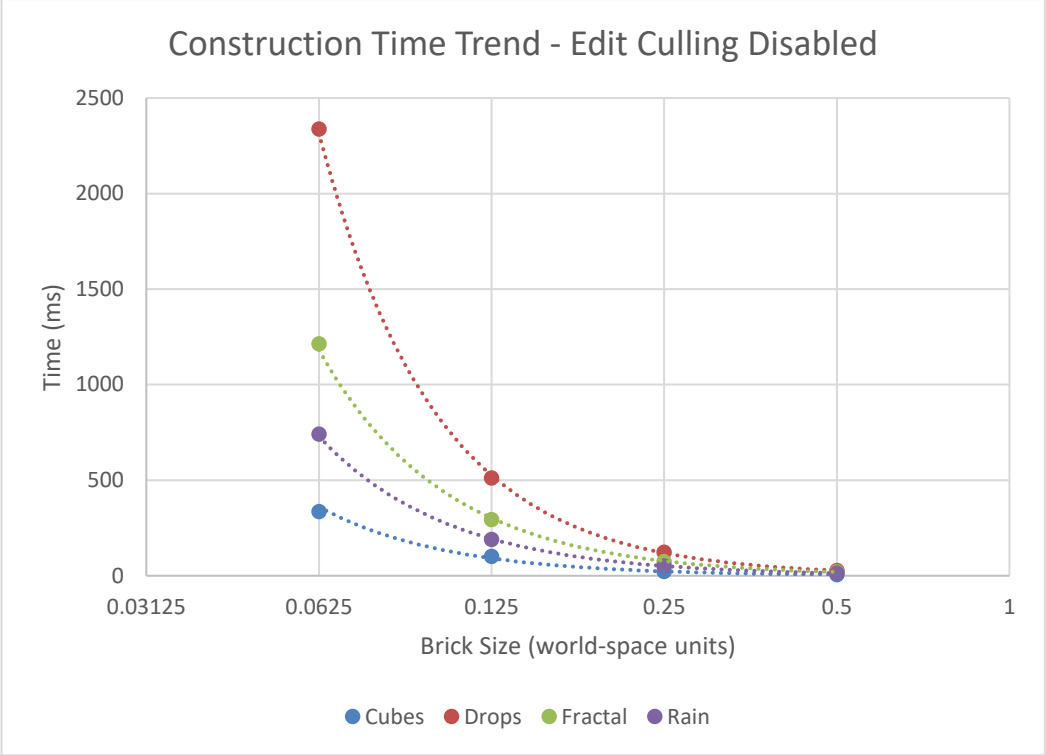

Figure 16: Construction time for each scene for each brick size with edit culling disabled.

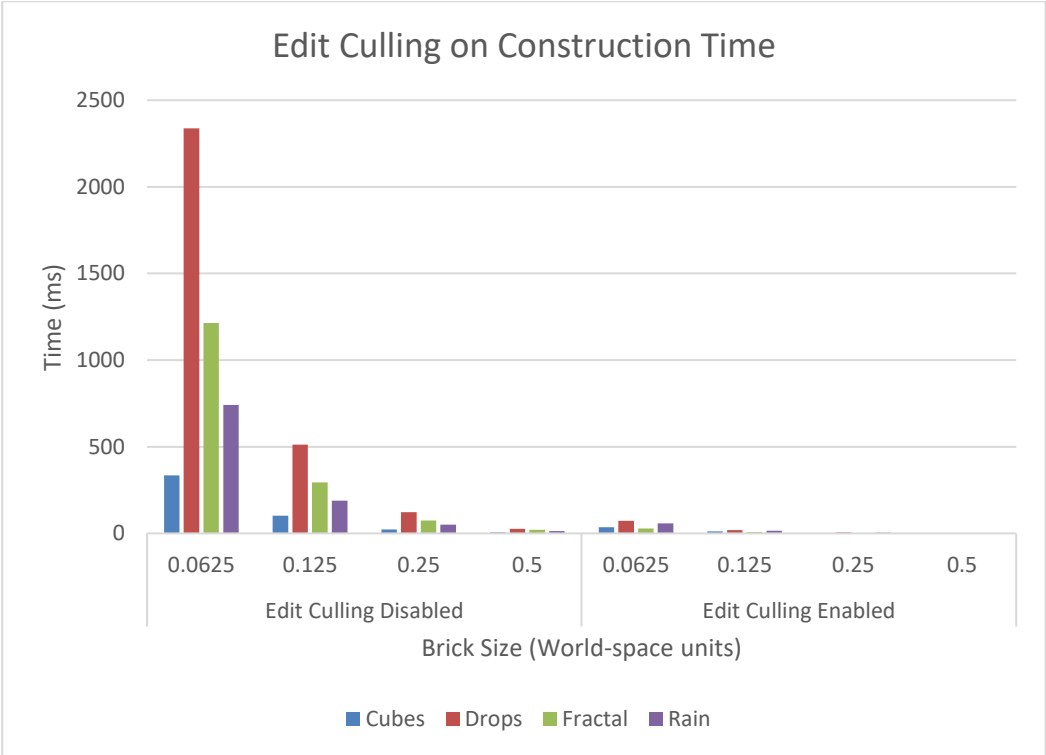

Figure 17: A direct comparison between construction times with edit culling enabled and disabled.

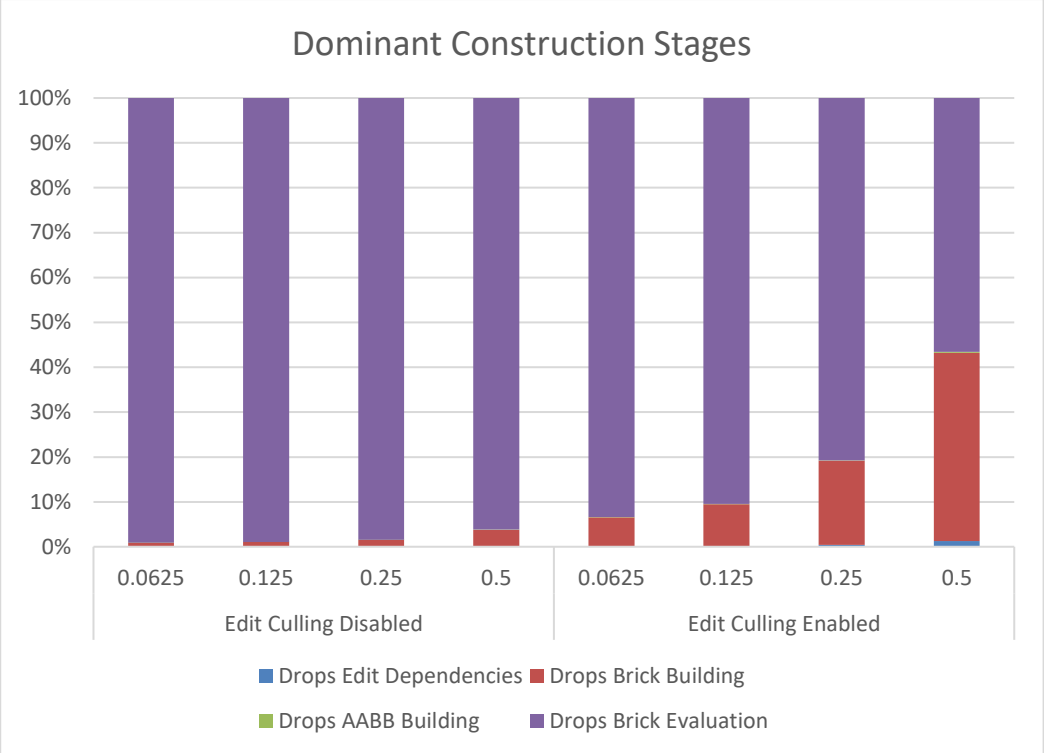

Figure 18: The time taken by each construction stage, as a percentage of total construction time.

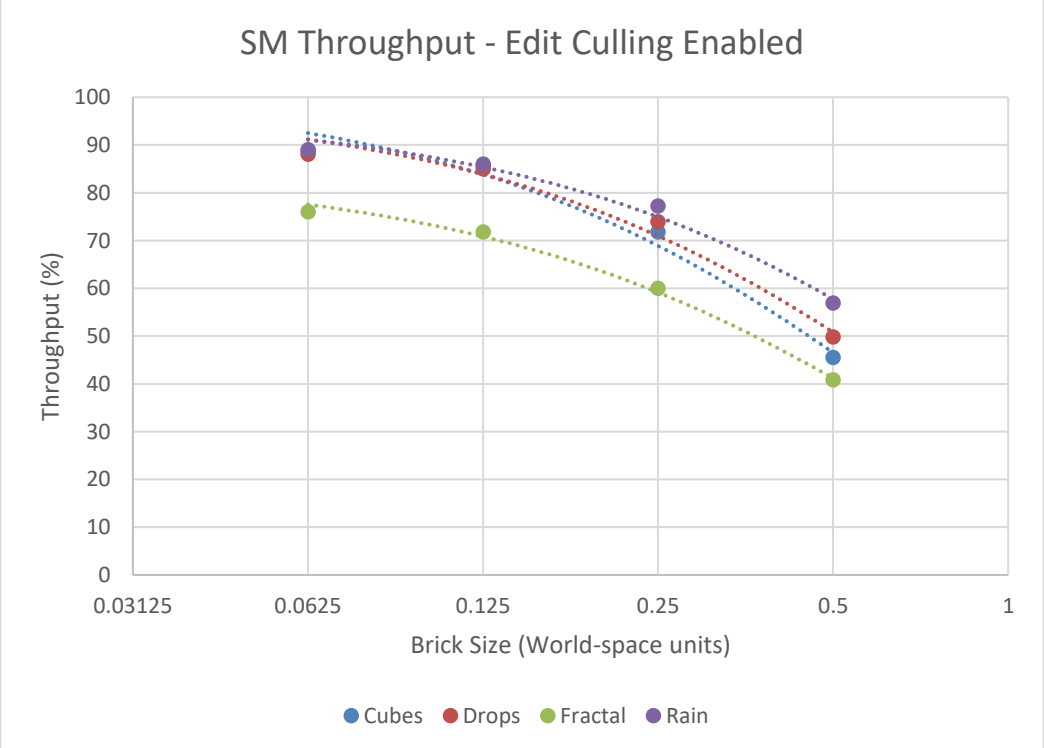

Figure 19: SM throughput as brick size increases for each scene, with edit culling enabled.

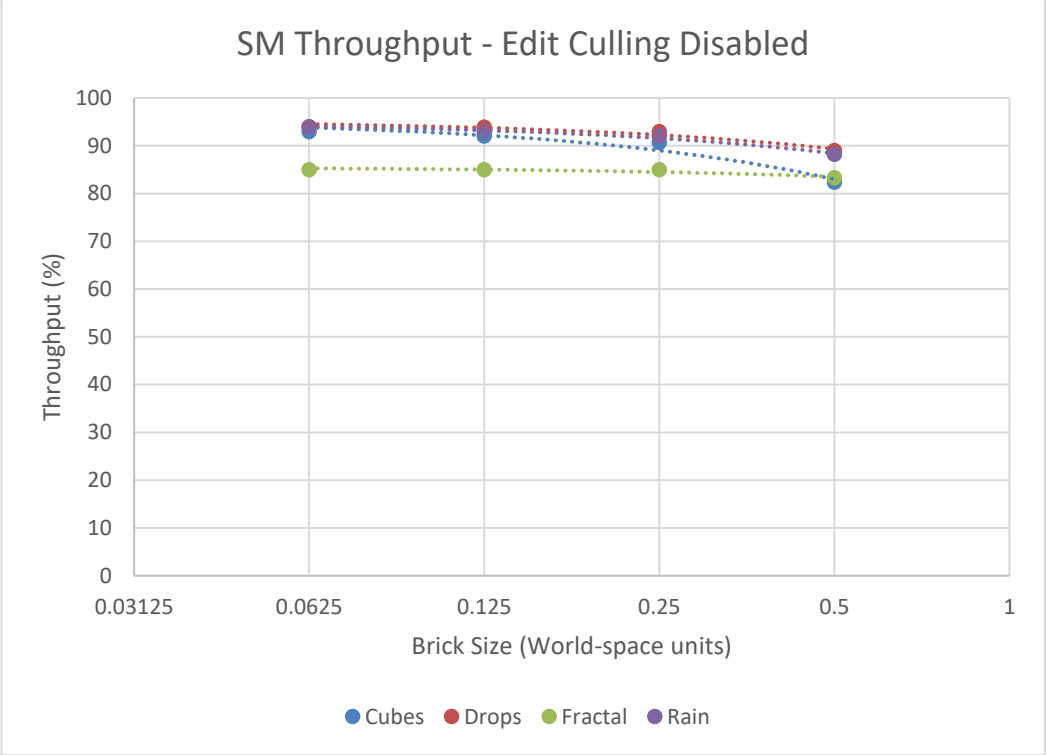

Figure 20: SM Throughput for each scene and each brick size, with edit culling disabled.

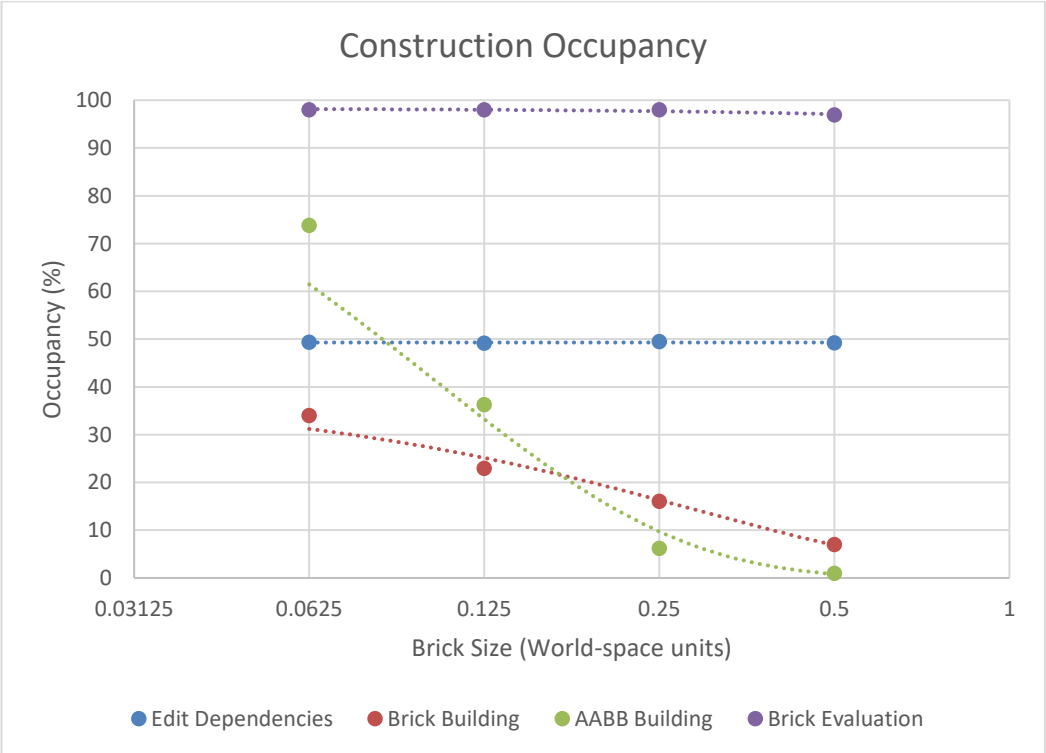

Figure 21: GPU Occupancy for each construction stage as brick size increases.

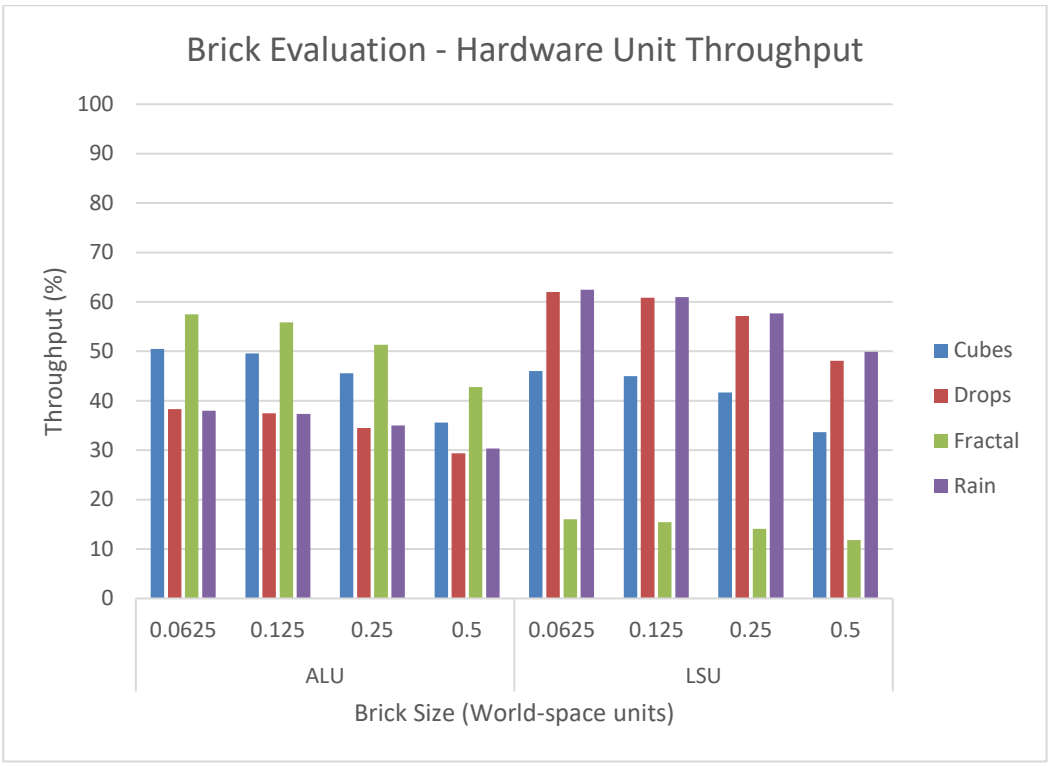

Figure 22: Unit throughput during brick evaluation for the ALU and LSU for each scene and brick size.

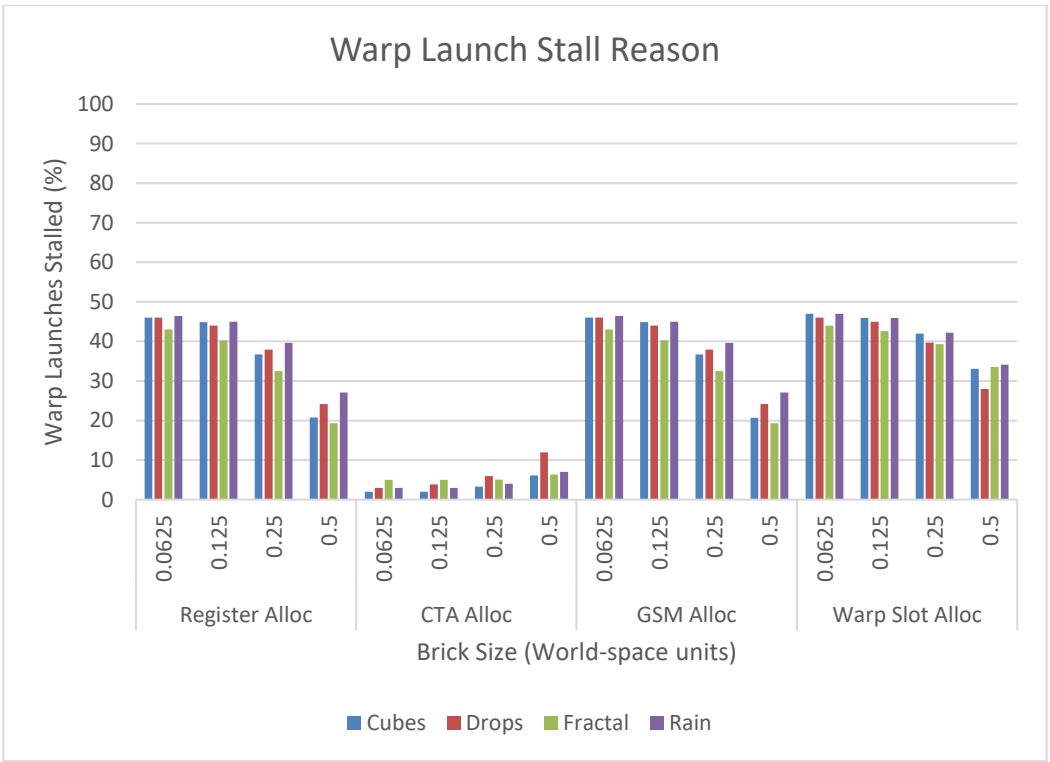

Figure 23: Warp Launch Stall Reason for each scene and brick size.

4.3 Memory

The memory usage of each resource within an SDF object is displayed in Table 6. The brick counts are also displayed for context.

Figure 24 shows the memory usage for each resource for each scene. Memory usage is displayed in megabytes with a logarithmic scale. Another visualization of the same data is shown in Figure 25, which illustrates the proportion of the total memory consumption of each object that each resource consumes.

Figure 26 plots memory usage, in megabytes, against the brick count of the object. A linear trendline is also shown.

	Cubes	**Drops**	**Fractal**	**Rain**
Brick Count	*185,862*	*469,619*	*444,955*	*231,256*
Brick Pool (MB)	*210*	*475*	*463*	*240*
Brick Buffer (MB)	*10*	*25*	*24*	*12*
AABB Buffer (MB)	*9*	*21*	*20*	*11*
Index Buffer (MB)	*15*	*44*	*2*	*30*
BLAS (MB)	*21*	*54*	*51*	*27*
Total (MB)	*265*	*620*	*560*	*320*

Table 6: Memory usage in MB of each resource for each scene.

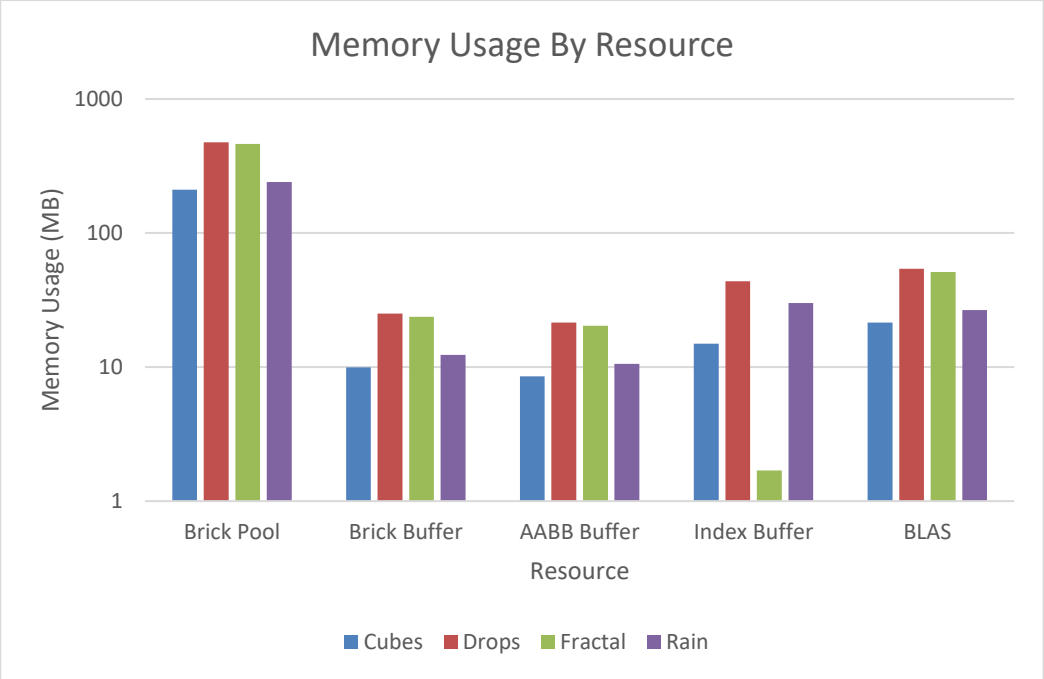

Figure 24: Memory usage for each resource for each scene in MB.

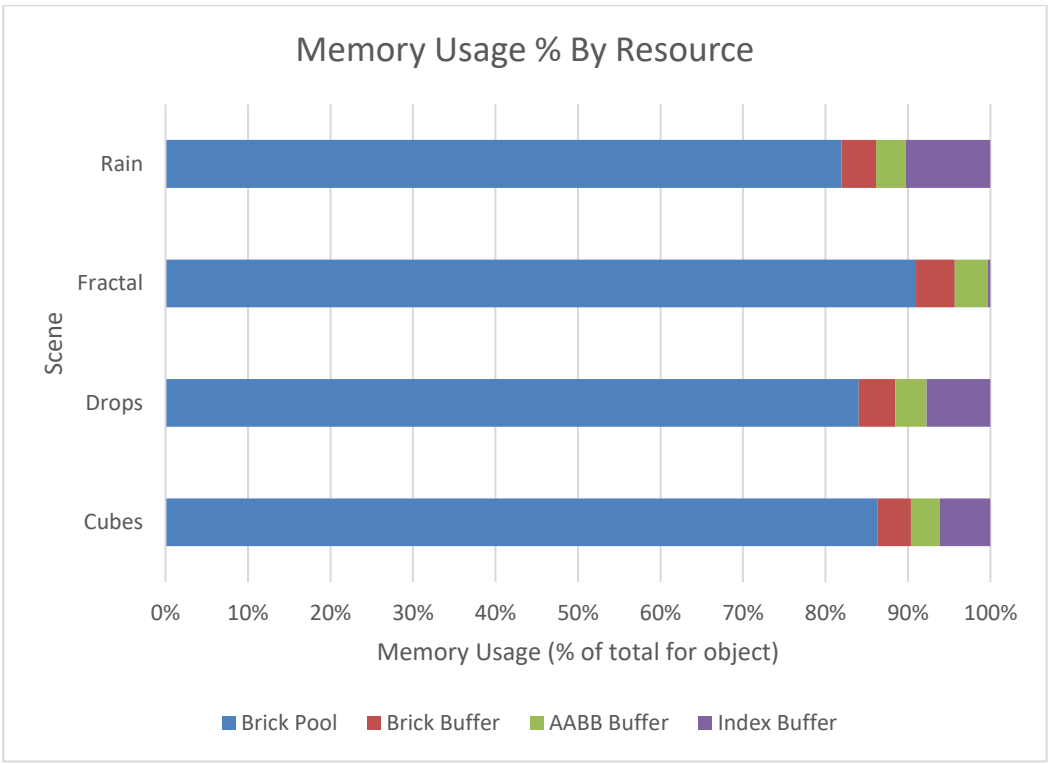

Figure 25: Resource memory usage, displayed as a percentage of the object's total memory usage.

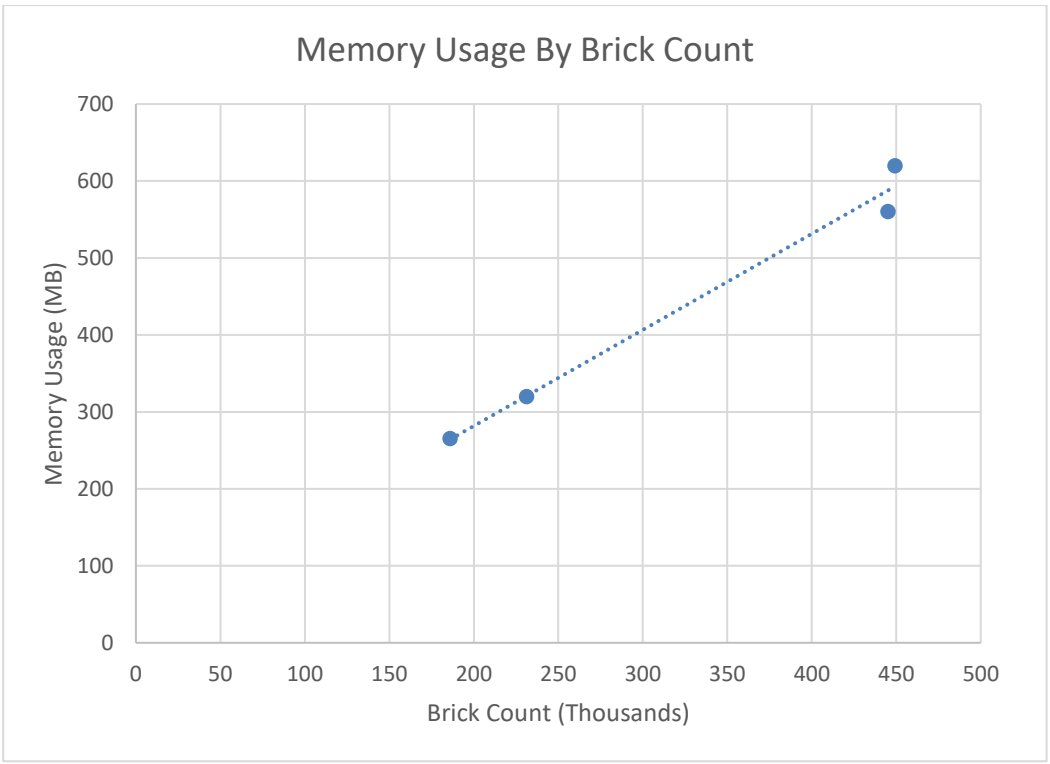

Figure 26: Memory usage in MB by brick count.

4.4. Visual Fidelity

The Spheres scene was constructed from brick sizes 0.5, 0.25, 0.125, and 0.0625, and rendered from the same viewpoint. The results of each render are displayed in Figure 27.

Each was then compared to the ground-truth render of the Spheres scene, which was obtained by rendering an analytic representation of the scene. The ground-truth render is displayed in Figure 8.

The difference between the scene rendered discretely and the ground truth for each brick size is shown in Figure 28. In Figure 29, the differences are greatly exaggerated to demonstrate for clarity, where the noise in the surface normals can be seen. An example of how this noise can affect an object once a shading model has been applied is shown in Figure 30.

The correctness of the edit culling algorithm is also tested visually, where the Drops scene was rendered with and without edit culling from an identical viewpoint. These renders are displayed in Figure 31.

The error in the construction algorithm is therefore the difference between these two renders, which is displayed in Figure 32, although the differences are too small to be perceptible. The differences were exaggerated and magnified by image processing software to produce Figure 33, where small error in the edit culling algorithm can be identified.

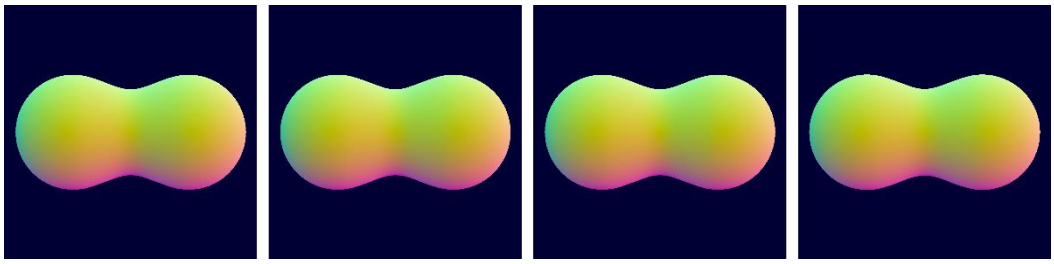

Figure 27: The fidelity test scene rendered at brick sizes (left-to-right) 0.5, 0.25, 0.125, 0.0625.

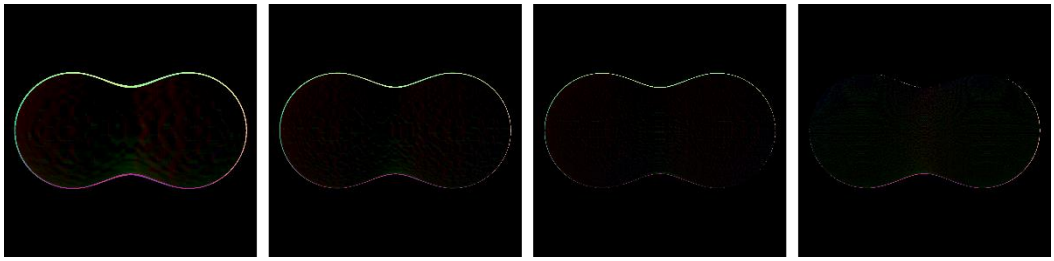

Figure 28: The difference between each fidelity test and the ground truth for brick sizes (left-to-right) 0.5, 0.25, 0.125, 0.0625.

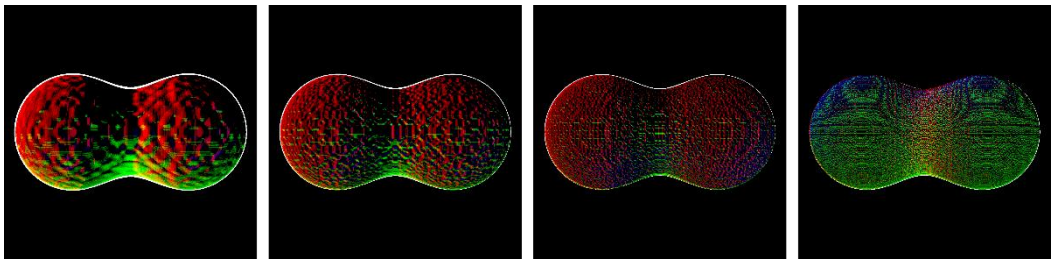

Figure 29: Figure 28 with highly exaggerated colours for clarity.

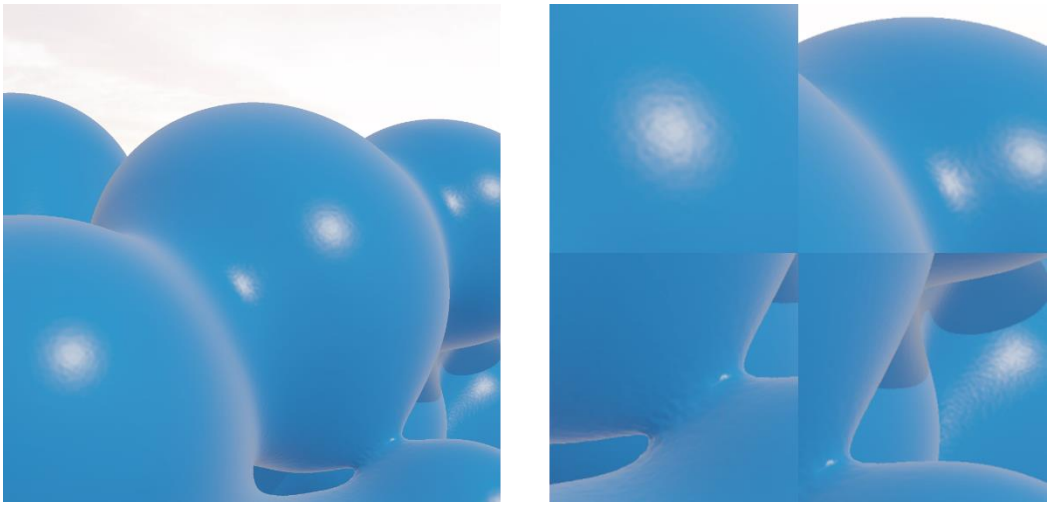

Figure 30: Noise can be seen in the surface normals. A series of magnifications are displayed on the right.

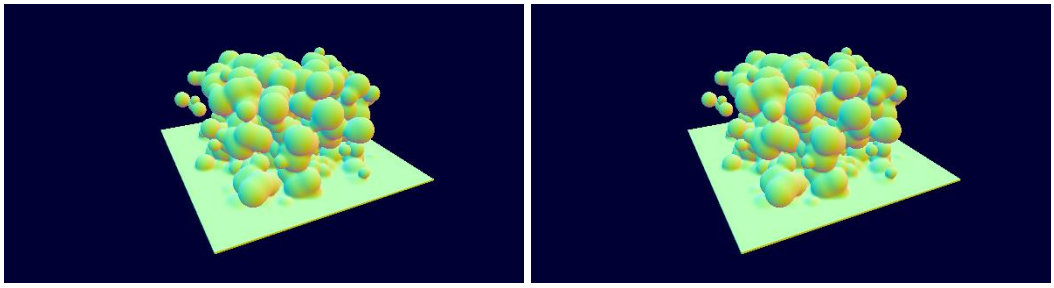

Figure 31: The Drops scene rendered with edit culling (left) and without edit culling (right).

Figure 32: Error in the edit culling algorithm. The differences are imperceptible when not exaggerated artificially.

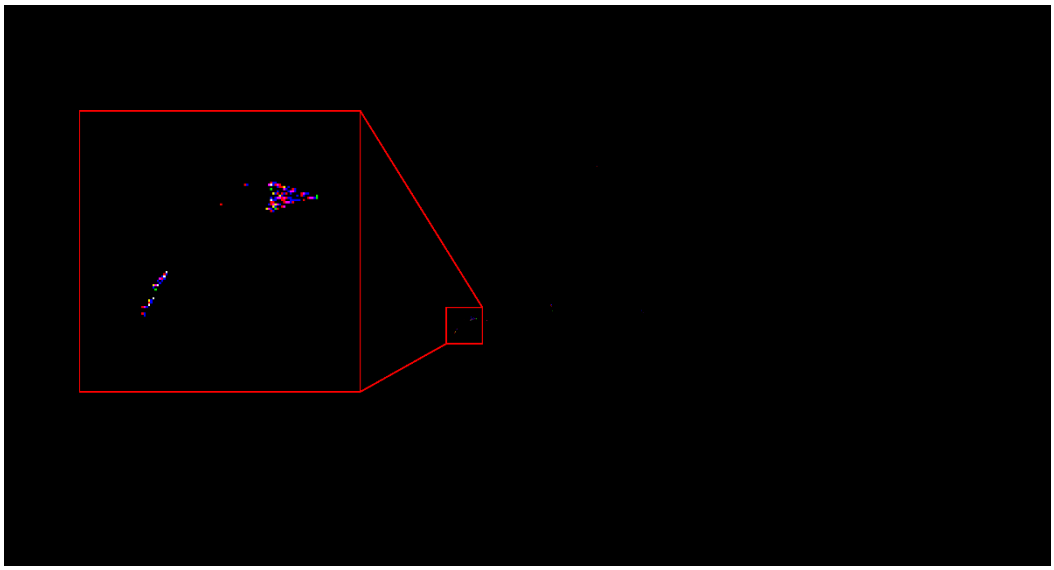

Figure 33: Error in the edit culling algorithm, greatly exaggerated and magnified to be perceptible.

Chapter 5 Discussion

5.1 Results Analysis

5.1.1. Rendering Performance

It can be noted from Figure 10 and Figure 11 that the Fractal scene takes considerably longer to ray-trace than all other test scenes at every brick size. There are several contributing factors toward this that illustrate the properties and challenges of both SDF rendering and raytracing in general. It is shown in Table 3 that Fractal contains significantly more bricks than the other scenes, although it will be discussed that this is not of significant impact to the rendering performance. However, the type of geometry in the Fractal scene is a typical poor case for sphere-tracing, where it contains fine structures with gaps between geometry. This requires a greater amount of sphere-tracing, as rays are more likely to intersect multiple AABBs as they skim between the geometry.

Furthermore, as rays originate from the camera position and travel in the camera's point of view, raytracing is a view-dependent process. The more rays that encounter geometry, the slower raytracing will be. In the case of Fractal, the edits are large and consume a larger portion of the screen than other scenes, and therefore more rays are likely to intersect. This view-dependence is a source of bias that is not accounted for in this study. It could be accounted for by measuring the total number of pixels that are covered by geometry, and either weighting performance figures accordingly to obtain an average performance per pixel, or by adjusting scenes such that each scene has the same amount of screen coverage. While the screen coverage is not consistent between scenes, it is consistent between brick sizes so meaningful conclusions can still be drawn about the rendering performance of this technology.

Either way, it is still important to acknowledge that the time taken to raytrace the scene will increase as objects become closer and consequently consume more screen space. This can be both an advantage and disadvantage in a game scenario – it is appropriate for more rendering time to be dedicated towards nearby objects than distant objects. However,

in this case there is no gain in quality for the increased rendering time due to the fixed resolution of the volume.

What is curious about the results is that raytracing time doesn't increase with brick count, as displayed by Figure 11. Indeed, the brick size that resulted in the fastest ray-tracing times of Fractal was the smallest of 0.0625. It could be expected that a greater brick count would result in slower acceleration structure traversal as there are simply more nodes through which to traverse, but this does not seem to be the case. One potential explanation could be that larger bricks will result in more empty space within a brick, which means more sphere-tracing iterations are required to reach the surface, and this negates any performance gains from faster BVH traversal. However, the distances encoded within the bricks are relative to the size of the brick and as such larger bricks will result in a larger world-space distance being travelled with every sphere-tracing iteration.

The reason is likely to be that roughly the same amount of screen space is covered by bricks for all brick sizes, and therefore similar number of rays intersect with bricks in the first place. Sphere-tracing in the intersection shader is much more expensive than hardware BVH traversal and is therefore the dominant factor affecting raytracing performance – so increasing the number of AABB's in the BVH does not significantly increase the time taken to traverse the acceleration structure, while the amount of sphere-tracing being performed is remaining relatively constant. Therefore, it can be suggested that rendering the same object from the same viewpoint will take similar times regardless of the brick size.

This is supported by Figure 12, which displays that raytracing core throughput decreases with brick size. This suggests that more work is being performed by the hardware raytracing units at smaller brick sizes, even though similar frame times are achieved. This demonstrates that the amount of time saved by faster BVH traversal at larger brick sizes is then lost through having to do more sphere-tracing, as larger bricks are worse approximations of the underlying SDF isosurface. This result is intuitive – the two scenes that resemble the worst-case scenario for sphere-tracing,

Fractal and Cubes, show the lowest RT Core throughput. From this result, it can be suggested that more time is spent in the software intersection shader, and this leads to the raytracing hardware to be leveraged to a lesser extent.

As mentioned in the literature review, (Evans, 2015) stated that rasterizing the bounding boxes of the bricks permitted the removal of any indirections in the tight ray-marching loop, which helped to improve cache coherence. It was anticipated that a similar result would be found in this artefact, as building a unique raytracing AABB for each brick similarly permits ray marching without indirection in the inner-most loop. However, it can be observed from Figure 13 that the cache hit rate decreases as the brick size decreases. It is speculated that this is due to higher contention for the cache when there are more bricks. This could potentially be improved by ordering the bricks within the brick pool along a Morton curve. While the bricks in the buffer are in order of their Morton codes, the respective order of the bricks in the pool is not. This would be a worthwhile optimization.

The difference in rendering times for objects constructed with and without edit culling is miniscule. This is demonstrated in Figure 14. This suggests that rendering time does not appear to correlate with the number of edits from which an object is formed. This is as expected, and indeed the main motivation behind discretizing the distance field in the first place.

5.1.2. Construction Performance

Figure 15 demonstrates that construction performs at a time complexity of $O(2^n)$. This is to be expected; with each iteration of brick-building, the brick count can multiply by up to a factor of 64. Quartering the brick size will requires another iteration of brick-building, and the brick count will increase exponentially. This makes selection of brick size critical for objects that will be reconstructed frequently. Bricks that are too large will produce geometry of too low a resolution and result in under-sampling during rendering. However, choosing bricks that are too small, such that the distance field precision is less than a single pixel, results in unnecessary work being

performed during construction for no gain in fidelity. The brick size that exactly matches the precision of the distance field to the screen resolution would be ideal.

Perhaps the most significant finding of this research is the effectiveness of the edit culling algorithm, demonstrated by Figure 17. At a brick size of 0.0625, the Drops scene takes multiple seconds to construct without edit culling. This is significantly too slow for an interactive application. However, enabling edit culling increased performance approximately by a factor of 32, with construction times of less than 100 milliseconds. As Drops contains the greatest number of edits, it sees the most benefit from edit culling. Nevertheless, all scenes saw significant gains in performance by enabling edit culling. These findings suggest that real-time reconstruction of SDF geometry would not be feasible without edit culling, and that developing the edit culling technology further would increase the suitability of this geometry representation to real-time interactive applications.

The difference in the trend displayed in Figure 15 and Figure 16 is also critical – both cases still exhibit a time complexity of $O(2^n)$, but it can be seen that the construction time increases at a much slower rate when edit culling is enabled. It can further be noted that the scalability of the algorithm differs for each scene. Drops can be determined to be the least scalable as the brick size decreases, as it exhibits the steepest curve. Conversely, Fractal appears to scale better as brick size decreases. In fact, the rate of change of construction time with respect to the brick size is precisely in order of which scenes contain the most edits. It seems that in general, therefore, the more edits used to construct a scene, the less well the construction of that scene will scale as the brick size decreases. This is particularly important when considering how this technology will scale and can advise the selection of an appropriate brick size.

It would be interesting to determine the scalability of constructing objects with edits counts beyond 1024. 1024 is too low a limit to be able to construct interesting game-quality assets. For example, assets in *Dreams* (Evans, 2015) commonly consisted of tens of thousands of edits.

It can be seen in Figure 18 that the brick evaluation duration grows at a much faster rate than the brick building duration. Edit dependency calculation and AABB construction are negligible by comparison. A possible explanation for this is that brick evaluation will generally processes a significantly greater number of bricks than brick building, up to a factor of 64 times more. It can thus be suggested that brick evaluation is the critical stage in construction in terms of latency. This could be combatted through the development of more precise edit culling solutions. It can also be seen from Figure 18 that the proportion of time spent evaluating bricks is significantly higher when edit culling is disabled. It can be suggested from this that edit culling is effectively reducing the total amount of work to be performed.

Edit culling in this artefact relies on approximately detecting intersections using bounding spheres, which in many cases can result in many more edits being evaluated per brick than in an optimal solution with precise signed distance function intersection calculation. While improved edit culling would decrease the time spent evaluating bricks, the number of bricks will still grow exponentially as the brick size decreases.

Figure 19 shows that the Streaming Multiprocessor throughput, particularly for brick building, is extremely poor for large brick sizes. As discussed in section 3.3.1, early iterations of brick building will never be able to produce enough work to fill the GPU. This is further evidenced by the results shown in Figure 21, where brick building occupancy is particularly low at larger brick sizes. Low occupancy in tree building algorithms is the motivation for solutions like (Karras, 2012), where all levels of the tree can be constructed simultaneously to fully exploit the parallelism of the GPU. Unfortunately, this method cannot be used in this case as evaluating next level of bricks depends on the preceding level. Evaluating the leaf bricks first would require the entire edit list to be evaluated at every candidate space. The iterative space and edit culling provided through top-down construction would not be able to be used, and the resulting workload would not be feasible for real-time construction.

However, as shown by Figure 18 brick building only consumes the minority of construction time. It is possible, therefore, that limited occupancy in brick building is not a significant bottleneck compared to the latency of brick evaluation.

Interestingly, Figure 20 shows that SM throughput is higher at larger brick sizes when edit culling is disabled. This does not necessarily mean improved performance; the lack of edit culling simply produces more work to be done, which causes a higher SM throughput. It is shown in Figure 17 that doing less work in the first place is of lower latency than doing more work at a higher throughput.

One interesting finding is that the properties of each scene are reflected through the ALU and LSU hardware unit throughputs displayed in Figure 22. The Fractal scene is composed of only a few, but expensive, edits, therefore it is anticipated that more time will be spent evaluating the distance field versus loading edits from memory. Indeed, ALU throughput is significantly higher than LSU throughput when constructing the Fractal scene.

Conversely, the Drops scene has a much greater number of edits, while each edit is faster to evaluate. This is once again reflected in the results; ALU throughput is low, and LSU throughput is high. It can be inferred that optimizing how edits are stored and loaded will see limited performance gains in scenes such as Fractal. Ultimately, it may be the case that it is largely hardware dependent – on a GPU that is bound by memory-bandwidth, scenes composed of a lesser number of ALU-heavy edits would be a preferable option.

The results in Figure 23 depict that group-shared memory is generally highly contended for each scene. This is due to the amount of group-shared memory allocated for temporary edit storage in brick evaluation. Making use of group-shared memory during brick evaluation allows all threads in a group to co-operate by loading edits together and evaluating the same edits in lockstep. This greatly reduces the latency involved in reading edit data. However, the number of edits loaded into group-shared

memory at once is configurable. In all scenes tested, temporary group-shared storage for up to 256 edits was allocated. This is excessive for a scene like Fractal or Cubes, as the entire scene is composed of fewer than 256 edits. If available group-shared memory is the bottleneck for launching warps, decreasing this value would allow for a greater number of warps to be executed simultaneously and is therefore likely to decrease edit evaluation latency.

However, not all threads will be able to contribute toward loading edits if the temporary edit storage is less than 64 edits in size, as there will be more threads than slots in the storage. This will result in threads sitting idly at barriers waiting for siblings to load edits when more edits could have been loaded from VRAM simultaneously.

The precise optimal value likely depends on the average number of edits per brick. Despite Figure 23 suggesting that a lack of available GSM is stalling warp launches, Figure 21 shows that near-maximum occupancy is being maintained for brick evaluation at all brick sizes. It is possible, therefore, that only minimal gains in performance will be seen through optimizing this value.

5.1.3. Memory Usage

The optimal case for edit culling is where each brick will point to a single edit, and consequently the index buffer will be as small as possible. The Fractal scene is the closest scene to this optimum, as it contains only 64 well-distributed edits. As such, it is expected that each brick will reference less than 10 edits. This is supported by the results shown in Figure 25. Despite the Fractal having the second most bricks, its index buffer is much smaller compared to all other scenes. By contrast, the Drops scene contains the maximum of 1024 edits, each with a large amount of smooth blending. As such, each brick will generally reference many edits. As expected, a proportionally much larger index buffer is required for the Drops scene despite it containing a similar number of bricks to the Fractal scene.

Utilizing edit culling requires the memory overhead of storing an index buffer, but evaluating objects built from hundreds of edits would be

infeasible without an edit culling solution. Nevertheless, the index buffer for all scenes is negligible in magnitude compared to the size of the brick pool. The decrease in evaluation time is clearly worth the memory overhead of an index buffer.

5.1.4. Visual Fidelity

It is important that any algorithm to construct geometry produces the expected results. As such, the geometry produced by the construction algorithm developed in this research was tested against a ground-truth representation of the same geometry to discern any inconsistencies.
Error in the produced geometry can be seen in Figure 28. It can be seen that geometry appears to become inflated at larger brick sizes. This is likely an artefact of the limited precision of the representation in the brick pool. At lower resolutions, the location of the isosurface can be determined less precisely, leading to rays terminating sphere-tracing earlier and producing an inflated surface. This inflation decreases as the brick size decreases, and a sufficiently small brick size would be able to match the surface shape precisely.

Through the artificially exaggerated visualization of the error presented in Figure 29, noisy deviations in the surface normals can clearly be identified. Unlike the inflation issue, this surface normal noise does not dissipate as the brick size decreases. Instead, the frequency of the noise increases as the brick size decreases, while its magnitude remains roughly constant. It can thus be suggested that noise in the surface normals will be present and visually discernible at all brick sizes, and this can be identified as a limitation of the applicability of this representation of geometry within a game context.

The accuracy of the surface normals is not only limited by the precision of the brick pool or resolution of the object. Error in the normals is further exacerbated as the 1-voxel neighbourhood surrounding bricks in the pool is insufficient to allow proper normal sampling.

To perform central differencing, the SDF should be sampled 6 times at 1-voxel offsets along the 3 primary axes from the point at which the

normal is to be calculated. As sampling the SDF using linear filtering requires an additional 1-voxel neighbourhood, therefore, calculating normals with this method requires a 2-voxel neighbourhood. Consequently, calculating the normals at the edge of a brick will cross the boundary into the adjacent brick and result in an incorrect normal. This could be solved by storing a 2-voxel neighbourhood around each brick. However, this would dramatically increase the percentage of adjacency data of total brick data to 78.4%.

To combat this, the offset for the central-differencing method was decreased to a ½ voxel offset. This prevents the sampler from reading across brick boundaries but exaggerates noise within the surface normals due to the low precision of the volume. This can be seen in Figure 30.
One solution would be to instead, in the cases where bilinear sampling would cross brick boundaries, point-sample the required brick and manually interpolate. This will lead to a significant increase in divergent texture accesses, but the required voxel will be correctly sampled. However, this is very non-trivial to implement in the structure implemented in this research, as compacting the brick array means brick indices can no longer be inferred from 3D co-ordinates. Consequently, there is no simple way to retrieve a texture co-ordinate within the brick pool from an arbitrary point in space. A separate look-up table of Morton codes to brick indices would need to be constructed and maintained. Nevertheless, even sampling the correct voxels will still contain noise due to the limited precision of the volume, so this would not entirely solve the issue.

An alternative solution is to use the analytical normal method presented by (Evans, 2022), which provides normals that are continuous across voxel boundaries but not quite true to the actual surface geometry. Nevertheless, Evans found that the continuity is enough for it to be aesthetically acceptable and the deviation from the true geometric normal is small enough to be imperceptible.

In general, noise in the normals only presents itself most notably in the specular reflections of highly smooth and reflective surfaces, such as smooth metallic materials. The noise is much less pronounced for rougher surfaces.

5.2 Critical Evaluation

The fact that different brick counts were obtained when constructing objects with and without edit culling, as shown in Table 4 and Table 5, could suggest an inaccuracy in the edit culling algorithm that will produce different geometry than expected. This is investigated in Figure 32, which displays the difference between the Drops scene rendered with and without edit culling. There does exist differences, which confirms that the edit culling algorithm does not correctly cull edits in all cases. However, these differences are imperceptible without the exaggeration and magnification exerted in Figure 33. It is unclear if these imperfections are significant enough to produce a perceptibly incorrect surface without magnification.

The difference in brick count between the scenes rendered with and without edit culling is larger than the visual error in the edit culling algorithm seen in Figure 33 would suggest. This could be explained by a limitation in the evaluation methodology. Upon disabling edit culling, the scene was reset and then progressed to timestamp t = 5s. Due to floating point precision and variable frame time, the timestamp at which the scene was measured with edit culling enabled and disabled could differ by a small amount, causing the slight difference in brick count. Nevertheless, it is unlikely that this had a marked effect on the construction performance.

It can be suggested that there is a trade-off to be made with selecting a brick size to construct an object with. As stated previously, the most expensive stage of construction is brick evaluation. The cost of evaluation is largely dependent on the nature of the edits constituting the object. This was the motivation behind implementing the edit culling scheme. An implication of this is the possibility that decreasing the brick size can improve the construction performance.

Regardless of the culling scheme, larger bricks are more likely to intersect with a greater number of edits. This is especially true for edits utilizing smooth blending. Conversely, it is likely that small bricks will intersect fewer edits. With a good edit culling solution, these small bricks will be very fast to evaluate. It can therefore be assumed that the quantity

and properties of the edits should be considered when selecting the optimal brick size to construct an object with.

Where large numbers of edits are clumped together in very close proximity, the potential performance benefit of edit culling is diminished. This is the worst-case scenario for construction. Thankfully, this is an uncommon situation for game assets – it is less likely that a clump of edits will be a natural way to construct a model.

However, this does motivate the concept of simplifying an edit list prior to construction. It is possible for two edits to precisely cancel each other out – a union followed by the subtraction of an identical edit will nullify any effect that the union may have had. Therefore, the union could be removed from the edit list without any affect to the resulting geometry. Solving this issue programmatically is not trivial, and perhaps simply best left as advice for end users to create performant geometry.

One of the issues that emerges from these findings is that the discretization of the distance field is a slow process and challenging to perform at fast enough rates to allow use within real-time. Furthermore, it is a waste of time to evaluate a brick if a ray never intersects it. Therefore, a scheme could be construed where only bricks that intersect with a ray are ever evaluated.

This could be achieved through a deferred evaluation scheme. If a ray intersects an AABB for which a brick has not yet been evaluated, then it could inform the CPU of this, and that brick can be scheduled for evaluation before the next render occurs. Therefore, bricks will be evaluated in a view-informed manner, at the expense of a latency of at least one frame. This works similar to the streaming technology in *Gigavoxels* (Crassin, 2009).

The other option is to not perform brick evaluation at all and avoid the discretization of the distance field outright. Instead, construction will only produce the bricks and their index buffers. Upon ray-AABB intersection, the analytical distance functions of the relevant edits can be evaluated for each iteration of sphere-tracing. The results obtained in this research showed that evaluation can be fast with good edit culling. In cases

where the object is far away or obscured, it is speculated this method could potentially be faster than evaluating the entire distance field, as only samples that are required to be visited will be evaluated. However, in poor cases where edit culling is not effective this method would likely become prohibitively slow. Another advantage of this method is the resolution and precision of the surface would no longer be bound by the discretization of the volume. This would allow the resolution of the surface to adjust dynamically to match the requirements of the viewpoint.

In this artefact, it was chosen that bricks would be 8^3 samples in size. This was chosen mainly due to the convenient mapping between samples and threads in a group but does not necessarily need to be so. (Crassin, 2009) suggests an example implementation where bricks are 32^3 samples in size. The effect of varying the brick resolution could be investigated.

Using higher resolution bricks, for example, 10^3 bricks (8^3 with a 1 voxel neighbourhood), could be implemented using a compute shader containing 10^3 threads – within the maximum limit of 1024 threads in a group. Alternatively, the thread group dimensions could also remain at 8^3, where 488 of the 512 threads would evaluate two samples to populate a 10^3 brick. The effect on construction performance between the two thread group sizes could be investigated to determine if the work imbalance in the 8^3-thread model has any impact. The effect on rendering performance and visual fidelity could also be investigated. 10^3 bricks could also facilitate the 2-voxel neighbourhood required for correct surface normals as calculated with the central-differencing method.

Chapter 6 Conclusion and Future Work

6.1 Overview

This study set out to investigate the feasibility of SDF-based geometry that can be modified in real-time within a game context. This was done by implementing an application that can sparsely store SDF geometry as a discrete distance field and render it via raytracing. The structure implemented in this application was based on the previous work by (Crassin, 2009), (Laine & Karras, 2010), and (Evans, 2022).

A construction algorithm was designed and implemented such that this geometry could be reconstructed in real-time. This algorithm worked in a top-down approach, iteratively refining bricks into sub-bricks. Edit culling optimizations were implemented into the construction pipeline to make it suitable for real-time performance.

This study has demonstrated the feasibility of using modifiable SDF objects in a real-time application. The construction algorithm could create objects composed of hundreds of primitive edits and hundreds of thousands of bricks at interactive framerates. Crucially, the edit culling algorithm implemented into the construction pipeline reduced construction times from multiple seconds to within 100ms. Through reconstructing each frame, the objects can be animated and dynamic, which was either not possible or documented in previous work. The construction algorithm was designed to make effective use of the GPU hardware. This was achieved in parts; at smaller brick sizes, SM throughput reached as high as 80%, and brick evaluation saw occupancy of over 95%. Hierarchical brick building only achieved a maximum occupancy of approximately 30%, so further optimizations are worthwhile.

The study has also shown that rendering SDF objects using a combination of hardware-accelerated raytracing and software sphere-tracing was sufficiently fast for real-time use. An interesting finding was that the time taken to render an object was not proportional to the number of bricks of which it was composed.

A limitation in this study is that correct surface normals were not able to be obtained numerically at any brick size. Increasing the resolution of

the object increased the frequency of the noise in the surface normals but did not reduce its visual prevalence. This is exacerbated by a lack of neighbourhood information within each brick, which limits the sampling offset and contributes to the visually displeasing noise in the surface normals. This noise produces perceptibly incorrect specular lighting and reflections from secondary rays and limits the use of this technology in a game context, where artefact-free rendering is extremely important.

While the study found that rendering time scales well with the number of bricks composing a scene, significantly improved edit culling would be required to make this technology useful for a game to allow for objects composed of a significantly greater number of edits. The culling technique in this application is a promising proof-of-concept that is effective for a project of this scale, and it has scope for future optimization for more widespread use.

6.2 Future Work

Further research might explore a precise optimization of brick size. For example, for an object to be constructed from a specific edit list and rendered from a specific viewpoint, there must exist an optimal brick size that would allow for the fastest construction time, while not compromising on the visual fidelity of the object. The fastest construction time will be some optimization between constructing as few bricks as possible, while also evaluating as few edits per brick as possible. This is unlikely to be generally trivial to identify, as many factors affect construction time, including the amount of smooth blending used within edits, and the complexity to evaluate each individual edit.

This could motivate future research to investigate the possibility and usefulness of a level-of-detail scheme, similar to the idea implemented in *Gigavoxels*. LOD systems are prevalent in games, for both improving visual fidelity and performance. However, maintaining multiple levels of detail for an SDF object will come with significant construction overhead. As shown in this study, rendering time does not correlate with brick size, and consequently LOD schemes may not improve rendering times either. The implementation and evaluation of such a scheme into the technology

presented in this study would be beneficial to further understanding the application of this technology within a game context.

Another potential direction for future research to tackle level-of-detail would be to investigate the possibility of on-the-fly edit evaluation, as mentioned in section 5.2, removing the need for a brick pool and the discretization of the distance field. This would likely require the investigation of highly improved edit culling schemes to be feasible for the real-time rendering of objects composed of many edits.

Future studies could also investigate to which degree the analytical intersection method proposed by (Evans, 2022) affects the time taken to render SDF objects and investigate if their method of calculating surface normals analytically improves the fidelity issues present in this study to make this technology suitable for a game.

Finally, future research might further explore how shading attributes for the geometry may be integrated into the technology presented in this study. Whether building photorealistic environments or stylised worlds, materials and shading models are critical to bring the geometry used in a virtual environment to life. Using a discrete distance field allows for the storage of attributes per voxel, which is generally much finer-grained than per-vertex as would be in a triangle mesh. This could be taken advantage of to produce highly detailed geometry. A future investigation could determine the methods, advantages, and limitations of storing shading attributes at a per-voxel basis.

References

Aaltonen, S. (2018) ‘GPU-based clay simulation and ray tracing tech in Claybook’ *Game Developers Conference.* Available at: https://ubm-twvideo01.s3.amazonaws.com/o1/vault/gdc2018/presentations/Aaltonen_Sebastian_GPU_Based_Clay.pdf (Accessed: September 2023).

Aeva (2022) *A Better Approach to SDF Decomposition*. Available at: http://zone.dog/braindump/sdf_clustering_part_2/ (Accessed: February 2024).

Crassin, C. *et al*. (2009) ‘Gigavoxels’, *Proceedings of the 2009 symposium on Interactive 3D graphics and games.* Available at: https://dl.acm.org/doi/10.1145/1507149.1507152 (Accessed: September 2023).

Evans, A. (2015) ‘Learning from failure: A survey of promising, unconventional and mostly abandoned renderers for Dreams PS4, a geometrically dense, painterly UGC game’, *Advances in Real-Time Rendering in Games, SIGGRAPH Course.* Available at: http://advances.realtimerendering.com/s2015/AlexEvans_SIGGRAPH-2015-sml.pdf (Accessed: September 2023).

Evans, A., *et al*. (2022) ‘Ray Tracing of Signed Distance Function Grids’, Journal of Computer Graphics Techniques, Volume 11. Available at: https://jcgt.org/published/0011/03/06/paper-lowres.pdf (Accessed: October 2023).

Harada, T. and Howes, L. (no date) *Introduction to GPU Radix Sort*, *AMD GPUOpen*. Available at: https://gpuopen.com/download/publications/Introduction_to_GPU_Radix_Sort.pdf (Accessed: January 2024).

Hart, J.C. (1996) ‘Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces’, *The Visual Computer*, 12(10), pp. 527–545. doi:10.1007/s003710050084.

Karras, T. (2012) ‘Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees’, *EGGH-HPG’12: Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics.* Available at: https://dl.acm.org/doi/pdf/10.5555/2383795.2383801 (Accessed: October 2023).

Kramer, L. (2023) *Real-time Sparse Distance Fields for Games*, *AMD GPUOpen*. Available at: https://gpuopen.com/gdc-presentations/2023/GDC-2023-Sparse-Distance-Fields-For-Games.pdf (Accessed: March 2024).

Laine, S. and Karras, T. (2010) ‘Efficient Sparse Voxel Octrees’, *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*. Available at: https://dl.acm.org/doi/pdf/10.1145/1730804.1730814 (Accessed: September 2023).

Quilez, I. (no date) *Distance Functions*, *Inigo Quilez*. Available at: https://iquilezles.org/articles/distfunctions/ (Accessed: September 2023).

Quilez, I. (no date) *Numerical Normals for SDFs*, *Inigo Quilez*. Available at: https://iquilezles.org/articles/normalsSDF/ (Accessed: September 2023).

Quilez, I. (no date) Shadertoy. Available at: https://www.shadertoy.com/ (Accessed: April 2024).

Tatarchuk, N. (2005) *Practical Dynamic Parallax Occlusion Mapping*. Available at: https://www.gamedevs.org/uploads/practical-dynamic-parallax-occlusion-mapping.pdf (Accessed: April 2024).

Appendices

Appendix 1 – Rendering Complete Data

Demo Name	Cubes	Cubes	Cubes	Cubes	Drops	Drops	Drops	Drops	Fractal	Fractal
Edit Culling	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled
Brick Size	0.0625	0.125	0.25	0.5	0.0625	0.125	0.25	0.5	0.0625	0.125
Brick Count	183334	55760	12407	2693	377049	82477	19556	4100	444780	107746
Edit Count	216	216	216	216	1024	1024	1024	1024	64	64
Duration (ns)	4247117	4255498	3616082	3570515	3712694	3048901	3058448	3289123	9418325	8404724
RT Throughput (%)	38	43	51	54	45	53	56	57	34	42
SM Throughput (%)	37	39	38	38	40	39	37	37	39	40
L2 Hit Rate (%)	72	79	84	88	73	81	85	86	73	81

Fractal	**Fractal**	**Rain**	**Rain**	**Rain**	**Rain**	**Cubes**	**Cubes**	**Cubes**	**Cubes**	**Drops**	**Drops**	**Drops**
Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Enabled	Enabled	Enabled	Enabled	Enabled	Enabled	Enabled
0.25	0.5	0.0625	0.125	0.25	0.5	0.0625	0.125	0.25	0.5	0.0625	0.125	0.25
27104	7221	238135	60702	15938	4162	176438	53549	12493	2693	362797	79221	19084
64	64	512	512	512	512	216	216	216	216	1024	1024	1024
8434266	9480483	3205068	2730344	2685439	3025711	4116628	4123380	3670040	3496113	3764208	3073218	3091858
47	50	41	51	56	57	38	44	51	54	45	52	56
39	36	37	38	38	35	36	39	39	38	39	38	36
86	88	65	83	84	86	71	79	84	88	73	81	85

Drops	**Fractal**	**Fractal**	**Fractal**	**Fractal**	**Rain**	**Rain**	**Rain**	**Rain**
Enabled	Enabled	Enabled	Enabled	Enabled	Enabled	Enabled	Enabled	Enabled
0.5	0.0625	0.125	0.25	0.5	0.0625	0.125	0.25	0.5
4042	444894	107629	27118	7113	234905	59884	15806	4148
1024	64	64	64	64	512	512	512	512
3301086	9535619	8458571	8496175	9372416	3248176	2698566	2701376	3063871
57	34	42	47	50	40	51	56	57
35	38	39	38	36	35	38	37	35
87	73	81	86	88	63	84	85	87

Appendix 2 – Construction Complete Data

Scene	Cubes	Cubes	Cubes	Cubes	Cubes	Cubes
Edit Culling	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled
Brick Size	0.0625	0.0625	0.0625	0.125	0.125	0.125
Range	AABB Building	Brick Building	Brick Evaluation	Edit Dependencies	AABB Building	Brick Building	Brick Evaluation
Brick Count	183491	183491	183491	183491	55765	55765	55765
Edit Count	216	216	216	216	216	216	216
Duration (ns)	26391	4474297	329974237	4473	5297	1308855	100329221
Warp Activity (%)	66	52	99	14	20	40	99
SM Throughput (%)	6	78	93	13	10	66	93
ALU Throughput (%)	1	43	53	3	1	35	53
LSU Throughput (%)	6	78	45	13	10	66	45
Stall (Register) (%)	0	0	50	0	0	0	50
Stall (CTA) (%)	48	94	31	20	48	87	41
Stall (GSM) (%)	0	0	50	0	0	0	50
Stall (Warp Slot) (%)	34	0	50	0	17	1	50

Cubes	**Cubes**	**Cubes**	**Cubes**	**Cubes**	**Cubes**	**Cubes**	**Cubes**
Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled
0.125	0.25	0.25	0.25	0.25	0.5	0.5	0.5
Edit Dependencies	AABB Building	Brick Building	Brick Evaluation	Edit Dependencies	AABB Building	Brick Building	Brick Evaluation
55765	12406	12406	12406	12406	2693	2693	2693
216	216	216	216	216	216	216	216
4465	3730	394254	22343000	4472	3382	261404	4893194
14	4	23	98	14	1	7	97
13	3	47	93	13	1	17	92
3	0	25	53	3	0	9	53
13	3	47	45	13	1	17	45
0	0	0	50	0	0	0	48
20	48	85	48	20	48	85	47
0	0	0	50	0	0	0	48
0	23	1	50	0	24	2	48

Cubes	**Drops**	**Drops**	**Drops**	**Drops**	**Drops**	**Drops**	**Drops**
Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled
0.5	0.0625	0.0625	0.0625	0.0625	0.125	0.125	0.125
Edit Dependencies	AABB Building	Brick Building	Brick Evaluation	Edit Dependencies	AABB Building	Brick Building	Brick Evaluation
2693	376973	376973	376973	376973	82439	82439	82439
216	1024	1024	1024	1024	1024	1024	1024
4472	51097	21399241	2315382964	24457	10390	5312269	506529378
14	74	58	100	50	39	49	100
13	7	86	94	52	7	76	94
3	1	30	41	15	1	26	41
13	7	86	62	52	7	76	62
0	0	0	50	0	0	0	50
20	48	93	5	36	47	91	24
0	0	0	50	0	0	0	50
0	40	0	50	0	16	0	50

Drops	**Drops**	**Drops**	**Drops**	**Drops**	**Drops**	**Drops**	**Drops**
Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled
0.125	0.25	0.25	0.25	0.25	0.5	0.5	0.5
Edit Dependencies	AABB Building	Brick Building	Brick Evaluation	Edit Dependencies	AABB Building	Brick Building	Brick Evaluation
82439	19547	19547	19547	19547	4102	4102	4102
1024	1024	1024	1024	1024	1024	1024	1024
24463	3927	1879615	120139739	24460	3413	980945	25470425
50	7	31	100	50	1	14	98
52	5	49	94	52	1	30	93
15	1	17	41	15	0	10	40
52	5	49	62	52	1	30	62
0	0	0	50	0	0	0	49
36	48	78	41	36	48	89	46
0	0	0	50	0	0	0	49
0	22	0	50	0	24	1	49

Drops	**Fractal**	**Fractal**	**Fractal**	**Fractal**	**Fractal**	**Fractal**	**Fractal**
Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled
0.5	0.0625	0.0625	0.0625	0.0625	0.125	0.125	0.125
Edit Dependencies	AABB Building	Brick Building	Brick Evaluation	Edit Dependencies	AABB Building	Brick Building	Brick Evaluation
4102	444894	444894	444894	444894	107750	107750	107750
1024	64	64	64	64	64	64	64
24450	59736	12713893	1200446628	2291	10665	3461068	290778378
50	75	58	99	1	34	53	99
52	7	80	85	1	9	77	85
15	1	53	59	0	1	51	59
52	7	26	9	1	9	24	9
0	0	0	50	0	0	0	50
36	48	94	10	15	47	92	34
0	0	0	50	0	0	0	50
0	41	0	50	0	15	0	50

Fractal	**Fractal**	**Fractal**	**Fractal**	**Fractal**	**Fractal**	**Fractal**	**Fractal**
Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled
0.125	0.25	0.25	0.25	0.25	0.5	0.5	0.5
Edit Dependencies	AABB Building	Brick Building	Brick Evaluation	Edit Dependencies	AABB Building	Brick Building	Brick Evaluation
107750	27107	27107	27107	27107	7201	7201	7201
64	64	64	64	64	64	64	64
2295	4156	943426	73215723	2296	3561	352392	19493626
1	9	45	99	1	2	19	99
1	6	70	85	1	2	48	85
0	1	47	59	0	0	32	59
1	6	22	9	1	2	15	9
0	0	0	50	0	0	0	49
15	47	84	44	15	48	79	47
0	0	0	50	0	0	0	49
0	21	1	50	0	23	2	49

Fractal	**Rain**	**Rain**	**Rain**	**Rain**	**Rain**	**Rain**	**Rain**
Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled
0.5	0.0625	0.0625	0.0625	0.0625	0.125	0.125	0.125
Edit Dependencies	AABB Building	Brick Building	Brick Evaluation	Edit Dependencies	AABB Building	Brick Building	Brick Evaluation
7201	238107	238107	238107	238107	60666	60666	60666
64	512	512	512	512	512	512	512
2296	33196	9853100	731195061	8792	7274	2900822	186327694
1	70	54	99	38	23	47	99
1	7	85	94	35	8	76	94
0	1	30	41	9	1	27	41
1	7	85	62	35	8	76	62
0	0	0	50	0	0	0	50
15	48	94	17	27	50	90	42
0	0	0	50	0	0	0	50
0	36	0	50	0	14	0	50

Rain	**Rain**	**Rain**	**Rain**	**Rain**	**Rain**	**Rain**	**Rain**
Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled
0.125	0.25	0.25	0.25	0.25	0.5	0.5	0.5
Edit Dependencies	AABB Building	Brick Building	Brick Evaluation	Edit Dependencies	AABB Building	Brick Building	Brick Evaluation
60666	15934	15934	15934	15934	4160	4160	4160
512	512	512	512	512	512	512	512
8783	3865	1005368	49028300	8797	3447	516496	12881333
38	5	29	99	38	1	12	98
35	4	48	94	35	1	28	93
9	1	17	41	9	0	10	41
35	4	48	62	35	1	28	62
0	0	0	50	0	0	0	49
27	48	78	46	27	48	88	47
0	0	0	50	0	0	0	49
0	22	1	50	0	23	1	49

Rain	**Cubes**	**Cubes**	**Cubes**	**Cubes**	**Cubes**	**Cubes**	**Cubes**
Disabled	Enabled	Enabled	Enabled	Enabled	Enabled	Enabled	Enabled
0.5	0.0625	0.0625	0.0625	0.0625	0.125	0.125	0.125
Edit Dependencies	AABB Building	Brick Building	Brick Evaluation	Edit Dependencies	AABB Building	Brick Building	Brick Evaluation
4160	176422	176422	176422	176422	53620	53620	53620
512	216	216	216	216	216	216	216
8798	25347	1923743	32465109	4465	5153	726104	10569806
38	66	35	98	14	19	25	98
35	6	73	92	13	10	56	92
9	1	35	49	3	1	26	49
35	6	73	46	13	10	56	46
0	0	0	50	0	0	0	50
27	48	92	45	20	47	85	48
0	0	0	50	0	0	0	50
0	34	0	50	0	18	1	50

Cubes	**Cubes**	**Cubes**	**Cubes**	**Cubes**	**Cubes**	**Cubes**	**Cubes**
Enabled	Enabled	Enabled	Enabled	Enabled	Enabled	Enabled	Enabled
0.125	0.25	0.25	0.25	0.25	0.5	0.5	0.5
Edit Dependencies	AABB Building	Brick Building	Brick Evaluation	Edit Dependencies	AABB Building	Brick Building	Brick Evaluation
53620	12491	12491	12491	12491	2693	2693	2693
216	216	216	216	216	216	216	216
4452	3966	306172	2208836	4458	3375	243107	476841
14	4	16	98	14	1	6	95
13	3	36	92	13	1	15	89
3	0	17	49	3	0	8	48
13	3	36	46	13	1	15	45
0	0	0	49	0	0	0	47
20	47	81	48	20	48	76	48
0	0	0	49	0	0	0	47
0	23	2	49	0	23	2	47

Cubes	**Drops**	**Drops**	**Drops**	**Drops**	**Drops**	**Drops**	**Drops**
Enabled	Enabled	Enabled	Enabled	Enabled	Enabled	Enabled	Enabled
0.5	0.0625	0.0625	0.0625	0.0625	0.125	0.125	0.125
Edit Dependencies	AABB Building	Brick Building	Brick Evaluation	Edit Dependencies	AABB Building	Brick Building	Brick Evaluation
2693	362731	362731	362731	362731	79248	79248	79248
216	1024	1024	1024	1024	1024	1024	1024
4461	49262	4719206	67220664	24359	9549	1759300	16965098
14	74	34	98	49	36	23	98
13	7	67	92	52	8	46	93
3	1	28	36	14	1	19	36
13	7	67	62	52	8	46	62
0	0	0	50	0	0	0	50
20	48	92	46	36	49	85	47
0	0	0	50	0	0	0	50
0	40	0	50	0	15	1	50

Drops	**Drops**	**Drops**	**Drops**	**Drops**	**Drops**	**Drops**	**Drops**
Enabled	Enabled	Enabled	Enabled	Enabled	Enabled	Enabled	Enabled
0.125	0.25	0.25	0.25	0.25	0.5	0.5	0.5
Edit Dependencies	AABB Building	Brick Building	Brick Evaluation	Edit Dependencies	AABB Building	Brick Building	Brick Evaluation
79248	19083	19083	19083	19083	4043	4043	4043
1024	1024	1024	1024	1024	1024	1024	1024
24272	3893	1034644	4463041	24195	3434	804910	1085517
49	6	16	98	49	1	7	97
52	5	25	92	52	1	14	91
14	1	10	36	15	0	6	35
52	5	25	62	52	1	14	61
0	0	0	50	0	0	0	49
36	48	82	48	36	47	78	48
0	0	0	50	0	0	0	49
0	22	1	50	0	23	1	49

Drops	**Fractal**	**Fractal**	**Fractal**	**Fractal**	**Fractal**	**Fractal**	**Fractal**
Enabled	Enabled	Enabled	Enabled	Enabled	Enabled	Enabled	Enabled
0.5	0.0625	0.0625	0.0625	0.0625	0.125	0.125	0.125
Edit Dependencies	AABB Building	Brick Building	Brick Evaluation	Edit Dependencies	AABB Building	Brick Building	Brick Evaluation
4043	444899	444899	444899	444899	107653	107653	107653
1024	64	64	64	64	64	64	64
24263	59801	3067383	24897335	2295	10834	877279	6033314
49	75	34	98	1	35	29	97
52	7	67	79	1	9	60	79
14	1	43	59	0	1	38	59
52	7	58	19	1	9	50	19
0	0	0	49	0	0	0	49
36	48	91	46	16	49	89	48
0	0	0	49	0	0	0	49
0	41	0	49	0	16	1	49

Fractal	**Fractal**	**Fractal**	**Fractal**	**Fractal**	**Fractal**	**Fractal**	**Fractal**
Enabled	Enabled	Enabled	Enabled	Enabled	Enabled	Enabled	Enabled
0.125	0.25	0.25	0.25	0.25	0.5	0.5	0.5
Edit Dependencies	AABB Building	Brick Building	Brick Evaluation	Edit Dependencies	AABB Building	Brick Building	Brick Evaluation
107653	27107	27107	27107	27107	7111	7111	7111
64	64	64	64	64	64	64	64
2287	4139	322590	1527001	2290	3563	195892	412924
1	9	22	97	1	2	11	96
1	6	47	79	1	2	28	78
0	1	30	59	0	0	18	58
1	6	35	19	1	2	17	18
0	0	0	49	0	0	0	48
16	48	86	48	16	48	81	48
0	0	0	49	0	0	0	48
0	21	2	49	0	22	3	48

Fractal	**Rain**	**Rain**	**Rain**	**Rain**	**Rain**	**Rain**	**Rain**
Enabled	Enabled	Enabled	Enabled	Enabled	Enabled	Enabled	Enabled
0.5	0.0625	0.0625	0.0625	0.0625	0.125	0.125	0.125
Edit Dependencies	AABB Building	Brick Building	Brick Evaluation	Edit Dependencies	AABB Building	Brick Building	Brick Evaluation
7111	234863	234863	234863	234863	59877	59877	59877
64	512	512	512	512	512	512	512
2293	32828	3341117	53870618	8799	7091	1186996	14406250
1	69	36	98	38	23	26	98
1	7	71	93	35	8	54	93
0	1	31	35	9	1	24	35
1	7	71	62	35	8	54	62
0	0	0	50	0	0	0	50
16	48	92	45	27	51	86	47
0	0	0	50	0	0	0	50
0	36	0	50	0	14	1	50

Rain	**Rain**	**Rain**	**Rain**	**Rain**	**Rain**	**Rain**	**Rain**
Enabled	Enabled	Enabled	Enabled	Enabled	Enabled	Enabled	Enabled
0.125	0.25	0.25	0.25	0.25	0.5	0.5	0.5
Edit Dependencies	AABB Building	Brick Building	Brick Evaluation	Edit Dependencies	AABB Building	Brick Building	Brick Evaluation
59877	15808	15808	15808	15808	4144	4144	4144
512	512	512	512	512	512	512	512
8784	3800	588333	4016043	8772	3609	427444	1060615
38	5	17	98	38	1	8	96
35	4	31	92	35	1	17	91
9	1	13	35	9	0	7	34
35	4	31	62	35	1	17	61
0	0	0	49	0	0	0	48
27	48	81	47	27	51	78	48
0	0	0	49	0	0	0	48
0	23	1	49	0	23	1	48

Rain
Enabled
0.5
Edit Dependencies
4144
512
8799
38
35
9
35
0
27
0
0