MASTERARBEIT

zur Erlangung des akademischen Grades
„Master of Science in Engineering“
im Studiengang Game Engineering und Simulation

Signed Distance Fields in Real-Time Rendering

Ausgeführt von: Michael Mroz, BSc
Personenkennzeichen: 1410585012

1. BegutachterIn: DI Stefan Reinalter
2. BegutachterIn: Dr. Gerd Hesina

Philadelphia, am 26.04.2017

FH University of Applied Sciences
TECHNIKUM
WIEN

Eidesstattliche Erklärung

„Ich, als Autor / als Autorin und Urheber / Urheberin der vorliegenden Arbeit, bestätige mit meiner Unterschrift die Kenntnisnahme der einschlägigen urheber- und hochschulrechtlichen Bestimmungen (vgl. etwa §§ 21, 46 und 57 UrhG idgF sowie § 11 Satzungsteil Studienrechtliche Bestimmungen / Prüfungsordnung der FH Technikum Wien).

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig angefertigt und Gedankengut jeglicher Art aus fremden sowie selbst verfassten Quellen zur Gänze zitiert habe. Ich bin mir bei Nachweis fehlender Eigen- und Selbstständigkeit sowie dem Nachweis eines Vorsatzes zur Erschleichung einer positiven Beurteilung dieser Arbeit der Konsequenzen bewusst, die von der Studiengangsleitung ausgesprochen werden können (vgl. § 11 Abs. 1 Satzungsteil Studienrechtliche Bestimmungen / Prüfungsordnung der FH Technikum Wien).

Weiters bestätige ich, dass ich die vorliegende Arbeit bis dato nicht veröffentlicht und weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt habe. Ich versichere, dass die abgegebene Version jener im Uploadtool entspricht."

Philadelphia, am 26.04.2017	[illegible]
Ort, Datum	Unterschrift

Kurzfassung

Die Branche der Echtzeit-Computergrafik ist bestrebt die Synthetisierung von möglichst realistischen Bildern voranzutreiben. Viele der heute in Spielen verwendeten Beleuchtungs-Techniken wurden vor mehr als 20 Jahren entwickelt, doch durch die stetige Verbesserung der Rechenleistung von Grafikkarten in den letzten Jahren rücken nun neue Techniken in den Fokus. Diese Techniken sind zwar langsamer, bieten aber einen höheren Realismus in der Bildsynthese. Diese Masterthesis beschäftigt sich mit einer dieser neuen Techniken, den Distanzfeldern. Durch Distanzfelder lassen sich viele der üblichen Lichteffekte, wie Schatten oder Umgebungsverdeckung, auf eine viel intuitivere Weise berrechnen. Diese Arbeit präsentiert eine detaillierte Beschreibung der populärsten Distanzfeld-Effekte, sowie eine umfassende Analyse der Anwendbarkeit im Bereich der Spieleentwicklung. Ein besonderer Fokus wird auf die Ermittlung der Leistungsfähigkeit gelegt, zudem werden neuartige Algorithmen vorgestellt um die Bildsynthese zu beschleunigen.

Schlagwörter: Distance Fields, Real-Time Rendering, SDF, Ray marching, Sphere tracing, Optimizations, Culling, Deferred Rendering

Abstract

In modern real-time rendering, there is a large focus on producing photorealistic qualities in images. Most techniques used to produce these effects were developed with older, less powerful hardware in mind and sacrifice visual fidelity for fast computation. Now that more and more powerful hardware has been developed, research into alternatives for these well-established techniques has gained traction in the field of computer graphics. Distance fields are one of these contemporary alternatives, and they offer more intuitive approaches to well-established techniques such as ambient occlusion and object shadows. This thesis will give an in-depth look into the most common distance field effects and present a variety of acceleration techniques to show the viability of distance fields for modern game development.

Keywords: Distance Fields, Real-Time Rendering, SDF, Ray marching, Sphere tracing, Optimizations, Culling, Deferred Rendering

Acknowledgments

I would like to thank DI Stefan Reinalter for the extraordinary support and the inspiration to aim for the stars. I am very grateful to Prof. Norman Badler and the University of Pennsylvania for the hospitality and everyone from SIG Lab for their encouragement and support. Last but not least, many thanks and much love to all my friends, family and Eli.
This project would not have been possible without the generous support of the Austrian Marshall Plan Foundation.

Table of Contents

1 Introduction

Since its inception over 40 years ago, the video game industry has strived towards more and more realistic graphics in their games. This desire has led to the development of both more and more powerful rendering algorithms and graphics hardware. Still, a big gap exists between current real-time renderers and raytracers used for offline rendering of movies. There have been attempts to bridge this gap in recent years[1], but real-time physically based global illumination still seems years away for deployment in high-end video games. In the meantime, a variety of other next-gen rendering approaches have come into focus, which aim to reduce this rendering gap. Signed distance field (SDF) rendering is one of these techniques. Distance fields have a variety of features which make them interesting for real-time rendering purposes, but their low performance compared to conventional algorithms has prevented their employment in all but the narrowest of cases. With increasing graphics processing power, signed distance fields are becoming more and more of an option as an enhancement or even alternative to conventional mesh-based rendering. Although SDFs have seen different uses in the past, from robotic obstacle avoidance to the visual demo scene, up until recently their use in videogames has been very limited. In the past few years, there has been a resurgence and renewed interest in the utilization of distance fields for real time rendering.

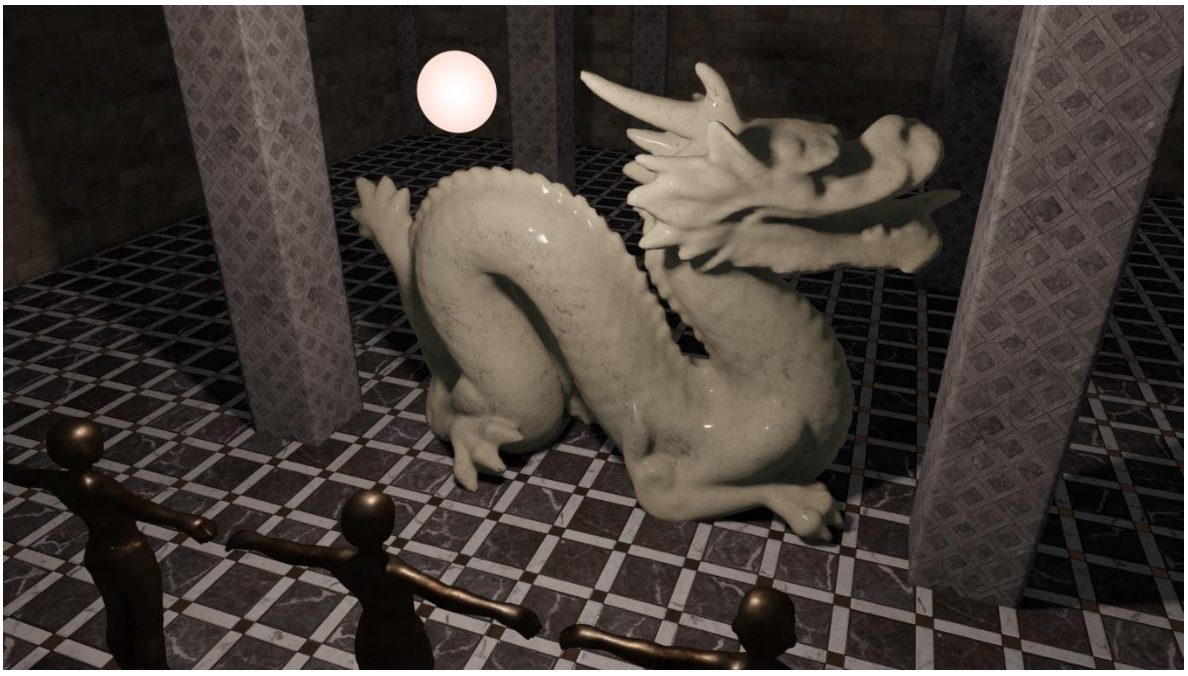
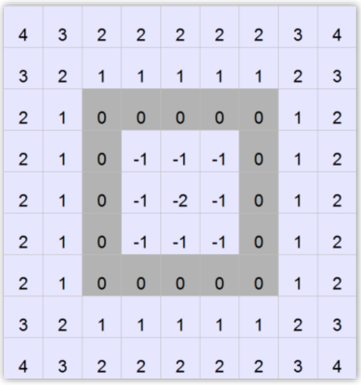


Figure 1: The default scene in the mTec renderer.

[1] Brigade, a real-time Pathtracer. https://home.otoy.com/render/brigade

This thesis will focus on the most popular rendering techniques that can be achieved with distance fields, accompanied with a thorough performance analysis. The first part of the thesis highlights the properties, generation and theory of distance fields, while the second part showcases our implementation of the previously described techniques: a renderer called mTec (Figure 1). The focus during development was set on implementing and expanding upon previously developed acceleration techniques in order to showcase the viability of distance fields for current and future generations of game engines. It will be shown that by employing the correct acceleration techniques real-time distance field rendering is feasible on modern computers. The renderer written for this thesis is open source software, and can be downloaded from the internet freely[2].

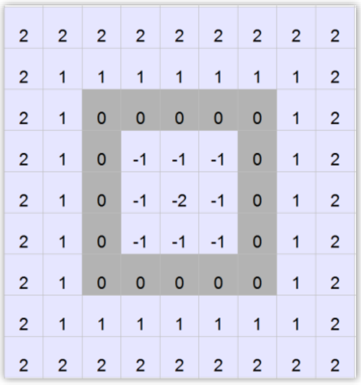
[2] https://github.com/xx3000/mTec



4	3	2	2	2	2	2	3	4
3	2	1	1	1	1	1	2	3
2	1	0	0	0	0	0	1	2
2	1	0	-1	-1	-1	0	1	2
2	1	0	-1	-2	-1	0	1	2
2	1	0	-1	-1	-1	0	1	2
2	1	0	0	0	0	0	1	2
3	2	1	1	1	1	1	2	3
4	3	2	2	2	2	2	3	4

Figure 2-a: Manhattan or Taxicab distance.

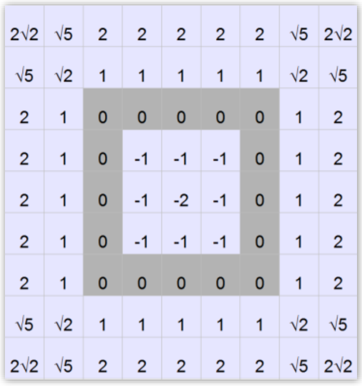
Distance = x+y



2	2	2	2	2	2	2	2	2
2	1	1	1	1	1	1	1	2
2	1	0	0	0	0	0	1	2
2	1	0	-1	-1	-1	0	1	2
2	1	0	-1	-2	-1	0	1	2
2	1	0	-1	-1	-1	0	1	2
2	1	0	0	0	0	0	1	2
2	1	1	1	1	1	1	1	2
2	2	2	2	2	2	2	2	2

Figure 1-b: Chessboard or Chebyshev distance.

Distance = max(x,y)



$2\sqrt{2}$	$\sqrt{5}$	2	2	2	2	2	$\sqrt{5}$	$2\sqrt{2}$
$\sqrt{5}$	$\sqrt{2}$	1	1	1	1	1	$\sqrt{2}$	$\sqrt{5}$
2	1	0	0	0	0	0	1	2
2	1	0	-1	-1	-1	0	1	2
2	1	0	-1	-2	-1	0	1	2
2	1	0	-1	-1	-1	0	1	2
2	1	0	0	0	0	0	1	2
$\sqrt{5}$	$\sqrt{2}$	1	1	1	1	1	$\sqrt{2}$	$\sqrt{5}$
$2\sqrt{2}$	$\sqrt{5}$	2	2	2	2	2	$\sqrt{5}$	$2\sqrt{2}$

Figure 1-c: Euclidean or straight-line distance.

Distance = sqrt(x^2+y^2)

2 Definition

A distance field by itself is a very simple structure. Jones et al. define a distance field is as follows:

"A distance field is a representation where at each point within the field we know the distance from that point to the closest point on any object within the domain." [1]

Therefore, in essence, distance fields are representations of surfaces.
In mathematical terms a distance field function expresses the following structure,

$$dist(p) = sign(p) * \min_{x \in S} ||p - x||$$

where S is a set of points on the surface of an object, which are themselves a subset of O, the points that comprise the entire surface of an object. The sign function above is defined as

$$sign(p) = \begin{cases} -1 \; if \; p \in O \\ \;\; 1 \; otherwise \end{cases}$$

In broader terms, a distance field can be represented by a function F, such that any point p that is passed will return a distance d from the object represented by the function [1].
The distances returned by such a function can then be stored as three dimensional matrices, or more common for rendering purposes, as 3D textures. Each cell of the texture denotes the closest distance from the grid element to the nearest surface. Therefore a grid

element containing a value of 0 represents a surface. Signed distance fields, or SDFs, are an extension in which elements inside objects are represented with negative distances. Figure 2 illustrates a 2D slice of an SDF, containing a rectangular shape. In theory, different distance functions can be used to generate the field. Prominent ones include the Manhattan distance, Chessboard distance and Euclidean distance.

In the context of practical SDF rendering, Euclidean distance is the representation of choice as it is the most physically accurate. There have been experiments with other distance functions such as l-norm [3], which have interesting properties and simpler distance functions for certain primitives. Still, the Euclidean distance remains the most widely used distance function for SDF rendering and it is this function that will be used for all further descriptions and implementations in this thesis.

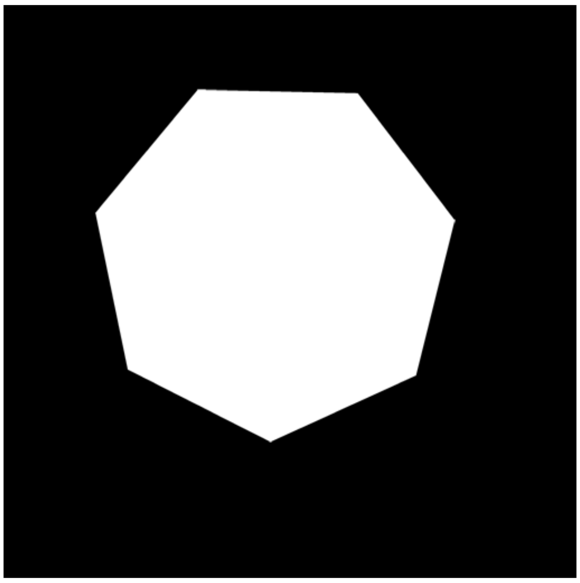


Figure 3-a: Reference image, 4096x4096.

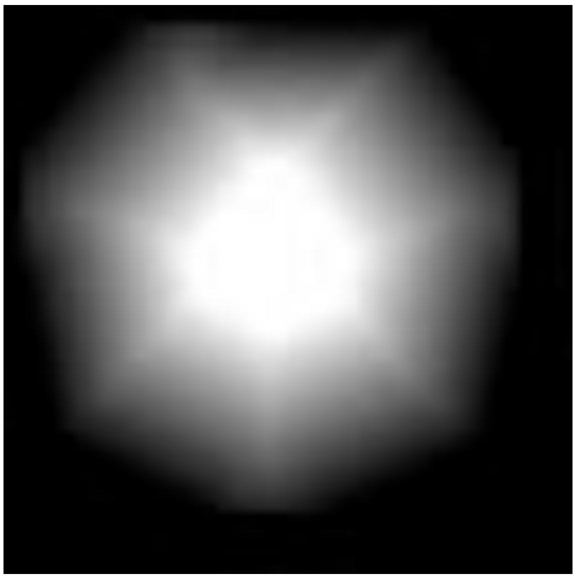


Figure 3-b: Image converted to SDF, 64x64.

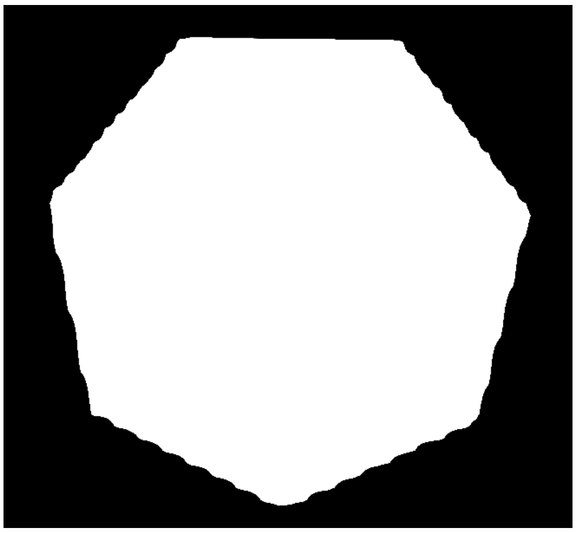


Figure 3-c: Reference image reduced to 64x64 (Photoshop CS3), rendered with alpha testing, bilinear filtering.

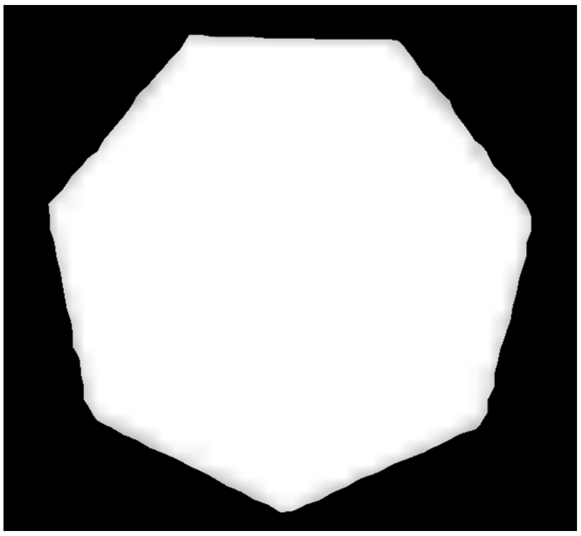


Figure 3-d: SDF from 3-b, rendered with alpha testing, bilinear filtering.

3 History

Distance fields have been in use in the computer graphics field for the past 25 years. The theoretical groundwork for distance transforms has its roots in image processing and algorithms for Euclidian distance mapping have been in use since as early as 1980 [4]. As these early examples were rooted in image processing, all distance maps were 2D in nature. In the early 90s the acceleration properties of distance fields were discovered for faster rendering of ray-traced fractal images. Rendering of fractals was significantly slower compared to solid geometry, as no ray-geometry intersection function was known. What could be calculated was the closest distance to the fractal's surface, which allowed for ray marching with variable step sizes [5]. This essential application of distance fields in ray-tracing will be further outlined in chapter 6. The first usage of distance fields in the area of volume visualization can be traced back to the paper "Acceleration of ray-casting using 3D distance transforms" by Zuiderveld et al, in which distance fields are presented as a novel acceleration method for volume visualization. The sampling points needed for ray tracing are reduced by polling the distance field to determine the closest surface [6]. The idea of distance fields itself wasn't limited to computer graphics: Koditschek et al. proposed a method of robotic navigation through potential fields in 1989 [7]. As the advantages of distance fields were discovered and as computing power increased, the field of application for distance fields diversified. Gibson utilized distance maps for the encoding of surface data and subsequently, rendering. This approach was used as it maintains an accurate surface representation even at lower-quality samplings [8] [9].

In the 2000s the computer 4k demo scene started to take advantage of distance fields for their compact renderers. The fact that geometry can be represented and rendered through distance functions allowed for a considerable reduction in memory usage. Complex objects and scenes can be created through blending of multiple distance functions into a coherent distance field [10].

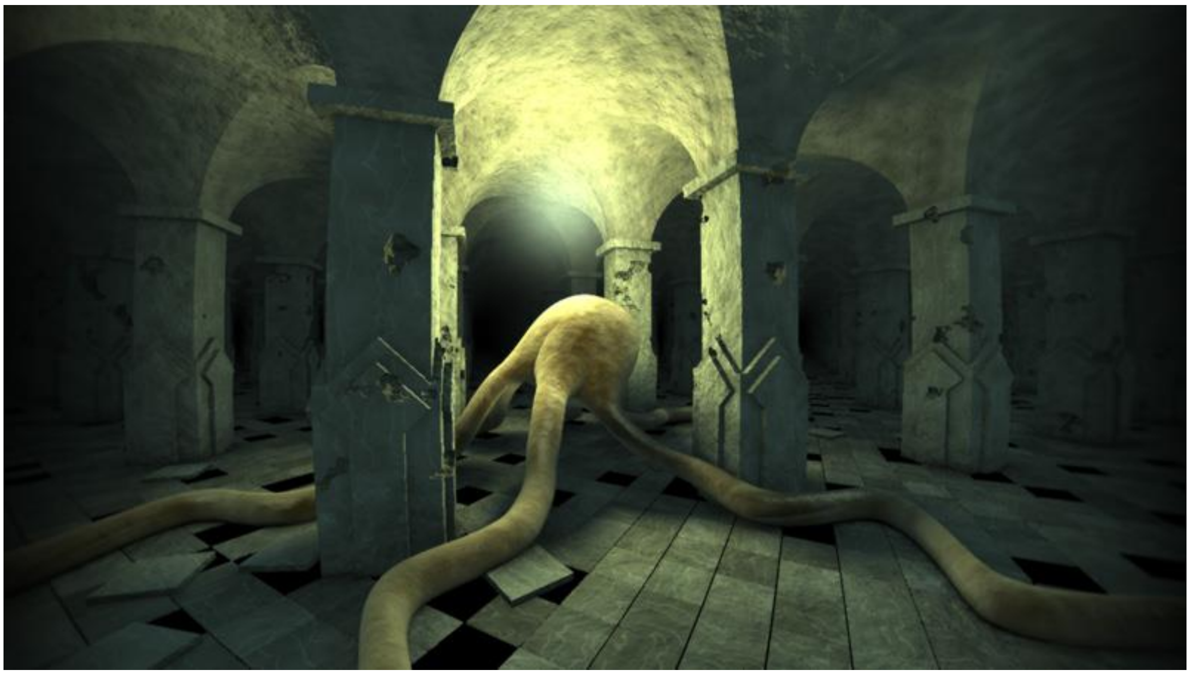


Figure 4 slisesix by Iñigo Quilez. A 4k demo, rendered by marching a distance field consisting of multiple primitives [10].

Another use of distance fields is found in the area of collision detection. Precomputed SDFs of static geometry can be used to quickly determine collisions, and are described for cloth simulation in 2003 by Fuhrmann et al [11].

Even though distance fields were being used in a scientific context, it wasn't until 2007 that they found their way into commercial videogames. Valve Corporation proposed the use of 2D SDFs for font rendering, allowing for the preservation of edge information in highly compressed textures. With their technique font images can be reduced up to 64 times and still maintain sharp edges without a significant performance overhead. A conventional image downscaled by this amount would result in a lot of blur, which would be unpleasant in a videogame [12].

The two most important metrics for the performance of today's videogames are their framerates as well as their memory consumption. In order to reach a framerate of 60 fps all game-related rendering, calculations, updates, physics and animations have to be finished in 16.6ms. In keeping with Moore's law, generally speaking the speed of microprocessors has doubled every two years, while the speed of memory has lagged far behind [13]. Therefore is it crucial for modern high-performance games to reduce their memory footprint as much as possible, such as by keeping texture sizes small. However, the reduction of texture size is often accompanied with blurriness and uneven edges, as interpolation algorithms blend multiple image pixels into one, which results in a loss of continuity and smoothness of the surface normals.

Distance fields can remedy this problem: as shown in Figure 3, a high-resolution image can be converted into a much smaller SDF representation. The distance field has a smooth

gradient across the object edge, therefore normals can be recalculated by following the gradient from an edge point. When rendering the distance field, the resulting edges are much sharper than in the reduced reference image. It is very simple to smooth the edges further, by applying simple anti-aliasing in a fragment shader, as described by Green [12]. With antialiasing, distance field textures have a similar rendering quality to high-definition images, while reducing file sizes by a considerable amount (32x-64x).
As with all things, this technique has its limitations: The SDFs do not hold any color information whatsoever, therefore only monochromatic images can be compressed this way.

4 Related Work

Even though distance fields have a long history in computing, there are only a few scientific works that apply these principles to modern real-time rendering. Many of the now common distance field effects were popularized by Inigo Quilez [10], but were mostly contained to the demo scene. The main contributions towards the techniques and optimizations presented in this thesis came from two SIGGRAPH 2015 presentations. Wright's 'Dynamic Occlusion with Signed Distance Fields' presents a modern approach to distance field effects. It shows the viability of distance fields as an enhancement to regular mesh rendering in the context of the Unreal Engine 4 [14]. The other contributing work was Evans' "Learning from failure: A Survey of Promising, Unconventional and Mostly Abandoned Renderers for 'Dreams PS4' ,a Geometrically Dense, Painterly UGC Game" which describes a variety of different unconventional distance field rendering techniques [15]. This thesis aims to merge and extend upon the techniques presented in these papers in order to create a modular distance field renderer.

5 Generating Distance Fields

All distance field techniques presented in this thesis need to poll values of distances multiple times per execution, and as such we require a well-defined function for generating such values for arbitrary positions in space. A wide variety of these functions exists for various platonic shapes, and more complex shapes can be created through Boolean operations. These distance functions create what are known as implicit surfaces and these implicit surfaces can even be formed from transformed explicit (mesh) surfaces. There are many different algorithms for achieving this transformation, but the inputs and outputs are universally the same. A 3D texture is spanned over the domain of the mesh, and each grid cell is populated by the closest distance to the mesh surface, as calculated by the algorithm. In essence, this is a way to cache a distance field of arbitrary computational complexity into a structure that can be accessed easily. Both implicit distance functions as well as explicit mesh conversion algorithms will be further described in the following chapters.

5.1 Implicit Surfaces

The easiest way to create distance fields is through a distance function. Many basic geometric shapes have distinct and well known distance functions, and more complex objects are aggregates of these functions. Therefore complex scenes can be created through the combination of a few simple platonic primitives. This is similar to Constructive Solid Geometry (CSG) which is the same principle for mesh creation. Distance functions have different calculation complexities, but have always a point as input and the distance as output. The simplest implicit distance function is that of a sphere, which can be seen in the GLSL code snippet below.

```
float sphere ( vec3 point,vec3 center, float radius)
{
return length(point-center)-radius;
}
```

It is easy to see why this is the case: the distance from the surface of a sphere is just the distance of two points subtracted by the sphere radius. The sphere is an essential structure in distance fields, as every value returned from a distance function can be seen as the radius of an unbounding sphere placed at the specified point. This property will prove essential to the sphere tracing algorithm presented in chapter 6.2.
In order to combine the distance fields of multiple primitives, a variety of functions can be used. The most common of which are Boolean operations like union, subtraction and intersection. These have the useful property of preserving Lipschitz continuity, therefore

maintaining correct gradients and distances across the domain. The following code snipped demonstrates the union function.

```
float union( float distance1, float distance2 )
{
return min(distance1,distance2);
}
```

Other, more complex blending functions exist which allow for interesting effects such as soft blending, but caution has to be taken as these do not preserve distance field continuity. A good reference source for a variety of shapes, domain and blending functions is HG_SDF, a distance field library from the Mercury demo team[3].

After creating multiple implicit primitives, a 3D texture can be populated with distance values, by querying the distance functions for each discrete grid element, and combining the values with Boolean functions. As most distance functions themselves are quite small and fast to compute, the computation of distances could even be performed at run-time, by simply calling the functions when needed. This approach is widely used in the graphics demo scene as no mesh data has to be saved, resulting in tiny application sizes[4].

5.2 Explicit Surfaces

Unfortunately, not every shape can be generated by an implicit function easily. A general distance function exists, but it is only suited for convex shapes, and is much slower than specialized primitive functions [16]. Concave shapes still prove difficult, and have to be generated as aggregates of simpler primitives.

Almost every game developed today uses triangle meshes for geometry and world representation. This stems from the highly optimized vertex transformation and rendering pipelines on modern GPUs. The transformation of mesh geometry into a distance field representation can be achieved through distance transform algorithms. The problem of detecting the closest point of a mesh is non-trivial, and only feasible in real time for small grid sizes [2].

A wide array of different distance transform algorithms have emerged over the past 30 years. The naive approach for distance calculation is the brute force algorithm. For each grid cell, the distance to each triangle of the mesh is plotted. The shortest resulting distance is used as the field value. Even though this algorithm is very computationally expensive, it is guaranteed to yield correct results. Spatial data structures like Quadtrees can be used to speed up the computations [17] and the algorithm can be parallelized easily which makes it a good fit for GPU calculations. Faster algorithms exist, but they work on a

[3] http://mercury.sexy/hg_sdf/

[4] http://www.shadertoy.com

speed/accuracy tradeoff basis. Two very popular algorithms are the Chamfer Distance Transform, and scan conversion methods. These algorithms will be further highlighted in this chapter.

5.2.1 Brute Force

The brute force algorithm is the most straight-forward of the commonly used distance transform algorithms. Initially the extents of the distance field grid and the distance between each grid element have to be chosen. Choosing a cube with equal dimensions (E.g. 64x64x64) leads to a lot of empty cells. The grid dimensions can be compacted by analyzing the to-be-converted mesh and scaling the dimensions according to the extents of the mesh. The same approach can be used to determine the cell-to-cell distance, except that it has to be identical for all spatial dimensions. If the scale of the grid cells would not be uniform, the grid cells themselves would stop being regular cubes, which would break the possibility of linearly interpolating between the cells. The dynamic grid size calculation is important to fit the grid as tightly as possible around the mesh, while reducing the overall memory footprint.
After these setup steps, the algorithm described below can be executed.

```
Foreach GridCell in DistanceFieldGrid:
      Foreach Triangle in Mesh:
            Find distance from Triangle to GridCell, save the closest one
```

This algorithm is very simple and can be parallelized easily: when executed on a multicore CPU each processor can focus on a small part of the grid.
This algorithm is used by Epic in their Unreal Engine 4[5] for mesh to distance field conversion. To speed up the computation they split the workload onto multiple processors, if available, and employ kd-trees as a spatial speedup structure, as suggested by Strain et al [17].
For our distance mesh conversion tool in the mTec renderer, a different approach was chosen: Instead of relying on the CPU for the brute force calculation, the work is relayed to the GPU, which is very suitable for these kind of computations with its highly parallel SIMD processing units. The algorithm is implemented in an OpenGL compute shader and each shader unit calculates a different grid cell. For further acceleration, a preprocessing step is introduced that calculates per-triangle data required for the distance and sign computation. As no state is shared between the each shader invocation this per-triangle data would have to be calculated again and again for each triangle. The preprocessing step is performed in a compute shader as well, and speeds up the overall computation by over 200%. Our conversion tool performs all the necessary calculations entirely on the GPU and

[5] Unreal Engine 4.10.4 was used for the analysis as well as all distance field calculation tests

can use any mesh data, as long as the mesh is well-formed and closed. This is a general problem with all distance transform algorithms, as no clear “inside” and “outside” for the sign computation can be determined, if the mesh is not a closed surface. In such a case, the mesh has to be repaired first, which can be done dynamically [18]. Other speedups used in our tool are squared distances for comparison, which are based on a suggestion by Erleben et al. [2]. Further suggested speedups include the reversing of the algorithm by going through all triangles and computing the distance to all grid cells. This approach yields a speedup on the CPU, but cannot be efficiently used on the GPU due to the highly parallel nature of the shader units. Atomic write operations and memory barriers would have to be used to ensure coherency when writing to the distance grid but would also slow the algorithm down significantly. The highly parallel nature of shader units also prevents the intuitive use of CPU-based spatial data structures such as kd-trees for this algorithm.

Mesh	*Triangles*	*SDF Size*	*Ue4*	*mTec Convert*
Cube	12	24x24x24	1.9s	0.0072s
Teapot	2,464	128x74x89	118.4s	8.05s
Glass	5,040	79x96x79	104.7s	11.78s
Dragon	100,000	42x31x22	6.5s	9.97s

Table 1: performance comparisons between UE4 and the mTec converter. Times in Seconds.

Table 1 shows the distance field computation time of various meshes with our conversion tool, as compared to the Unreal Engine 4[6]. It can be seen that Unreal’s kd-tree accelerated CPU approach excels for large polygon counts, but is considerably slower when dealing with larger distance fields.

Our shader-based approach has a few practical limitations that should be noted. From Windows Vista onward, Microsoft employs a system to detect tasks that take long time to perform GPU calculations. This system is called Timeout Detection and Recovery (TDR) and is enabled on all Windows computers by default. When TDR detects that the GPU did not finish a dispatched operation within a certain timeframe (2 seconds is the preset value), the display driver is reinitialized and the GPU reset[7]. Although this is a useful mechanism for the average user to prevent freezes requiring a reboot, it is an obstacle for our purposes. When converting meshes containing a large amount of triangles, the computation can take longer than the preset threshold, therefore triggering TDR and crashing the display driver. Unfortunately it is not possible to disable TDR from the application itself. The only way to deactivate the system is to alter values in the Windows

[6] The Computer used for these measurements was running Windows 7 and had following hardware: Intel Core i5-2400 3.1 GHz Quadcore, AMD Radeon R9 200 Series, 8 GB RAM

[7] https://msdn.microsoft.com/en-us/windows/hardware/drivers/display/timeout-detection-and-recovery

registry. The key "TdrLevel" under "HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\GraphicsDrivers" has to be set to 0 in order to disable TDR entirely.

Another issue stems from the fact that there is a limitation on the maximal number of work groups that can be dispatched simultaneously. As defined by the OpenGL 4.3 core specification, the maximal work group count has to be at least 65535 per spatial dimension. This results in 2^48 work groups that can be dispatched simultaneously, which is more than enough for our purposes. Unfortunately this total amount cannot be used for a single dimension and has to be split over all three axes. The preprocessing shader is dispatched once for each triangle in the mesh, therefore the shader would not run for meshes that are larger than 2^16 triangles. In order to circumvent this limitation, the dispatch has to be split into multi-dimensional groups, even though only a one-dimensional operation is performed. The shader has to recalculate its correct index in the triangle list during computation.

5.2.1.1 Triangle-Point Distance Calculation

The most important step of the brute force algorithm is the actual distance calculation. Even though this might seem trivial, a point to triangle distance calculation is everything but. The closest point falls onto one of three characteristic elements of a triangle: points, edges and the face. The distance finding algorithm has to determine which triangle feature the grid cell is closest to, and then compute the distance between those two points. The most commonly used algorithm is laid out in the book "Real-Time Collision Detection" by Christer Ericson. At first, the grid cell point is projected onto the plane created by the triangle. Then the resulting projected point is converted to barycentric coordinates to determine if it lies within the triangle itself. If this is the case, the shortest distance to the triangle is simply the length of the vector created between the grid point and the projected point on the triangle. Further checks are necessary to find the edge or vertex closest to the projected point if this is not the case. When the closest feature is found, the distance of the grid cell point to this feature can be calculated easily.

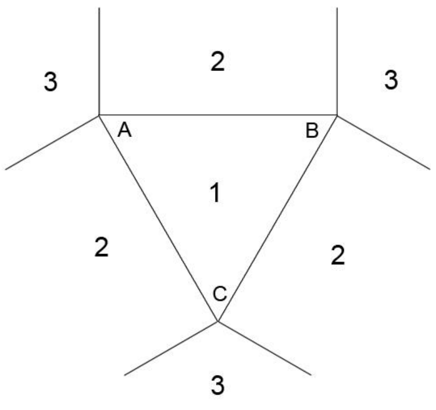


Figure 5: The three cases for a point-triangle distance. Closest to the triangle face (1), closest to an edge (2) and closest to a vertex (3).

A different algorithm is presented by Jones [19]. Their algorithm tries to reduce the problem to two dimensions, by rotating the triangle onto the yz-plane. A precomputation step is required to find a transformation matrix which performs the required rotation. When trying to find the distance of a point to the triangle, the point can be simply rotated by the same matrix as the triangle, and its distance to the triangle plane becomes its position on the x-axis. As with the 3D algorithm, special checks have to be performed in order to determine the closest feature of the triangle. Even though the paper states otherwise, in our implementation the 3D algorithm has been considerably faster than the 2D approach. Furthermore the large amount of rotations needed by the algorithm introduced a high numeric instability, which lead to a lot of inaccuracies in the distance field.

5.2.1.2 Computing the Sign

After computing the distance, the sign of the grid cell has to be determined. A negative sign means that the cell lies “inside” the given mesh. As “inside” is only well defined for closed meshes, the sign computation returns incorrect results on thin geometry. The Unreal Engine is using a kd-tree for their distance calculations which simplifies the sign computation, which can be performed by casting out rays and comparing the normals of the intersected triangles. If more than 50% of the hit triangles are facing the grid cell, the cell is “outside” and the distance has therefore a positive sign [14]. The facing of a triangle in relation to the point can be computed by taking the dot product of the triangles’ normal with the point-triangle ray.
On the GPU, all triangles are visible for the algorithm, and simply summing all triangle facings would yield incorrect results in all but the simplest cases.
The naïve approach would be to just use the closest triangle for sign computation. Unfortunately this also yields incorrect results in the case when an edge or a point are the closest features of a triangle, as each vertex is shared by at least three triangles in a well-formed mesh, while each edge is shared by two. As seen in Figure 6, if the algorithm would try to find the closest triangle both would qualify and just the triangle ordering would determine which triangle is used for sign computation, even though the latter would return a different sign.
This could be alleviated by using a pseudo-normal for the dot product instead of the actual normal of the triangle. This bent normal is simply the sum of the normals of all triangles which share the closest feature.

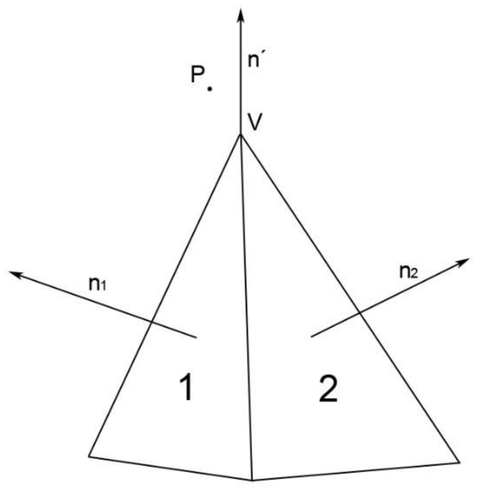


Figure 6: A point (P) lies closest to a vertex (V), which is shared by two triangles (1, 2). Both triangles are equally close but their normals face in different directions. A normal is chosen based on the enumeration order of the triangles, resulting in errors. A pseudo-normal (n´) can be constructed by summing the triangles’ normal, which reduces the number of incorrect results.

This modification alleviates the main problem, but is still insufficient in certain edge cases. A problem with these pseudo-normals arises when one of the faces of the mesh is tessellated into finer triangles, while still sharing the same vertex. The sum of all triangles would weigh the normal more into the direction of the face with more triangles, regardless how small they might be. This problem can be solved by the introduction of angle weighted pseudo-normals [20]. In addition to the summation of the normal, each normal has to be weighted by a value which amounts to the included angle of the two edges connecting to the closest vertex. This results in smaller weights for smaller triangles and larger weights for larger triangles (Figure 7).

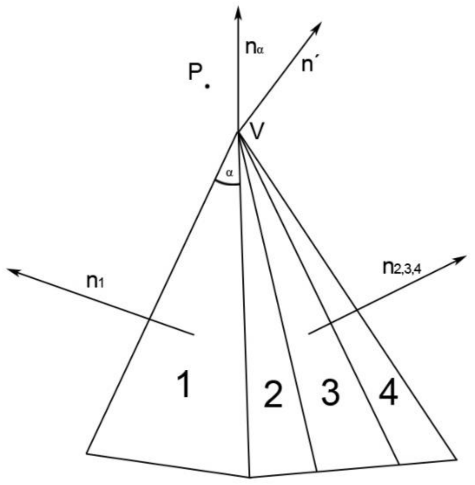


Figure 7: Triangle 2 from Figure 6 gets tessellated into triangles 2, 3, 4. As the Vertex (V) is now shared between 4 triangles, all the normal are summed for the pseudo-normal. This results in a skewed, incorrect normal (n´). To produce the correct, angle-weighted pseudo-normal (n_α), each normal (n) has to be weighted with its corresponding angle (α).

Even though this method of sign computation works perfectly in theory, it has a few problems in the actual implementation due to floating-point inaccuracies. As IEEE floating point numbers have a limited precision, small, adjacent triangles may result in slightly different distances, which would prevent their pseudo-normal merging. Therefore a threshold has to be introduced, so that all points and distances within a certain, small range are correctly merged together for sign computation. We use a ULP-based float comparison, as described by Dawson[8].

Max describes a more accurate sign computation for special cases in “Weights for Computing Vertex Normals from Facet Normals”, but their algorithm is much more computationally expensive and was ultimately not used in mTec [21].

5.2.2 Chamfer Distance Transform

A popular method for creating distance fields from images is the chamfer distance transform (CDT) algorithm. It uses a simple distance propagation scheme to generate the unsigned distance field of a monochromatic image. A 3x3 mask is moved over each pixel sequentially, propagating the shortest distances of its neighbors [22]. The entire distance field is created in two passes of moving the mask over the pixels, once from the top-left and the other from the bottom-right. As this process is inherently sequential it cannot be

[8] https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/

run on graphics hardware. The resulting distance field is not very accurate due to the crude summation of neighboring distances. This can be improved by an extension called the dead reckoning algorithm (DRA) [23]. In the classical CDT, when a neighboring value is lower than the current cell value, the cell is simply set to this neighbor's value plus the Euclidean distance to the neighbor. DRA improves this scheme by saving the neighbor that was chosen for each cell. When determining the value for a cell, another indirection is resolved by jumping to the closest neighbors' closest neighbor, and using that value for further calculations. This leads to much more accurate distance fields.
As CDT produces only unsigned distance fields, the base image has to be inverted and run again through the algorithm in order to produce an inverted unsigned field which can be composited into the final signed distance field.
Even though the algorithm is sequential, it is still fast enough for real-time computation of small distance fields, as shown by Kessler [24].

5.2.3 Scan Conversion Algorithms

The rise of programmable graphics hardware at the beginning of the 21st century led to the development of a wide variety of GPU accelerated distance transform algorithms [41][42][43]. These algorithms generally try to reduce the number of triangles that have to be checked during distance field calculation by generating a set of geometric prisms for the feature of each triangle (vertices, edges and faces). These prisms create a generalized Voronoi diagram, which means that all points within a prism are closer to its particular triangle feature than any other. These prisms can be rendered by the GPU which results in a fragment shader dispatch for each distance field cell contained by a prism. By definition the distance field of the region can be determined by calculating the distance between the grid cell and the prism feature. As multiple prisms may overlap, a depth testing scheme has to be used in order to preserve only the shortest distances. After running this algorithm, a single 2D distance field layer can be retrieved from the depth buffer [41]. By repeating this multiple times a full 3D field can be composited. Erleben et al. propose tetrahedra as a bounding volume, which reduces problematic cases of the algorithm, by directly computing the tetrahedron-layer intersections on the GPU side [2]. The parallel nature of these algorithms allows for distance field computation in real time. In most cases it is sufficient to just calculate distance fields a single time and cache it in a texture for later run-time use but specific cases such as dynamic destructible environments may require on-the-fly recalculation.

6 Rendering

A scene consisting of distance fields can be rendered with quite simple algorithms. This is done by finding the intersections of rays shot from the camera with the surface of the field, denoted by a distance of 0. The resulting points can be used to approximate the surface normal and the distance from the camera which are essential for further lighting calculations. Algorithms for intersection finding of rays with distance fields have been widely researched for the past 30 years. The three techniques most commonly used today ray marching, sphere tracing and cone tracing will be presented in this chapter. They pose a complete departure from the standard way that meshes have been rendered in the past decades, and have more in common with ray tracing than rasterization.

6.1 Ray Marching

In order to render an object that is represented by a distance field, a common technique is to cast a ray from the camera into the SDF, until a surface is hit [25] [26]. The information of this surface point can then be used for further calculations (normal, material). The intersection point of the ray with the implicit surface can be calculated by applying a root finding method. This is a nontrivial mathematical problem, as there is no general root finder for polynoms of arbitrary degree [27], which led to the development of the ray marching technique by Perlin and Hoffert [28]. The intersection point with an implicit surface can be determined by sampling the field in fixed steps, until a value of zero (the surface) is found.

```
For each pixel:
      samplingPoint=O;
      while(SDF(samplingPoint) > 0)
            samplingPoint += s*D;
```

This is the classic ray marching algorithm. SDF() is the function of the distance field, O and D are the origin and direction of the ray and s is a fixed step size.

The resulting samplingPoint will be an estimate of the intersection between the ray and the distance field object. In practical implementations more termination conditions are added to the inner loop. First, the distance travelled so far is tracked and the algorithm terminates if a predetermined maximum distance is reached. Secondly, a maximum number of steps is often enforced. This is done in order to ensure a termination of the loop in case the ray misses the object entirely.

This algorithm can be parallelized very easily by delegating the work to the GPU. The calculation has to be done for every pixel, so it is well fitted for a fragment shader. Each invocation casts a ray and performs lighting and texturing operations according to the resulting intersection point.

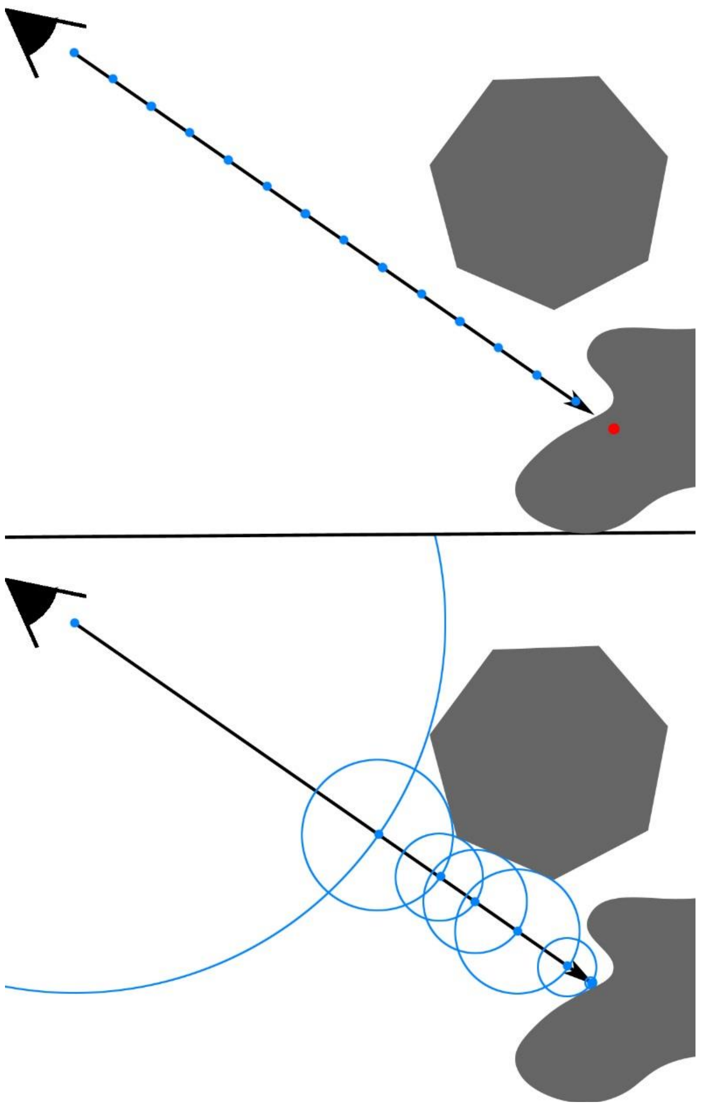


Figure 8: Comparison between the ray marching and sphere tracing algorithms. 15 fixed steps are needed to reach the object with ray marching, while sphere tracing reaches the surface in 7 steps.

Unfortunately the conventional ray marching algorithm has multiple issues: As the step size is fixed, the ray can overshoot thin objects if the chosen value is too large, or take a long time to traverse the field if the value is too small.

Various tweaks have been proposed for the algorithm in order to address these issues. If the algorithm determines that the currently sampled point lies within an object (returned distance is negative), the position of the point can be refined before returning it. This is done by reducing the step size and marching back on the ray until the sample point exits the object. This process can be repeated an arbitrary amount of times to get better and better approximations of the intersection point.

These refinements do not help with the potential huge amount of sample points that have to be checked in order to get a reasonable surface point and mitigate issues with thin objects.
The sphere tracing algorithm was developed as an extension of ray marching in order to address these issues by taking into account inherent properties of distance fields. Ray marching is still the go-to solution in all cases where intersections have to be found for a domain in which the polled function does not represent an actual distance field. A prominent example is surface finding in procedural, height-map based terrain.

6.2 Sphere Tracing

The main problems of ray marching, namely the fixed step size issue were addressed by sphere tracing, an extension of the original algorithm developed by Hart [27]. In sphere tracing the step size is dynamically adjusted along the marching path, by setting it to the sampled value from the distance field. As the SDF holds the distance to the nearest surface, stepping this amount along the ray will never overshoot an object and large empty spaces can be traversed quickly (Figure 8). The following code snipped showcases the sphere tracing algorithm.

```
For each pixel:
      samplingPoint=O;
      step = SDF(samplingPoint);
      while(step > 0)
            step=SDF(samplingPoint);
            samplingPoint += step*D;
```

Through this simple step size adjustment the issues of overstepping are fixed. Small objects cannot be missed as the step size will decrease automatically when near an object and no further refinements are necessary as the algorithm will never overstep into the object itself. This is of course only the case if the sampled function itself is a correct distance field. The algorithm takes its name from the virtual unbounding spheres that are created from the ray point and the distance value.
A disadvantage of this algorithm is the fact that the number of samples cannot be determined beforehand, as it is highly dependent upon the ray’s origin. If a ray passes parallel to an objects’ boundary, the step size is reduced and can incur a significant performance overhead. Therefore a maximum amount of steps should be defined as well as a minimum step size threshold [26].
A scene composed of distance fields can be rendered using sphere tracing. Rendering a 3D scene onto a 2D screen is a multi-step process, which, in modern computers, is governed by the GPU. 3D objects are represented in memory as triangles that form more complex meshes. These triangles are projected onto the screen through rasterization

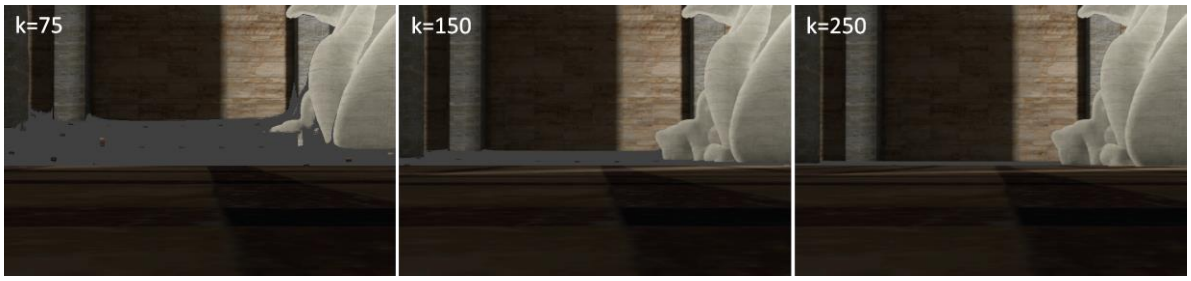


Figure 9: Sphere tracing artefacts at various max step sizes k. The gray color marks areas where no surface was found due to insufficient steps.

before being passed to the fragment shader stage. A shader is executed by the GPU for each pixel occupying each on-screen triangle [35]. The shader is provided with information about the position of the fragment and triangle-specific information, like the texture coordinates and normal. Its task is to determine the color of the fragment, which will be displayed on the screen in the end, baring a discard from the following depth test and blending stage.

SDF rendering through sphere tracing, as proposed by Hart can be done entirely in the fragment shader stage [27]. A quad covering the entire screen is passed into the rendering pipeline, ensuring that a fragment shader is executed for each screen pixel. Once the shader stage is reached, it has to be determined which object should be rendered for the current pixel. This is done by sphere tracing a ray from the position of the camera into the scene and finding its intersection with a distance field. Once an intersection point is found, lighting and a color can be calculated for the pixel. The algorithm is fairly simple and can be implemented easily. Even though results can be achieved fast, the algorithm has some issues which become apparent when rendering more complex scenes.

```
float spheretrace(vec3 p, vec3 d, float push, float max, float threshold)
{
      float T = push;
      vec3 P = p + d*T;
      float steps;
      for (int i = 0; i<max; i++)
      {
            float S = GetSdfDist(P);
            T += S;
            P = d*T;
            if ((T > MAX_DIST) || (S < threshold)) {
                  break;
            }
      }
```

```
	return T;
}
```

This is the HLSL code of the actual sphere tracing implementation used in our renderer, baring a few additional optimizations described more in-depth in the implementation chapter.

In many cases, the intersection point returned by sphere tracing is only an estimation of the mathematically correct intersection, which could be acquired by root finding. The amount stepped on the ray is dependent on the value of the distance field at the current location. This results in a different amount of iterations needed to intersect the object, based on the direction of the ray and the surrounding distance fields, and can become problematic when a ray passes close by an object without actually hitting it. The step size becomes very small and lots of cycles are wasted just to pass by the object. A maximum amount of steps has to be defined in the algorithm, otherwise some pixels might take orders of magnitude longer to find an intersection than others. If a ray is marched along the surface of an object, it may run out of steps, before finding an intersection point. This results in very characteristic artefacts, which are especially visible when the camera is looking parallel to a surface (Figure 9).

The issues are reduced by choosing a larger step threshold, but remain inherent to the algorithm. Increasing this threshold slows the whole algorithm down significantly and the correct value should be chosen carefully. Many distance field renderers in the visual demo scene try to hide sphere tracing artefacts by applying distance fog to the scene, which works well but is obviously just a workaround. The actual impact of this issue is highly dependent on the scene itself. It is far more apparent on a long flat plane than in a closed room, for example. In the distance field community ray marching and sphere tracing are often used interchangeably, which can lead to confusion when there is no clear indicator towards the referred algorithm. For the sake of clarity, this thesis will refer to these algorithms by their actual names.

6.3 Cone Tracing

Cone tracing is an extension of the sphere tracing algorithm. In addition to the ray, a cone is created and expanded in the marching direction. At each sampling step, the current cone radius is compared with the minimum distance from the distance field. If the SDF value is smaller than the current radius, the cone has intersected an object [14]. This incurs very little additional cost to the normal sphere trace, while providing useful intersection information.

6.4 Field Visualization

When developing distance field renderers, it is very helpful to have different methods of visualizing the distance field for debugging purposes. These visualizations can help in quickly identifying underlying issues in a more intuitive way. The most basic way to display a distance field is completely independent of ray marching. By transforming the screen coordinate of each pixel into the [0,1] range and plugging the resulting value as a point into the distance function, a 2D SDF can be visualized easily (Figure 10). The same method can be applied to 3D fields by looking at single slices over a given timeframe. This is very similar to computer tomography imaging and can be very useful for the verification of the correctness of the field itself, before plugging it into larger scenes.

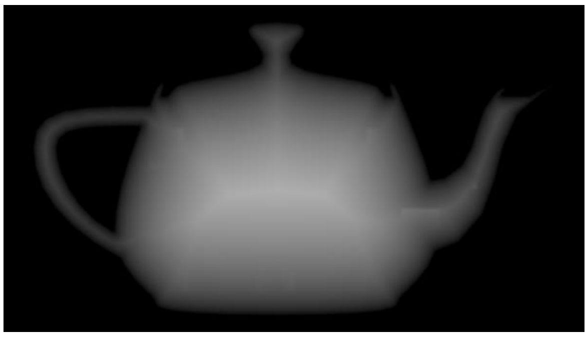


Figure 10: Slice through a distance field.

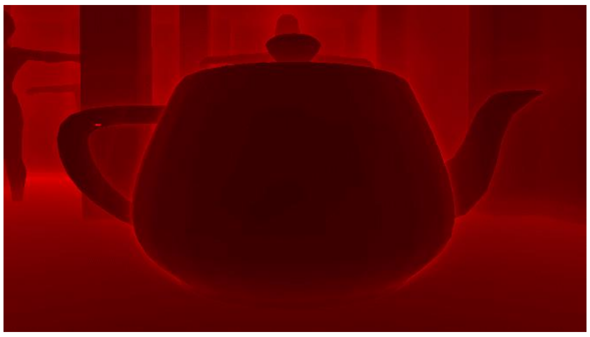


Figure 11: Step count visualization.

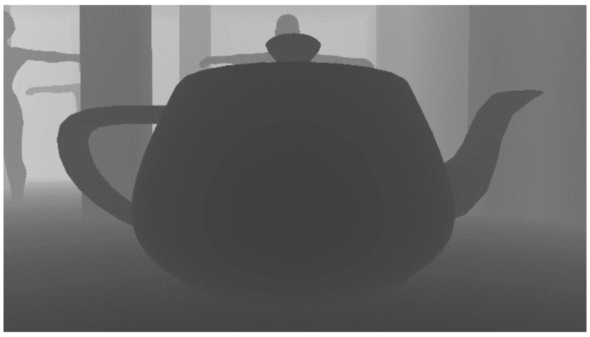


Figure 12: Depth visualization.

Other ways of visualization use the data produced by the sphere tracing algorithm. The scene depth can be displayed by calculating the length between the ray origin and the intersection point, and normalizing it to a [0, 1] range. The resulting image is very similar to a regular z-buffer except that the values will be distributed linearly. Finally, a very interesting way of visualizing the scene can be achieved by recording the number of steps that were taken on each ray before hitting an object. This image will have bright auras around the silhouettes of objects, as the step sizes will get very small in those areas. This visualization can help in identifying choke-points in the scene where artefacts could occur.

6.5 Material Management

In order to map a specific texture or lighting parameters onto each rendered pixel, information about which pixel corresponds to which individual object is required. During sphere tracing all individual distance fields are merged through Boolean function to a single output value, resulting in the loss of all material information. The material id can be retrieved by performing a single lookup of the sphere traced intersection point and returning a unique id of the closest object instead of the distance. This id can then be used as an index to fetch relevant information or textures from a buffer. Because the classic full-screen shader approach renders everything in a single pass, there is no easy way of using different lighting functions for different materials. By fetching material information from a buffer, the color tint and specular parameters can change, but the processing function itself would always stay the same. GLSL is a very basic language which does not support advanced features such as function pointers or polymorphy. In order to get different lighting calculations for different materials smaller sphere tracing renderers often resort to switch statements that call different functions depending on the material. This problem can be addressed by using the rasterized sphere tracing technique presented later in this thesis: as each object can be rendered in a separate draw call, the tracing shader itself could be unique for each material.

6.6 Texturing

Texture mapping has been used for the past 20 years in order to increase the visual fidelity of a drawn mesh in triangle-based renderers. This is a multi-step process, which starts by pre-calculating a projection of a flat texture onto the 3D triangle mesh. Each vertex of the mesh is assigned a Texture coordinate (also called uv coordinate), which is a 2-dimensional vector on the range [0,1]. During the rendering of a frame, the uv values of all visible triangles are interpolated for each fragment and provided to the fragment shader. Sampling of a texture at the correct position can then be performed in the shader according to the interpolated uv's [29].

Unfortunately, regular, uv-based texture mapping cannot be used in distance field rendering. SDF's only contain information about the surface of the object, so additional properties like texture coordinates are not preserved. This leaves few other possibilities for texture mapping. The Playstation 4 game "Dreams" by MediaMolecule, which uses a distance field based renderer resorts to flat coloring for each object in the scene, skipping the problem of texturing altogether [15]. Our renderer employs tri-planar projections for texture mapping. This technique was developed by Geiss and Thompson for the Nvidia "Cascades" demo[9]. For tri-planar mapping, a total of three samples has to be taken from one or more textures. The textures are sampled by taking the world space position of the current fragment and performing a lookup in each spatial plane of the vector. In essence, this results in a 3-dimensional planar projection of the texture onto the object. The three resulting albedo colors are then blended according to the surface normal at the fragment. Tri-planar mapping is most commonly used for texturing of natural terrains and geographic

[9] https://www.slideshare.net/icastano/cascades-demo-secrets

features as it produces very natural-looking, continuous colors over surfaces. It is not very well suited for texturing of objects where the exact positioning of the texture is important. For example, the texturing of a model of a human face would be problematic for the technique, as there are well-defined areas (lips, eyes, eyebrows) that require very specific colors.

6.7 Normal Calculation

The surface normal is an essential information required for many shading algorithms. In conventional mesh rendering a normal are stored with each vertex, or can be calculated for each triangle through the cross product of the vertices. This approach is not possible with distance fields, but there are other algorithms which allow for the approximation of the surface normal. Our renderer uses the gradient computation technique, as described by Hart et al [5]. This technique can be seen in the formula below.

$$NX = F(PX-E,Y,Z) - F(PX+E,Y,Z)$$
$$NY = F(PX,Y-E,Z) - F(PX,Y+E,Z)$$
$$NZ = F(PX,Y,Z-E) - F(PX,Y,Z+E)$$

In order to approximate the normal, the field has to be polled at least six times, two times in each axis. More frequent polling can be performed in order to increase the precision of the algorithm, but this decreases the run-time performance, as every shader texture lookup is an expensive operation. For that reason our renderer performs the minimal amount of lookups. The constant factor E is used to determine the offset around the point for polling: increasing the value of E results in smoothed normals over an object's surface.
In conventional engines, rendering speed is determined by the amount of triangles that have to be culled, rasterized and drawn. To simulate a more detailed, rough, surface on otherwise flat triangles game engines often employ a technique called normal or bump mapping. Normal mapping modifies the normal at each rendered fragment by a value read from a separate texture called a normal map. This gives triangles non-uniform normal which matches the apparent surface albedo texture more closely. This bent normal can then be used in lighting calculations to increase the visual fidelity of the image [30].
In order to use normal mapping with distance field rendering, the original technique has to be altered slightly. This stems from the fact that we perform tri-planar texturing to get our albedo color for each pixel. The modified technique requires polling the normal map three times, at the same positions that were used for initial texturing. These values can then be blended together with our initial normal which results in a new, bent normal [31].
One could argue that despite needing three texture lookups the technique is simpler for tri-planar texturing than for regular uv-mapping: no values need to be precomputed, and the only inputs needed by the algorithm are the texture, the normal and the point in world space. This is compared to the most widely used normal mapping algorithm, which

requires the pre-computation of tangents and the transformation of values from tangent to object space [30].

6.8 Meshing

The conversion between mesh geometry and distance fields is not a one-way road. Similarly to distance transform algorithms, a variety of polygonisation algorithms have been described over the years. These allow for the transformation of implicit surfaces to a standard triangle mesh. The most popular algorithm in this category is the marching cubes algorithm. It was first presented by Lorensen and Cline in 1987 [32] for applications in the visualization of computer tomography data. When transforming a field with the original, non-modified version of marching cubes, the distance field has to be overlaid with a 3D-grid. Each grid cell is evaluated by sampling the distance function at the eight corners of the cell and comparing the number and position of corner points that lie within the object with a lookup table. This lookup table contains a set of well-defined triangle positions which are used to create new triangles for the grid cell. By processing the whole grid this way a triangle mesh can be created from the distance field. It does not really matter if the sampled field is created from a 3D texture or an implicit surface function, the only difference being that an implicit function is continuous and could be used to create very high resolution meshes. Evans describes that using polygonised distance field meshes can be problematic in game development. In order to achieve an acceptable level of detail the meshes become very dense, with lots and lots of tiny triangles [15].

7 Distance Field Effects

Beyond determining the visible elements of a given scene, distance fields prove quite useful for the calculation of other classical real-time rendering effects, with the most common being presented in this chapter. This should in no way be seen as an exhaustive list of possible effects as many techniques are still being discovered today. The main motor for the development of distance field techniques is the visual demo scene and websites such as shadertoy[10]. Soft shadows and ambient occlusion are the two oldest and most popular techniques. These are well established in the distance field community and have been described in many scientific works [10] [26] [33]. The reason for their popularity lies in the simplicity of the underlying algorithms, as well as their inherent importance for realistic rendering. Other techniques that will be presented in the following chapters include sub-surface scattering, reflections and approaches for distance field animations.

7.1 Shadows

In computer graphics shadows add a lot of realism to a scene. In the real world, shadows are the result of less photons being bounced and reaching the eye from a specific area compared to its surroundings. This is most often the case when another object occludes the direct path towards a light source. Even though this sounds simple in theory, there are many more factors which make physically-correct shadow calculation not feasible in today’s real-time rendering applications. Correct real-world lighting is described with the lighting equation [34]. With current computational means it is not possible to solve this equation efficiently in real time, as the sheer amount of photon data is too large to simulate. Therefore different techniques have been developed over the years to simulate shadows. In conventional, polygon real-time renderers the de-facto standard for shadow simulation is the shadow mapping technique as it is fast and well suited for graphics hardware.

[10] http://www.shadertoy.com

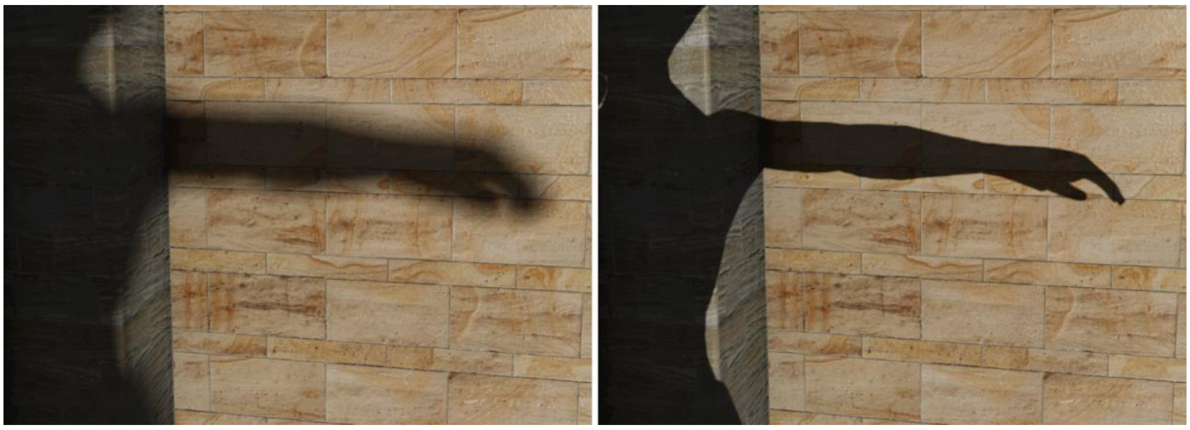


Figure 13: Soft and hard shadows.

When using distance fields, shadow calculation becomes very straight-forward. In order to determine if a pixel lies in shadow, a ray has to be traced from the pixel position in world space towards the light. If the ray hits an object before reaching the light, the pixel is occluded. There are different methods of getting a pixel's world-space position, as it depends on which type of renderer is used. Conventional renderers can use different methods of pixel reprojection to get its position from the depth buffer. Distance field renderers acquire this information automatically after tracing the scene and shadowing could even be performed in the same pass. In order to create approximated soft shadows, the scene needs to be cone traced from each pixel's position towards the light. This trace returns information how close the ray has passed an occluder and a penumbra can be created by darkening the pixel according to that proximity information. This technique is demonstrated as an alternative to cascaded shadow maps by Wright [14]. The algorithm for calculation soft shadows can be seen in the code snippet below and is just a slight modification to regular sphere tracing.

```
float SoftShadow(vec3 p, vec3 d, float maxSteps, float cutoff, float k,
float push, float distanceToLight, float maxDist)
{
      p =p+d*push;
      float radius = push;
      for (int i = 0; i<maxSteps; i++)
            {
                  float S = GetSdfDist(p);
                  closestPass = min(closestPass, (k*S / radius));
                  radius += clamp(S, 0.02, 0.1);
                  dist +=  S;
                  p += d* S;
                  if (dist >= distanceToLight ||
                       dist>maxDist ||
                         S < cutoff ) {
                        break;
```

```
                    }
            }
return clamp(closestPass, 0, 1);
}
```

Practical implementation of Inigo Quilez‘ soft shadow formula [10] in GLSL. This is the basis which is used and expanded in our mTec renderer. p and d are the basis for the traced ray, the point to be shaded and the normalized vector towards the light source. maxSteps, cutoff and maxDist are the regular termination parameters from sphere tracing. They are extended by another parameter, distanceToLight, which is essentially just the length of the point-light vector. The push parameter is used to start sphere tracing a small distance away from the actual surface. Without it, the algorithm could get stuck as the first position might lie on the surface, thus terminating instantly. Finally the parameter k is used to control the softness of the resulting shadow. Small values between 2 and 20 result in very soft shadows.

A very distinctive artefact can be observed when the shadows of multiple objects intersect. The shadow of the object further away from the light source has a clearly visible bright outline. This is caused by the cone traces from this border region needing to pass close to the first object before reaching the second object. As shadow cone traces are performed with a much smaller maximum step threshold, the cone trace might terminate before reaching the second object, which would cause the shadow to appear as if there was no second object at all. The problem can be alleviated by increasing the maximum amount of steps for the cone trace, but this also leads to much slower algorithm executions, without solving the underlying issue. Figure 14 demonstrates the artefact for varying amounts of steps. A possible solution for keeping the step count low while reducing the visibility of the artefact is to force a fixed minimal step size after a certain distance. This can help in passing choke points, but might cause incorrect shadowing with thin objects. Unfortunately no optimal solution for this problem has yet been found, and an appropriate workaround has to be chosen based on the scene geometry.

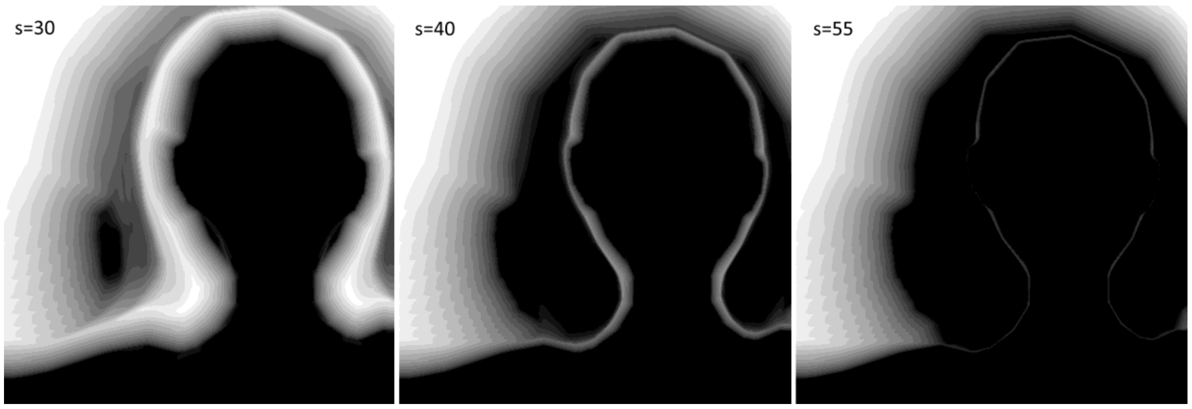


Figure 14: The shadow silhouette artefact at various step counts.

7.2 Ambient Occlusion

Ambient occlusion tries to simulate ambient lighting in a rendered scene. In the real world, ambient lighting occurs due to photons bouncing off of surfaces, thus reaching areas which are not in direct sunlight. With this effect, rooms can be lit entirely by indirect lighting and shadows can have varying degrees of darkness. Simulating this effect in computer graphics would require tracing an enormous amount of photons, which is not feasible in real-time. Ambient lighting is a global effect and independent of the viewer's position or look direction. It can be simulated by assuming that a surface is reached by more photons, the more it is exposed to its surroundings. Therefore, the floor of a room should be brighter than a corner. This occlusion can be precomputed offline for static objects and stored in a texture for run-time lookup [35]. Most real-time rendering applications feature dynamic geometry and animated characters, where prebaked ambient occlusion is of limited use. The need for fast approximation of AO lead to the development of screen-space ambient occlusion (SSAO) [36]. SSAO generates occlusion information by generating a number of points in a sphere around a pixel, and then checking them against the z-buffer. All points failing the z-test are assumed to lie inside of geometry. The results for each pixel are summed and averaged, which results in a rough AO approximation. SSAO was further expanded by Ritschel et al. with their introduction of screen-space directional occlusion (SSDO), which takes into account the direction of incoming light as well as a single bounce of indirect illumination [37]. These technique are fast and easy to integrate into existing renderers, but have limitations stemming from the fact that they operate solely in screen-space. No information about the actual geometry is known, and artefacts occur at the screen borders, due to the inability to sample points outside the screen-sized z-buffer.

In recent years, different techniques have arisen to calculate ambient occlusion through distance fields. They offer the advantage of being based on actual geometric information, while remaining very simple in principle. Two established algorithms, one by Evans [38], one by Wright [14] will be compared in this chapter. Furthermore we will present a modification to Quilez' algorithm, which reduces incorrect occlusion in concave geometry.

7.2.1 5-Tap Ambient Occlusion

This algorithm was presented by Evans [38] and later Quilez [10]. It is a very simple algorithm which results in believable ambient occlusion terms. The AO factor is determined by tracing a cone in the normal direction of the pixel. At each step the current stepped distance is compared to a sample from the distance field and the difference between these values is weighted and added to the occlusion factor. The weight decreases the farther away we step from the original point, in order to ensure that closer objects contribute more occlusion. A maximum of 5 steps is performed before the algorithm terminates. The resulting algorithm is quite fast, as only one distance field check has to be performed at every step. Unfortunately the resulting AO factor is incorrect for concave objects. This is especially visible on right-angled wall corners, which appear to have a bright stripe in the corner while darker in the surrounding areas. The reason for this error lies in the fact that the normal of points lying exactly on corners will split the corner angle exactly in half. When tracing along this normal, only a minimal amount of the surrounding walls will be hit. Fortunately this artefact is barely visible in a fully shaded and lighted scene. Therefore this algorithm is still a valid and cheap approach to distance field ambient occlusion.

7.2.2 Multi-Ray 5 Tap Ambient Occlusion

The desire to improve the quality of the 5 tap approach while still maintaining fast calculation times led to the development of the multi-ray 5 tap algorithm. It extends Quilez' algorithm by performing the cone tracing in multiple directions. In addition to the normal-aligned cone, 4 additional cones are traced in a normal-oriented hemisphere. These additional cones mitigate the corner-wall artefacts and provide more accurate ambient factors while still maintaining reasonable calculation times. The rays can be distributed evenly across the surface of the hemisphere through spherical Fibonacci mapping, which yields more natural occlusion results than fixed angle rays [45].

7.2.3 Fully Cone-Traced Ambient Occlusion

This approach to ambient occlusion was first presented by Wright [14], and is being used in the Unreal Engine 4. It can be seen as an extension to the 5-tap multi-ray approach. In order to calculate the AO factors, 9 cones are traced in a normal-oriented hemisphere. Our tests gave reasonable results with 7 taps and a maximum cut-off distance of 2 units. The algorithm features many parameters that can be tweaked easily in order to create ambient occlusion of different intensities and dimensions. The runtime costs of this algorithm are the highest of all the presented algorithms, which stems from the fact that a lot of distance field lookups have to be performed, but it is also the most physically correct approach. A comparison of all three algorithms can be seen in Figure 15.

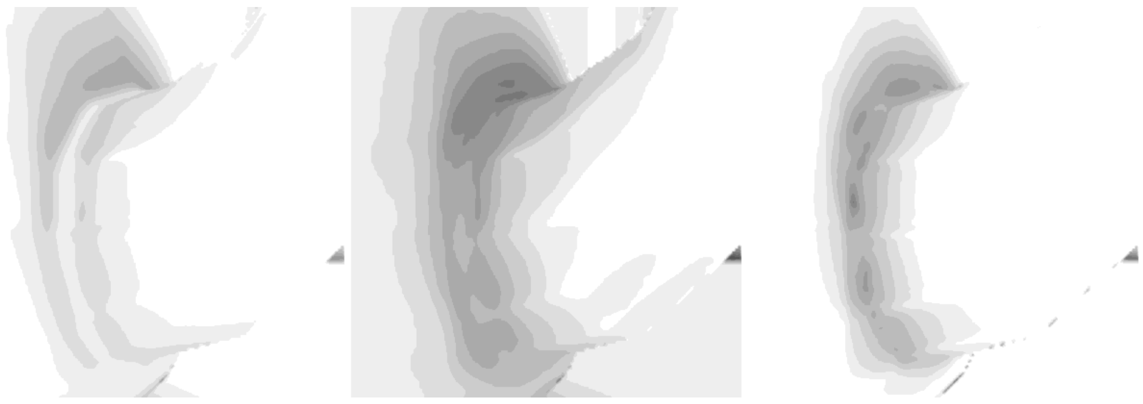


Figure 15: Comparison of the three presented AO techniques. All images have their contrast increased to make the AO influences more apparent. Left Image: 5-Tap, the incorrect occlusion at the corner is clearly visible. Center image: Multi-Ray 5-Tap. Right Image: cone traced AO.

7.3 Sub-Surface Scattering

Sub-surface scattering (SSS) is an effect that occurs naturally when light hits a specific surface such as human skin or marble. In this case, the incident light penetrates the surface rather than getting scattered immediately. The light is then absorbed or reflected from inside the material, which results in a soft, blurred look of the surface [46]. This effect is very desirable for modern videogames as it significantly increases the realism of human characters. In conventional real-time subsurface scattering algorithms, the soft light look is often simulated by blending of different colors or textures based on the incident light vector and the thickness of the object. There are different methods for determining the thickness of an object, but this can be roughly estimated by a separate thickness texture map, or through the pre-baked AO map [39].

A more accurate thickness approximation can be produced by ray marching a distance field. After determining the visible surface through sphere tracing or conventional rasterization, a ray can be traced into the object. The tracing is terminated as soon as a predefined maximum depth, step count or another surface is reached. The resulting distance that was traversed inside the object can then be used as thickness for SSS calculations. The aforementioned maximum thickness is a property of the object itself that simulates how deep the light can penetrate into the object. The lower this parameter is, the more solid an object will appear.

This algorithm can be extended in order to allow for correct thickness of concave or hollow objects. Rather than terminating on a surface, the tracing could continue and only add the distance traversed while inside the object (distance that has a negative sign). With this modification the sphere tracing algorithm has to be given a non-zero minimal step size, as no further progress would be made on reaching a surface.

The resulting thickness estimate can then be plugged into an arbitrary SSS lighting algorithm to get a nice translucency effect.

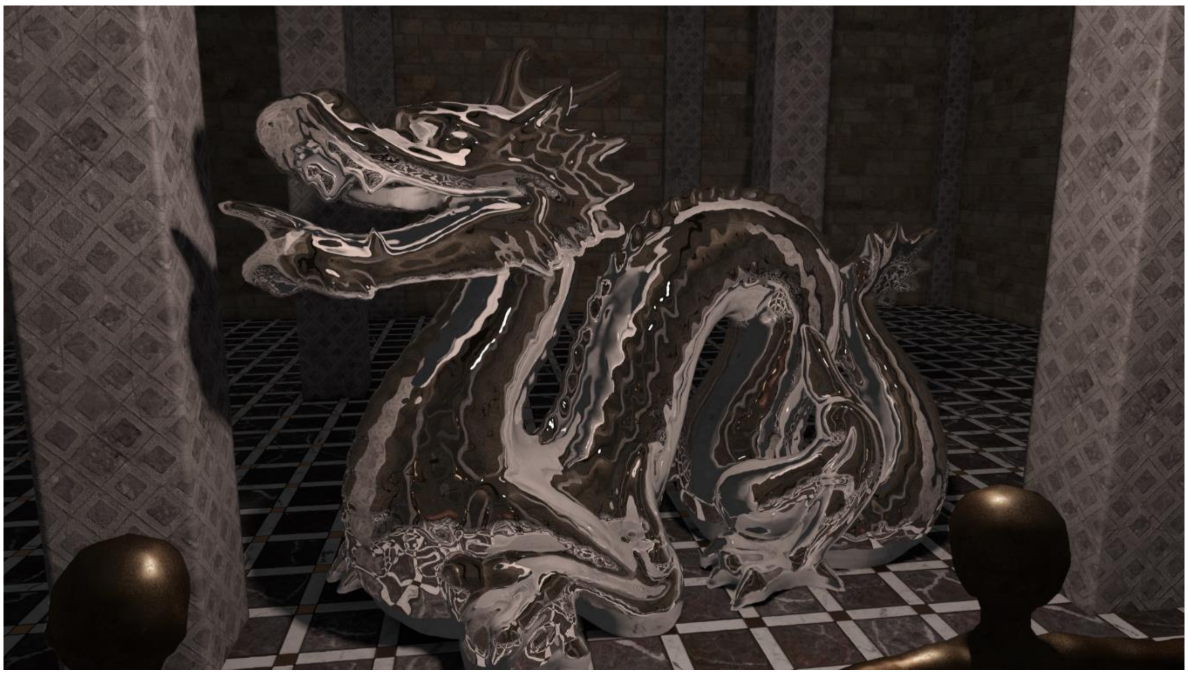


Figure 16: Fully reflective material through a single sphere traced light bounce.

7.4 Animation

Common matrix transformations can be applied to distance field rendered geometry similarly to classical triangle meshes. There are two main differences to conventional rendering which should be kept in mind. First, the transformation matrix has to be applied during ray marching multiple times, at each distance field polling step. This is much more expensive than mesh rendering, where the matrix transformation is applied a single time in the vertex shader. Secondly, only uniform scaling is possible. Non uniform scaling would break the linear interpolation property of distance fields.

The most common animation approach for conventional meshes, skeletal animation, cannot be performed with distance fields. The main animation method for SDFs are blend shapes. Different animations can be generated by blending linearly between different distance fields. Even though this might seem very restrictive compared to triangles, quite unique effects are possible which would be very hard to achieve otherwise. Distance fields of arbitrary complexity can be morphed into each other with just a single line of code. The best results of morphing can be achieved by using convex objects. When morphing between concave objects, visible animation artefacts can occur, in which shapes start to materialize out of thin air.

7.5 Reflections

Reflections can be calculated easily with distance fields. In order to calculate physically correct reflections the reflected vector of every visible and reflection-enabled pixel needs to be calculated and then sphere traced. This requires the vector from the camera towards the pixel in world space, as well as the world space normal. The reflection of the incident vector along the normal can be calculated through the formula $r=i-2(n*i)n$ where i is the incident vector, and n the normalized normal vector. Luckily OpenGL as well as DirectX support this calculation as a build-in function in their respective shader languages. This reflection vector can then be used to sphere trace through the whole scene and to determine the contributing reflection color, which allows for fully dynamic reflections with arbitrary amounts of bounces. These calculations are more expensive than shadows or AO, as the point determined through the reflected trace still has to be textured, lighted and ideally also supplied with shadow and AO factors. This essentially means that the whole scene has to be rendered twice. This process could be optimized depending on the requirements of the scene. If all the reflections are glossy-diffuse in nature, the reflected scene could be calculated in much lower resolution, and post-processed with a blur to achieve believable results for a much lower cost.

8 Implementation

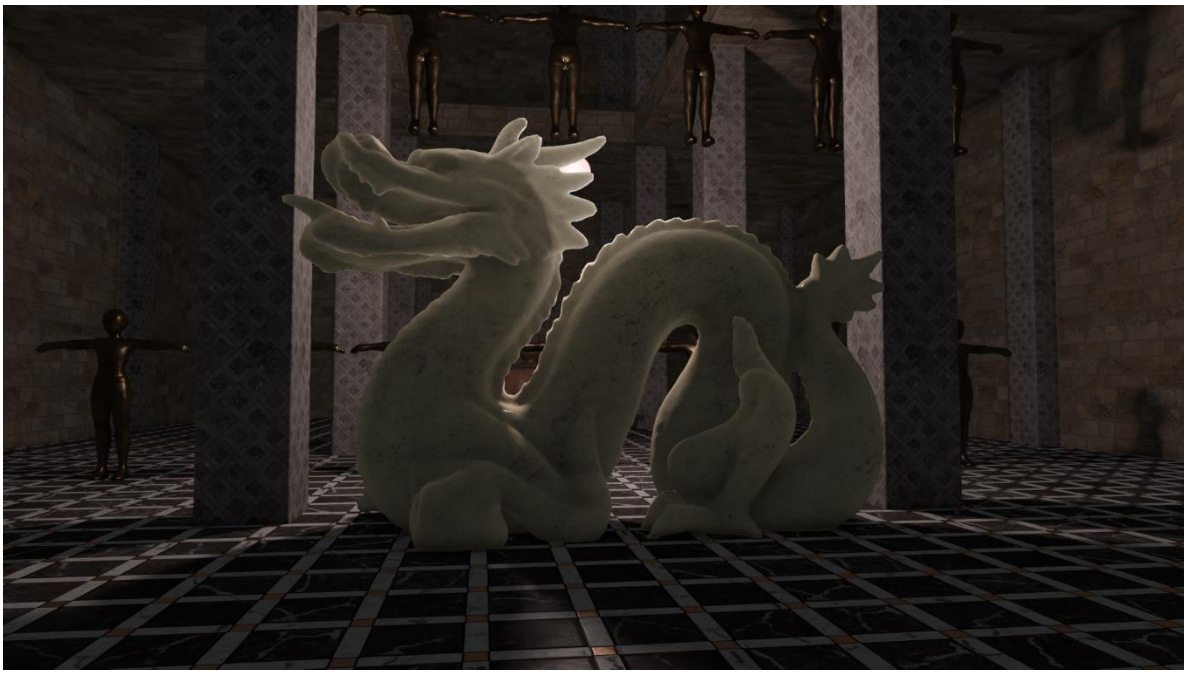


Figure 17: mTec default scene with Cook-Torrance BRDFs. The translucency of the dragon is clearly visible.

The previously described techniques were implemented as a C++/OpenGL renderer. This renderer is called mTec, is licensed under the MIT license, and its complete code can be downloaded from our depot[11]. The goal of development was to implement and test various optimization techniques in order to gauge the feasibility of distance field rendering in modern game development or real-time visualization. These techniques will be described in detail, with analysis of their effects on performance. The only SDF technique described in the previous chapters that was not implemented in the main mTec renderer is the reflection effect. Only some test renderings were performed with fully reflective sphere traced materials, but this was eventually abandoned for the more essential techniques. The following chapters will give an overview over mTec’s architecture and rendering pipeline.

[11] https://github.com/xx3000/mTec

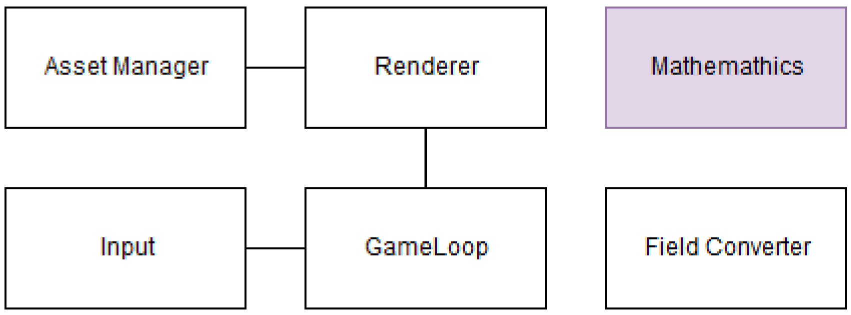


Figure 18: an overview over all mTec subsystems. The mathematics module is an external library.

8.1 Foundations

Modern engines consist of many subsystems and millions lines of code. The goal of mTec was to keep the renderer as focused as possible, therefore skipping all but the most essential systems. This chapter will give a short overview over these systems, as well as describe the external libraries that are in use. The most important part of mTec is arguably the renderer, therefore a far more in depth description of this component will be given in the later chapters. The mathematics module, GLM, will not be described in detail, as it is an external library[12].

8.1.1 Field Converter

The field converter is technically not connected to the main engine. It is a standalone executable that can be run in order to convert meshes to distance fields, which can then be used by mTec. Triangle meshes in the OBJ file format are read and then converted with the brute force method described in chapter 5.2.1 to our proprietary SDF format. Our distance field files are simple binary files with a header consisting of the dimensions (width, height and depth) of the field in object space, as well as a resolution factor. The resolution factor is used to convert the object space dimensions to actual world space distances. One could say that the resolution factor is the scale of the distance field. In order to preserve the linear interpolation properties of the distance field only uniform scaling is allowed. The file header is followed by the actual distance field grid values serialized into a one-dimensional array. The separation of this tool was chosen deliberately in order to stay flexible and be able to integrate it into a conventional asset pipeline easily. As soon as a new mesh would be generated by an artist, a new SDF could be calculated and provided to the engine.

[12] http://glm.g-truc.net/

8.1.2 Asset Manager

The asset manager is responsible for loading the distance fields, textures and metadata from the hard drive. mTec uses simple json files for specifying relevant information of each renderable object. The content of such a file can be seen below.

```
{
"Field" :"dragon_med",
"Texture" : "marble",
"Color" : [1.0,1.0,1.0,1.0],
"FillColor" : [2.0,2.0,2.0,1.0],
"Specular" :[8.0,164.0],
"Density": 1.9
}
```

The asset manager will try to load the specified distance field and texture, as well as create an appropriate structure holding the files and metadata in memory.
This module uses several external libraries for IO operations. The open-source library rapidJson is used for the parsing of the json object files[13]. To keep everything as simple as possible only png textures are supported, which are decoded by the lodepng library[14]. Finally, the loadOBJ class is used to load classic OBJ meshes[15]. It should be noted that only a single mesh is loaded during the entire run time of mTec: one big triangle to render some of the fullscreen effects to. The use of this triangle over direct compute shader based rendering will be further explained in the renderer chapter.
Once the objects are loaded, all relevant textures and SDF buffers are transferred into GPU memory. The only information that has to be kept in main memory are the object's transformation matrices, which are retransferred to the GPU on every frame. Therefore mTec has a low usage of main memory and the CPU as all relevant calculations are performed on buffers in GPU memory only.

8.1.3 Input and Gameloop

These two essential engine modules are kept as bare-bone as possible. The gameloop is a very simple module which is responsible for updating the game time, running an input module update and issuing a draw command to the renderer. The input module is responsible for checking for user mouse and keyboard inputs, which then get forwarded to

[13] http://rapidjson.org/index.html
[14] http://lodev.org/lodepng/
[15] http://www.opengl-tutorial.org/beginners-tutorials/tutorial-7-model-loading/

the appropriate places in the renderer. Keyboard events are propagated to the correct functions by events which are based on Reinalter's type safe delegates[16].

8.2 The Renderer Module

The renderer is the heart of the mTec engine. It dispatches all draw commands and compute shaders, is responsible for the setup and maintenance of all OpenGL states and buffers. This chapter will give an in-depth description of the renderer, its deferred rendering pipeline and all of its components.

For more convenient interfacing with the OpenGL drivers, two external libraries are used. These are Freeglut[17] and the GL extension wrangler library (Glew)[18]. Freeglut is used as a base framework for requesting an OpenGL context from the operating system, and as an interface between the OS input functions and our events. The Glew library is only used in mTec to provide vertical sync functionality.

Many programming languages support file includes through preprocessor directives, but unfortunately this is not the case for GLSL. The lack of this essential feature makes it very hard to have proper separation of concerns and reuse common functions between different shaders. To solve this problem a custom include solution was developed for mTec, utilizing a general preprocessor[19]. The visual studio build pipeline of mTec is set up to run the GPP executable over the shader files before copying them to the output directory. This preprocessor resolves all custom include commands by directly pasting the code from the referenced files into the current shader which allows for compact and easy to read shader files during development.

[16] https://blog.molecular-matters.com/2011/09/19/generic-type-safe-delegates-and-events-in-c/

[17] http://freeglut.sourceforge.net/

[18] http://glew.sourceforge.net

[19] https://github.com/logological/gpp

8.2.1 The Deferred Rendering Pipeline

mTec is using a deferred architecture for rendering. All relevant information and effects are written into a buffer (the so called gBuffer), before being properly composited in a final shading step. Figure 19 visualizes the pipeline and all discrete steps that are executed for each frame. Each step consists of at least one shader dispatch or draw call. For easier visualization the diagram is divided into several distinct groups. Many steps in the pipeline are either preprocessing (ex: culling) or postprocessing (ex: upsampling) of the actual calculations or rendering effects. These distinct groups or modules are also present in the actual code. The reason for this is twofold: First, it enables easier optimization and work on a specific part of the pipeline while not having to touch the other elements. Secondly, each of these modules could be switched out easily for a conventional rendering technique, if so desired. The modules operate independently of each other, on buffers provided by the preceding steps. The sphere tracing step could easily be replaced by conventional rasterization of geometry, providing just depth and normal information to the following steps. Distance field shadows and ambient occlusion could be replaced by shadowmapping and SSAO, in order to improve runtime performance or compare these effects. Finally, the lighting step supports simple Blinn-Phong as well as Cook-Torrance lighting modes. This could be upgraded to PBR with ray marched or voxel cone tracing based global illumination.

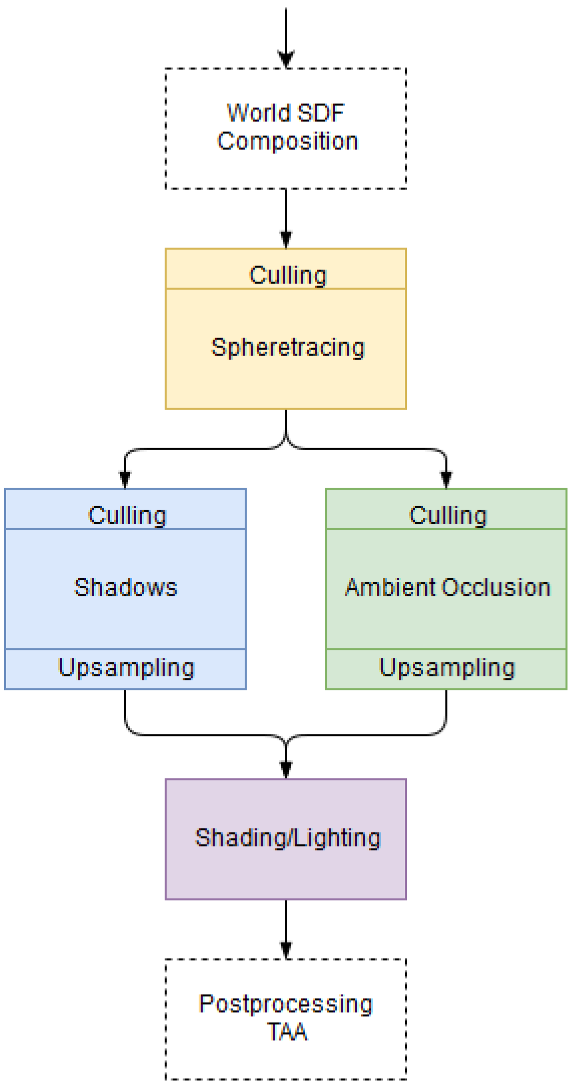
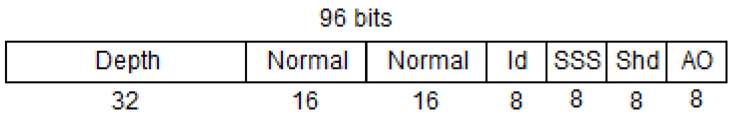


Figure 19: The mTec deferred distance field rendering pipeline. All colored blocks correspond to seperate rendering modules.

All calculations are performed by compute shaders, except for the final lighting and postprocessing steps, which are fullscreen draw calls using fragment shaders. Compute shaders have the advantage of bypassing the regular OpenGL rendering pipeline and just performing calculations on buffers and data directly, which is faster than the whole trip through the rasterizer. A local group size of 64 threads was chosen for the compute shader



96 bits

Depth	Normal	Normal	Id	SSS	Shd	AO
32	16	16	8	8	8	8

Figure 20: mTec gBuffer composition.

calculations. This is two times the size of a GPU warp on our test system[20]. Empirically this resulted in the fastest execution times. Depending on the kind of calculation that is performed in each step, the compute shader ether writes its outputs directly into a texture through OpenGL's ImageReadWrite system, or into a simple linear buffer. Writes into a buffer are used for all culling calculations, as their outputs are just arrays of indices.

Each frame starts with the recalculation of model matrices and transfer of uniform buffers to the GPU. The first step, the world SDF composition, is only run on the first frame of the engine's execution. During run-time the world field is only updated when necessary. Before sphere tracing the distance fields, the camera frustum is subdivided into many sub-frustums and all objects are culled to these tiles. This reduces the amount of fields that have to be fetched in the next step. The depth of the scene as well as object normals and material ids are determined through sphere tracing and written into the gBuffer. Furthermore the thickness of SSS enabled objects is calculated in this step and added to the buffer. These buffers are then passed to the shadow and AO modules which are dispatched simultaneously, as their calculations are independent of each other. There are still discrete steps which have to be executed sequentially within the modules itself: both culling steps are run in parallel, followed by the effect calculations and finally the upsampling shaders. mTec's AO module supports all three ambient occlusion calculation techniques described in the previous chapters but the default technique used is our hybrid approach and this is reflected in the culling calculations, which create object lists according to each pixel's maximal AO influence range. The culling as well as the actual ambient occlusion calculations are performed in quarter the actual screen resolution (1/4 of width and height, 1/16 of the actual screen pixels). In order to increase the final visual quality, a bilateral filtering shader is run in the end, which upsamples the reduced buffer to full resolution. The same process is performed in the shadows module, which is rendered in half screen resolution (1/4 of the actual screen pixels) and then upsampled. Further undersampling results in very visible and hard to deal shadow seams in the final image which is why these specific dimensions where chosen.

[20] http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capability-5-x

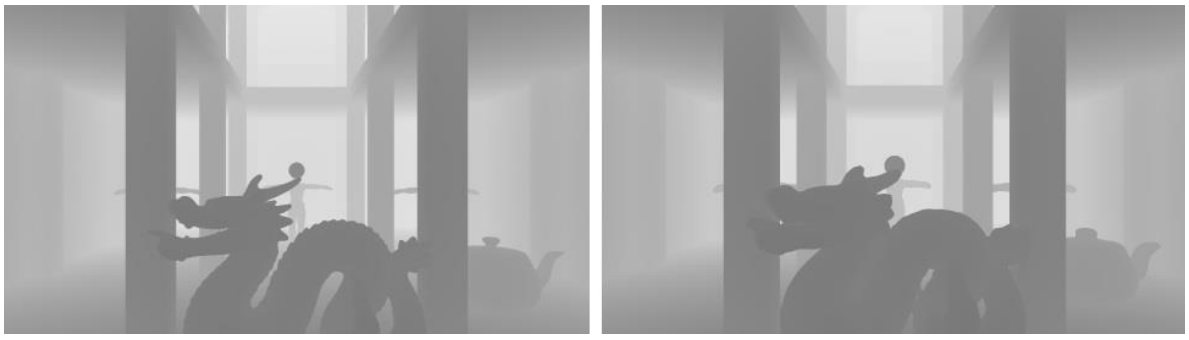


Figure 21 : The depth buffer after sphere tracing. On the left, the full algorithm with all optimizations is run. On the right only the world distance field is polled. The conservative distance estimate is clearly visible by the bulkiness of the dragon in the front. Due to the low resolution of the field, details like the dragon's scales disappear.

After all calculations are finished, the gBuffer is passed to the shading/lighting step. This module performs texturing and normal mapping for each pixel and determines final pixel color by running the collected information through a Blinn-Phong lighting model. The more advanced Cook-Torrance model is supported by the renderer, but disabled by default. The lighting models are completely separated from the distance field techniques, therefore all performance evaluations were performed with Blinn-Phong lighting to reduce the complexity of the non-distance field calculations to a minimum. The final step in the mTec rendering pipeline is a post-processing step, in which temporal anti-aliasing as well as camera motion blur are applied to the composited image. It is a part of the renderer but completely decoupled from and oblivious to the presence of distance fields. A more detailed description of each rendering step will be given in the following chapters.
The composition of the gBuffer can be seen in Figure 20. It has a total size of 96 bits per pixel. 32 bits are used for high precision depth storing, 2x16 bits for compressed normals, 8 bits for object thickness and another 8 for the material id. The last 16 bits are evenly distributed amongst shadow and ambient occlusion factors. Other buffers are used, but not accounted for in this gBuffer calculation. These are, amongst others, the TAA history buffer, various culling lists, the global distance field and low-res shadow and AO buffers, which get upscaled and transferred to the gBuffer. It is of uttermost importance to keep the gBuffer as small as possible, and each bit that can be shaved off results in significant performance gains. As an example, during development the id and thickness fields in the buffer were reduced from 16 to 8 bits per entry. This reduction of the overall buffer size by about 4 Mb (in 1920x1080 resolution), has caused an increase in overall performance of about 1.2 ms for 20 objects.

8.2.2 World Distance and Id Field Composition

When sphere tracing through a scene consisting of multiple distance fields, each field may be polled hundreds of times, depending on the maximum number of steps and the location of the objects. Arbitrary distance fields that are generated by our converter tool from meshes are kept in GPU memory as 3D textures. Our test system has a texture cache of 24 kb[21], which is too small to hold all of our fields during execution. Therefore the SDF textures have to be re-fetched constantly from main GPU memory, which is very slow. Reducing the amount of texture reads that need to be performed is essential in increasing sphere tracing performance. The first reduction technique that is performed in mTec is based on Wrights' global SDF optimization [14]. In the first frame of execution, a shader is dispatched that creates a low-resolution distance which spans the entire scene, by polling all object fields at each grid position. A fixed value is added to the result in order to create a more conservative distance estimate (Figure 21).

As an additional, novel optimization a second buffer with the same resolution is created, but is filled with the object ids at each grid cell instead of distances. When sphere tracing, instead of going over all fields, the algorithm polls the world distance field to get an estimate of the correct distance and this distance is used for further iterations, as long as it is above a certain threshold value. The threshold value is carefully chosen to be the size of a single grid cell, in order to prevent loss of detail in the final image, which would result from only using the low-resolution world field. If the distance is smaller than the threshold value, the algorithm additionally polls the world object id field in order to determine which object it is closest to. The resulting object is the only distance field that has to be polled during this iteration. Without any further modifications the resulting sphere tracing calculation would be much faster, but return incorrect results in cases where two objects are close to each other. This is caused by the fact that the id field basically contains indices, which cannot be interpolated linearly over the grid cells like a normal distance field. The ids saved in the field are only correct for the center point of the cell, which can lead to errors when a cell is equally close to two different objects. This problem is addressed in mTec by a post-processing step, which goes over the id field and compares the value of each cell with its direct neighbors. If the neighbors of the cell have different values than the cell itself, the region is considered unstable and an invalid id is inserted into the field (Figure 22). This creates a border region of invalid values around the intersection point of object field influences. When the sphere tracing algorithm polls the id field and gets an invalid result, it is forced to poll all individual object fields. This is much slower but results in the correct distance. Further optimizations for this edge case are described in the culling section of the sphere tracing module chapter. The world distance

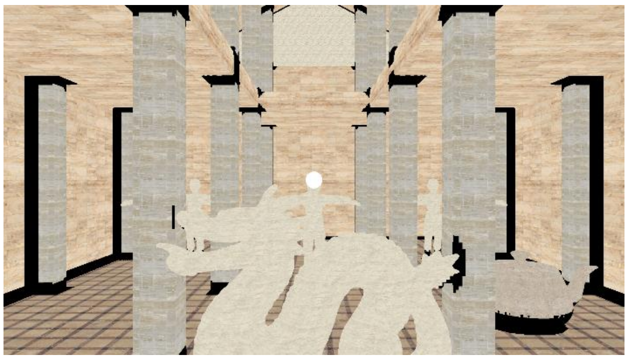


Figure 22: Visualization of the world id field. All black regions are „uncertain“, i.e. intersections between different field influences.

[21] http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capability-5-x

field is “coated” by a thin layer of negative distances along its edges in order to force a switch to regular object field polling should a ray leave the field. The texture sampler options for the field are set to clamp at the borders to enforce this behavior. In theory, both the distance and the object ids could be merged into a single texture. All algorithms that use the world distance field switch to the high-res object fields if close enough to an object. This means that the distance information from the world field is not used around or inside of objects. It could be possible to write the object id information directly into these areas instead of the distances. Unfortunately this approach poses a few problems. First and foremost, the distance field needs to be linearly interpolated in order to get accurate values for each polled point. If object ids would be stored in the same texture, bilinear filtering would be applied to them as well, which would nullify the meaning of these integer indices. Even if the same texture would be provided to the appropriate algorithms with two different filtering schemes the main issue would still be the border region, where distances switch to ids. In this border region inside the distance field, distances would be interpolated with neighboring object ids which would result in catastrophic distance overestimates. A final problem is the fact that algorithms poll the object id as soon as the distance field reports a value lower than the precision threshold. This happens in the same calculation step, with the same texture coordinates. Therefore we would need both distance and id information in the border regions, making two textures necessary.
The field size in our test scene is fixed to a resolution of 512x256x512. This is sufficient and fairly high-res for a small scene like ours. The higher the resolution of the field, the lower can the switch-to-object threshold be without visible artefacts. Lower switching thresholds result in higher overall performance, as less textures need to be polled during sphere tracing. For big, open-world scenes a single world distance field would be too small to provide any significant benefits and a dynamic update approach has to be chosen. The field could be divided into regular-sized chunks and new chunks could be recalculated once the player leaves the current chunk. This would effectively generate 9 small world fields surrounding the player. Dynamic world distance fields for large scenes are currently not supported by mTec, and are a possible direction for future research.
The world distance field would only allow for static scene geometry if it would be generated only a single time, at startup. To allow for object animations and morphing, the world field needs to be updated at run-time. Due to the high resolution of our world field, a recalculation of the whole field takes about 110 ms. This is passable as a start-up cost, but absolutely unacceptable for run-time updates, which is why mTec only recalculates relevant portions of the buffer during run-time. Whenever a change in an object’s transformation is detected, the asset manager calculates the current bounds of the object and generates a bounding box that is axis aligned to the world distance field. This AABB is then used to calculate which buffer cells have to be updated. The resulting grid cells are added to a list and a compute shader is dispatched for each element of this update list during run-time. Each update needs to add two separate entries into the update list: An entry for the current bounds of the object, and an entry for the bounds before the

transformation. This has to be done in order to avoid a “smearing” of moving objects across a scene. As mentioned earlier, the world field is a conservative representation of the actual scene: even though the actual object may have moved, it will still be rendered correctly as long as the distance that it has moved by is smaller than the world-to-object threshold value. Therefore the field does not need to be updated every frame to result in correct images. Different metrics could be used in order to determine when to dispatch updates and this is highly dependent on the scene and nature of transformations. The simplest approach would be to wait a fixed amount of frames before scheduling an update, which works surprisingly well for simple transformations in mTec. The maximum amount of frame that can be skipped before updating the sample scene is about 7 and any more result in visible object flickering. This is the case because of the high frame rate on our test system. Of course this naïve method only works correctly in the narrowest of cases and no reliable estimates can be made about its reliability for different systems, transformations and scenes. A better heuristic would be to calculate the difference between the original AABB and the AABB of the current frame. Should the difference in any frame be bigger than the world-to-object threshold, an update should be scheduled. This heuristic is independent of the renderer’s FPS and transformations performed. Overall, the cost of dynamically updating the world SDF is highly dependent on the amount of objects transformed and the update heuristic used. For the mTec sample scene, the dragon model is rotated at a fixed speed around its y-axis. The object distance field has a resolution of 128x91x58 cells, which translates into 100x72x48 world grid cells that need to be updated. This animation takes about 0.8 ms per frame, which is just the world field recalculation costs averaged over all frames. This is a significant cost for such a simple animation, but the overall update process is fairly un-optimized in mTec, and there is still a lot of potential for speed ups in the future. The update cost is as high as it is because many cells have to be updated in the high-res world field in addition to the world id field. The performance could be increased drastically by reducing the world field size or the amount of cells that need to be recalculated.

8.2.3 The Sphere Tracing Module

Sphere tracing is used in mTec for depth finding in a scene and is performed on a full-screen buffer in a compute shader. The shader receives the world distance field and a culled list of object fields and proceeds to march along a ray while checking the distance fields at each step until an intersection is found or a maximum amount of steps is reached. As the object distance fields are represented as 3D textures, multiple transformations have to be performed before being able to sample them in a shader. The current marching position in world space has to be multiplied with the objects inverse model matrix. Then this object-space point is scaled and divided by the SDF resolution which results in a coordinate that can be plugged into the shader sampler functions directly. When converting meshes to 3D distance textures, it is desirable to have the texture surround the mesh as tightly as possible. This reduces the amount of empty, unnecessary grid cells and increases the effective resolution of the field. During sphere tracing however, tight fields lead to issues when the polling point lies well outside the texture bounds. Texture samplers are set to clamp texture coordinates to the [0,1] range, which would lead to a huge underestimation of step sizes in scenes with tight distance fields, as no matter how far our point would lie outside the 3D texture, it would always be clamped to texture borders. This problem is solved by calculating the distance between the clamped and the original, un-clamped point, and adding it to the polled result. In the case that our point lies within the texture, the clamped and un-clamped points will be equal, resulting in a distance of zero being added to the polled value. In our implementation the algorithm steps no more than 250 times before being terminated.

In addition to determining a pixel's depth, the shader proceeds to calculate the normal, the material id as well as the thickness of SSS-enabled materials. Normals are calculated through the gradient method and packed into two 16 bit fields using octahedron vector encoding[22]. Each distance field is assigned a unique material id during mTec initialization, which is used during the shading step of the pipeline to determine which texture and lighting parameters to use. This id is determined by checking which field is closest to the calculated pixel depth and then stored in the gBuffer. A multi-layered optimization system is used by the sphere tracing algorithm in order to keep texture fetches to a minimum. Every distance field check starts by polling the world distance field first. Only if the resulting value is smaller than a certain threshold a second check will be performed by polling the world id field. This fetched id can lead to two possible outcomes: if it is a valid object id, then this specific field texture will be checked and its value returned. If the id is invalid (because of a border region of multiple objects in the scene) the algorithm will resort to polling multiple field textures and checking for the smallest distance. Even in this worst case not all scene fields have to be checked, as a list of culled objects is used. This culling process is described in the following chapter. It should be noted that the world id field is not only used for sphere tracing but also for the determination of the final material id.

In order to simulate sub-surface scattering, the thickness of an object has to be estimated. This is calculated in mTec by continuing to sphere trace the ray into the object until a surface or a maximum thickness is reached. The maximum thickness is a per-object constant which determines at what depth an object becomes fully opaque to light.

The thickness trace could be optimized by performing it in much lower resolution, but due to the small maximal step count and the fact that only one specific distance field has to be polled at every step, it is a quite fast algorithm compared to the other techniques.

[22] https://knarkowicz.wordpress.com/2014/04/16/octahedron-normal-vector-encoding/

8.2.4 Sphere Tracing Culling

Even though the world distance and id fields results in significant speedups of sphere tracing, the worst case scenario of multiple field influences on a single grid cell would require to go over a list of all distance fields in the scene. Thankfully this list can be efficiently culled to include only fields that lie in the vicinity of the sphere tracing ray. The generation of these lists is performed in mTec before the sphere tracing step during each frame. The camera frustum is subdivided into 34560 sub-frustums, which corresponds to 8x8 pixels per tile. During the uniform update step a bounding sphere and oriented bounding box (OBB) is calculated for each object. This bounding object list is used by the culling shader to perform basic intersection tests between the frustum and the objects. Initially a sphere-ray intersection is performed, which is then refined by an OBB intersection. The id of every object that passes both tests is added to the culled list. The culling list has 128 8-bit slots per tile. 127 of these slots can be used for the object ids, as the first slot contains the length of the array contained in the tile. This results in the maximum number of objects per scene being 127. Of course this could be increased very easily to any arbitrary number. It should be noted that if the array size would be increased to accommodate more than 256 elements, the slot size itself would have to be increased from 8 bit as well in order to hold more than 256 unique object ids. Changes of this kind should be profiled extensively to determine how well the new buffer sizes harmonize with the GPU caches and how it affects performance.

Objects in scene	*No Optimizations*	*Only Culling*	*Only World Ids*	*Only World Distance*	*All Optimizations*
20	44.54	5.92	23.03	19.68	3.93
40	449.01	7.46	248.97	198.96	4.52
60	1477.26	8.81	761.68	606.7	4.96
80	2513.91	13.73	1303.86	1009.9	6.35

Table 2: Total sphere tracing time of the scene in ms, with various amounts of objects and optimization features. The algorithm results in the calculation of pixel normal, depth and id. Please refer to Appendix I for details on our testing setup.

The relevant question that should be asked of all optimization schemes is how they impact the performance. Table 2 illustrates the sphere tracing times of a scene with various numbers of objects. It can be seen that by employing all three techniques in combination results in tremendous speedups, especially in scenes with higher object counts. Objectively culling has the highest impact on the overall performance, as it reduces the rendering time by a factor of 100 in comparison to the other techniques. However, both the world distance and world id field are also being used in other modules as speedup structures, which is why their performance impact should be evaluated over all rendering modules combined. These measurements were taken in the mTec sample scene and

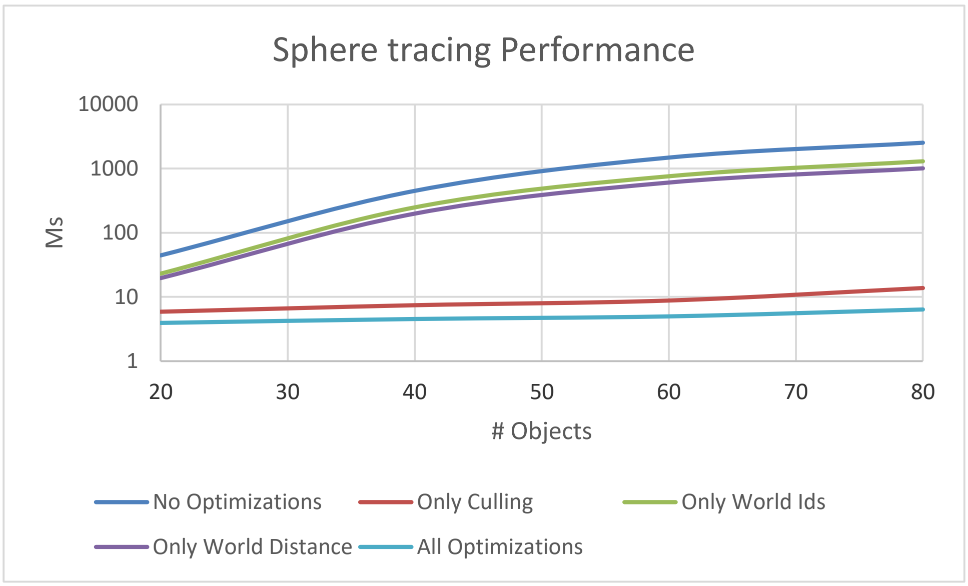


Figure 23: The results of Table 2 plotted on a graph. The y-axis uses a log scale for better visualization.

different geometries, field sizes, and object positions might affect the results. Therefore a scene should always be profiled to determine which of the optimization techniques should be used in order to maximize rendering performance.

8.2.5 The Shadows Module

Soft shadows are calculated by cone tracing a ray from each pixel towards the light. The pixel position in world space is reconstructed from the linear depth saved in the depth part of the gBuffer. In order to speed up the algorithm, the calculations are performed in half resolution. A post-processing step is dispatched after the main calculation is finished and this half-resolution buffer is upscaled using bilateral upsampling. The undersampled shadows calculation results in visible artefacts on edges, but these artefacts can be visibly reduced through TAA post-processing. The cone tracing itself employs the world id field as well as culled object lists as means of speeding up the algorithm. The world distance field is not used as our tests have shown no relevant speedups with this technique.

8.2.6 Shadow Culling

The calculation of the culled shadow object lists works similarly to the culling for sphere tracing. The shadow calculation dimensions are halved again, which results in the culling shader being run in ¼ resolution in each dimension, each culling tile covering 4 (2x2) shadow pixels. The shader loops over all objects in the scene and performs intersection tests of the object bounding spheres against a ray from the interpolated pixel depth towards the light. As the shadow are calculated through cones and not rays, a constant factor is added to all spheres in order to achieve a more conservative culling. This

Objects in Scene	*No Optimizations*	*Downscaling*	*Culling*	*World Ids*	*All Optimizations*
20	38.04	11.65	6.69	18.83	2.49
40	455.03	131.66	10.75	297.45	3.77
60	1465.31	420.89	13.55	921.06	6.02
80	2502.38	690.23	17.2	1596.69	10.4

Table 3: Shadow render times in a scene with various amounts of objects. Please note that per pixel depth and normals are required for shadowing. Therefore these tests are run with sphere tracing enabled, and the sphere tracing times (Table 2, last column) are subtracted from the results.

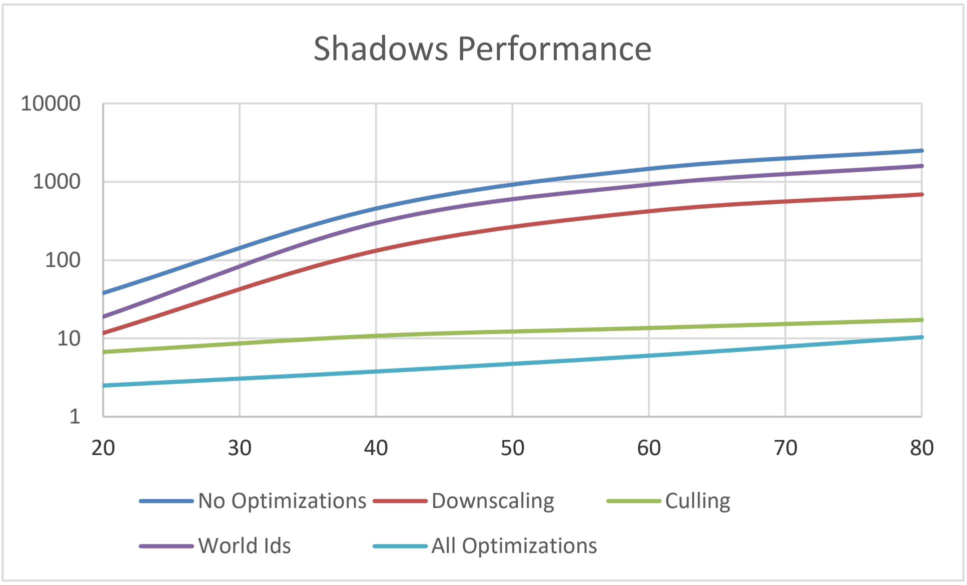


Figure 24: The results of Table 3 plotted on a graph. The y-axis uses a log scale for better visualization.

is a cheap way to ensure that even at the maximum cone distance no objects are culled, which might be relevant for soft shadowing. Because each tile averages the depth of 4 pixels, artefacts arise on border regions with high depth disparities. The frequency of the appearance of these artefacts is highly dependent on the scene and objects, but in the mTec default scene they appear very rarely, and are alleviated through TAA postprocessing. This edge case could be handled by calculating the difference between the depths of the four tile pixels, and invalidating the culled list in cases where the maximum depth difference passes a certain threshold. If an invalid list is encountered by the cone tracing algorithm, it would fall back to checking all scene objects.

The performance impacts of the various acceleration techniques are quite similar to the sphere tracing optimizations (Table 3, Figure 24). Culling has the highest impact on the overall performance by far. Downscaling the calculations to a quarter of the original pixels leads to a speedup factor of about 3.5. This is to be expected from reducing the dispatch size, as the optimal factor of 4 is impossible to reach due to overhead in the non-parallel portion of the program.

Objects in Scene	*No Optimizations*	*Downscaling*	*Culling*	*World Ids*	*All Optimizations*
20	33.28	2.78	4.14	10.91	0.93
40	384.65	24.67	4.69	112.7	1
60	1166.39	69.4	5.32	354.39	1.09
80	1959.31	125.51	5.26	598.38	1.76

Table 4: Ambient occlusion calculation times in a scene with various amounts of objects. Please note that per pixel depth and normals are required for ambient occlusion. Therefore these tests are run with sphere tracing enabled, and the sphere tracing times (Table 2, last column) are subtracted from the results.

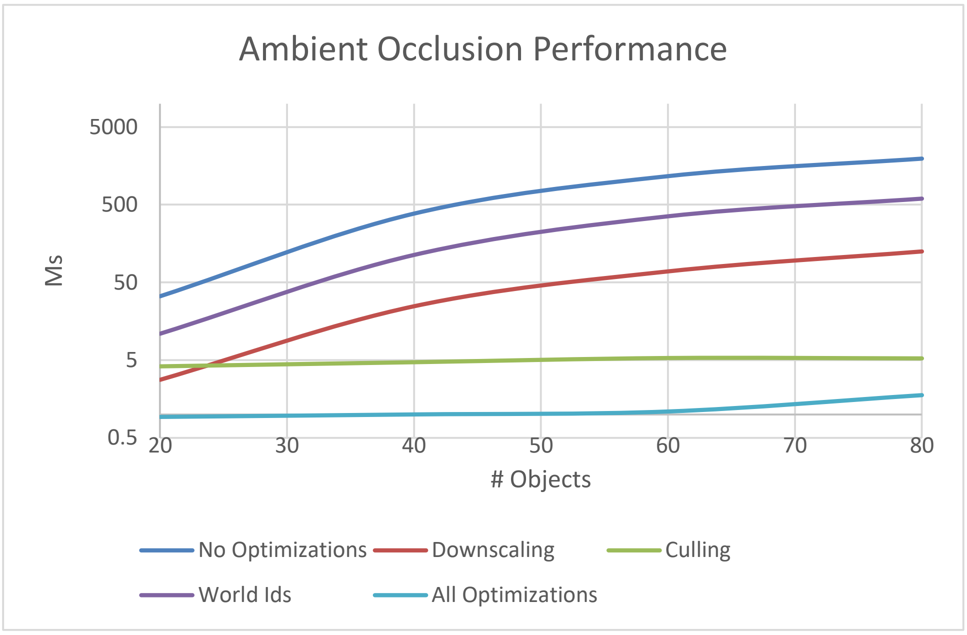


Figure 25: The results of Table 4 plotted on a graph. The y-axis uses a log scale for better visualization.

8.2.7 The Ambient Occlusion Module

Ambient occlusion, which is dispatched simultaneously with the shadow module, is calculated in ¼ of the rendering resolution. With such a low resolution, regular bilinear upscaling would result in very pixelated AO in the final image. Therefore, similarly to shadows, upsampling is performed though bilateral filtering. Even though mTec supports AO calculation with all three AO algorithms described in the earlier chapters, the default algorithm used is our hybrid approach as it has a much better performance than cone tracing while still maintaining good visual quality and eliminating artefacts from the naïve 5-tap approach. The algorithm is accelerated through the world distance field, the world id field as well as a culled object list. In order to guarantee correct occlusion of small features close to the pixel itself, the world distance field is only polled if the ray has reached a certain distance from the original pixel position.

8.2.8 Ambient Occlusion Culling

The AO object lists are calculated using a compute shader that is dispatched in the same resolution as the AO shader itself. The culling algorithm is configured for the hybrid ambient algorithm, but could be adjusted to accommodate the other two quite easily. As the hybrid algorithm performs 5 taps along 5 rays in a normal-oriented hemisphere, we can pre-calculate the maximum influence range of every pixel. Only objects that lie within this range can contribute to the pixel's ambient occlusion. A culled object list can be created easily by intersecting the bounding sphere of each scene object with a sphere at the pixel's position with the maximum influence as its radius.

The performance measurements reveal a similar picture to the shadows calculations, except that the AO calculation is much faster in general (Table 4, Figure 25). A big contributing factor is the much smaller dispatch size of only 1/16 of the full screen size. This results in speedups by a factor of over 15 for higher object counts. The culling of distance fields yields the highest speedups of any single optimization technique.

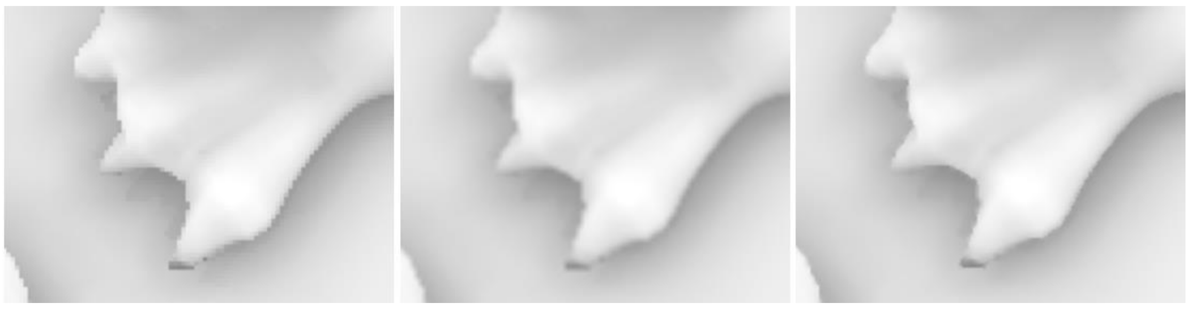


Figure 26: Comparison of different upsampling techniques. The left image is bilinear upsampling, the pixelization issues are clearly visible at object edges. The center image was upscaled with a Gaussian blur using a 3x3 kernel. The pixelization disappears, but the AO factor of different objects gets smeared across object boundaries. The right image is upscaled using bilateral filtering. The pixelization issues disappear, but the object boundaries stay sharp.

8.2.9 Bilateral Upsampling

Lighting effects such as ambient occlusion can be calculated in lower resolutions and then upsampled without relevant degradation in visual quality of the final image. These techniques mostly produce low-frequency information, which is preserved in lower resolution images. Such undersampling can result in significant increases in rendering performance, but the question becomes how to properly upscale the low resolution image to match the rendering size before the composition step. A comparison of three AO images can be seen in Figure 26. The left image is generated through simple bilinear upsampling. This method is very fast, as it can be performed by the GPU automatically, but results in visible pixelization of the image. The visible pixels can be removed by running a Gaussian blur over the image after bilinear upsampling. Even small kernel sizes remove most of the pixelization artefacts. Unfortunately the blurring of the image leads to smearing of occlusion information across object boundaries. Optimally we would want an algorithm which blurs away the pixelization but preserves sharp edges. That is exactly what can be achieved through bilateral filtering [44]. This algorithm works similarly to a classical Gaussian blur, but every neighboring pixel in the kernel is not only weighted by its distance but by other geometry-aware factors. In our case the additional weighting is provided by the differences in depth and normal of the center to its neighboring pixels. Furthermore a simple tri-weight kernel as proposed by Herzog et al is used [44]. The result of this upsampling can be seen in the rightmost image of Figure 26. Pixel artefacts are blurred away while still maintaining sharp edges.

8.2.10 The Lighting Module

In the last step of the rendering pipeline the actual color of all pixels for the final image has to be determined. This step is done in mTec by a full-screen draw call, with all relevant work performed in a fragment shader. Conventionally two triangles aligned to create a full screen quad would be used to guarantee a fragment shader execution for each on-screen

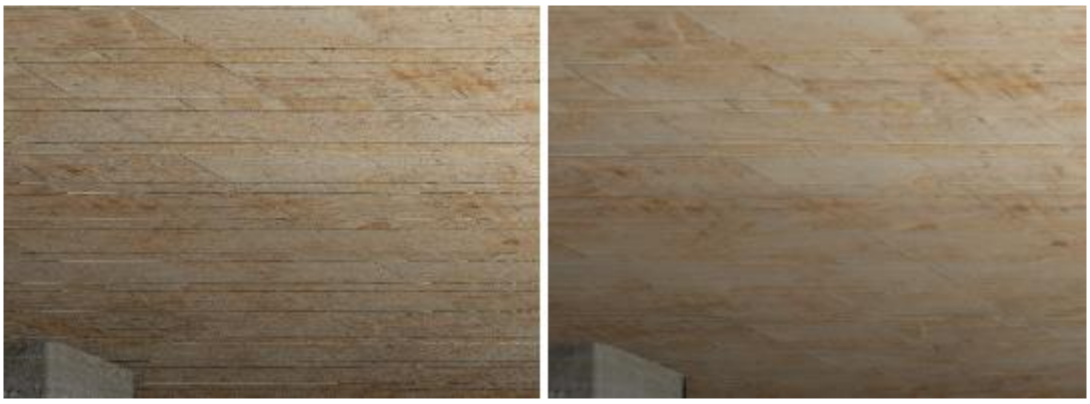


Figure 27: Comparison of compute vs fragment shader rendering. Left image: compute shader. Right image: fragment shader. The missing anisotropic filtering is clearly visible.

pixel. In mTec a single huge triangle is used instead. This is an optimization originally proposed by Bilodeau[23], which aims to eliminate pixel overdraw at the diagonal when using a triangle-quad. The triangle is big enough to cover the entire screen and is clipped by the GPU in the rendering pipeline to match the screen boundaries.

The fragment shader performs texturing and normal mapping before passing the color and gBuffer information to our Blinn-Phong lighting model where each pixel is correctly lighted according to the accumulated information. This requires the reconstructed world space position and the normal of the pixel. If we would just calculate the texture color based on the world space position, textures would start to “flow” on moving objects. This can be avoided by performing the tri-planar mapping in object space, and multiplying all pixel positions with the inverse model matrix before plugging them into the texturing calculations. The same adjustment has to be performed on the world space normal by multiplying it with the inverse transpose of the model matrix, ensuring that our normals are only rotated and not translated by the operation.

Tri-planar normal mapping is performed in order to make the terrain look “rougher” and simulate a higher distance field resolution. Similarly to texturing, the calculations are performed in object space. The material properties such as texture, normal map and lighting parameters are retrieved by polling a buffer according to the object’s id.

As previously noted, the lighting is calculated using the regular rendering pipeline. Performing this final step with a compute shader and blitting the resulting buffer into the default framebuffer lies well within the realm of possibility and tests with such a setup were performed during development. In the end the reasons for abandoning such approach were twofold: First, the OpenGL windowing framework used by mTec, freeglut does not support a default SRGB framebuffer. This leads to gamma issues when trying to blit directly from an SRGB texture into a linear buffer, as the blit operation does not perform gamma correction. The second reason for abandoning compute shader rendering lies in texture level-of-detail and anisotropic filtering calculations. The ability of a fragment shader

[23] https://www.slideshare.net/DevCentralAMD/vertex-shader-tricks-bill-bilodeau

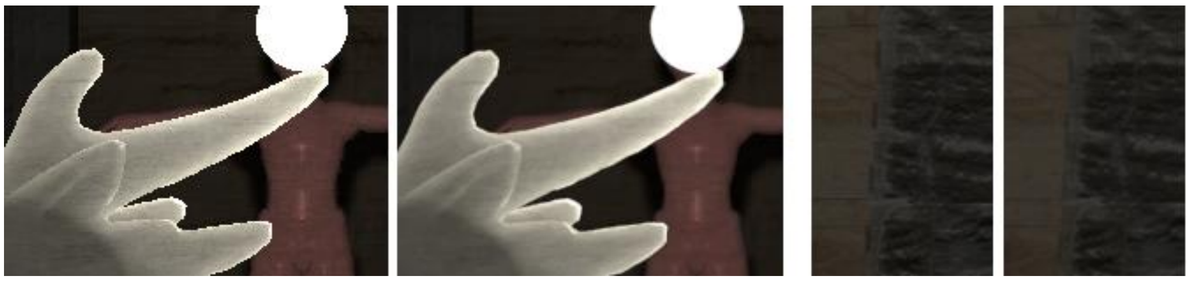


Figure 28: Comparison of a rendered image with and without TAA. The left image pair demonstrates TAA's blurring and anti-aliasing effect. The right image pair shows shadow bleeding artefacts from undersampling. These artefacts are visibly reduced through temporal anti-aliasing.

to choose the correct mipmap level of a texture based on pixel derivatives leads to big increases in visual fidelity. Anisotropic filtering in particular leads to smooth textures perpendicular to the viewer, and eliminates texture flickering during movement. Unfortunately these features cannot be used in a compute shader. The dFdx and dFdy functions for calculating screen space derivatives are only accessible from a fragment shader, and sampling a texture in a compute shader will always return a LOD-level of zero, ignoring mipmaps and anisotropic filtering. A comparison between both rendering techniques can be seen in Figure 27.

The calculated color, normal, thickness, shadow and AO factors are then passed to the Blinn-Phong lighting equation. The SSS approximation is calculated in this step, by blending between two light colors according to the object's thickness. The thickness information from the texture calculated in the sphere tracing step is plugged into the translucency formula presented by Barré-Brisebois et al [39]. The output of this step is a simple RGB buffer. At no point in the rendering pipeline is blending or a hardware depth buffer used, as the sphere tracing of the scene results in explicit depth finding without the need for multiple draw calls.

8.2.11 Postprocessing

The finalized color buffer from the lighting step could be directly transferred to the backbuffer if one was willing to accept ugly aliasing artefacts along object edges. In order to improve overall image quality and reduce the visibility of undersampling artefacts, mTec employs a combination of temporal anti-aliasing and static motion blur. The TAA implementation itself is a port of PlayDead's TAA implementation for Unity, which was presented at GDC 2016 [40]. Except for a few modifications it follows the same frustum jitter, calculate pixel velocity, unjitter and reproject scheme. The main modification in mTec is that the velocity buffer is not calculated in a separate shader step, but instead the velocity is reprojected on the fly in the TAA shader. This increases the overall performance and removes the need for a dedicated velocity buffer. The same optimization could be performed in a regular renderer, but it would make per-object motion blur impossible. Conventionally object motion blur is calculated by drawing object velocities into the buffer

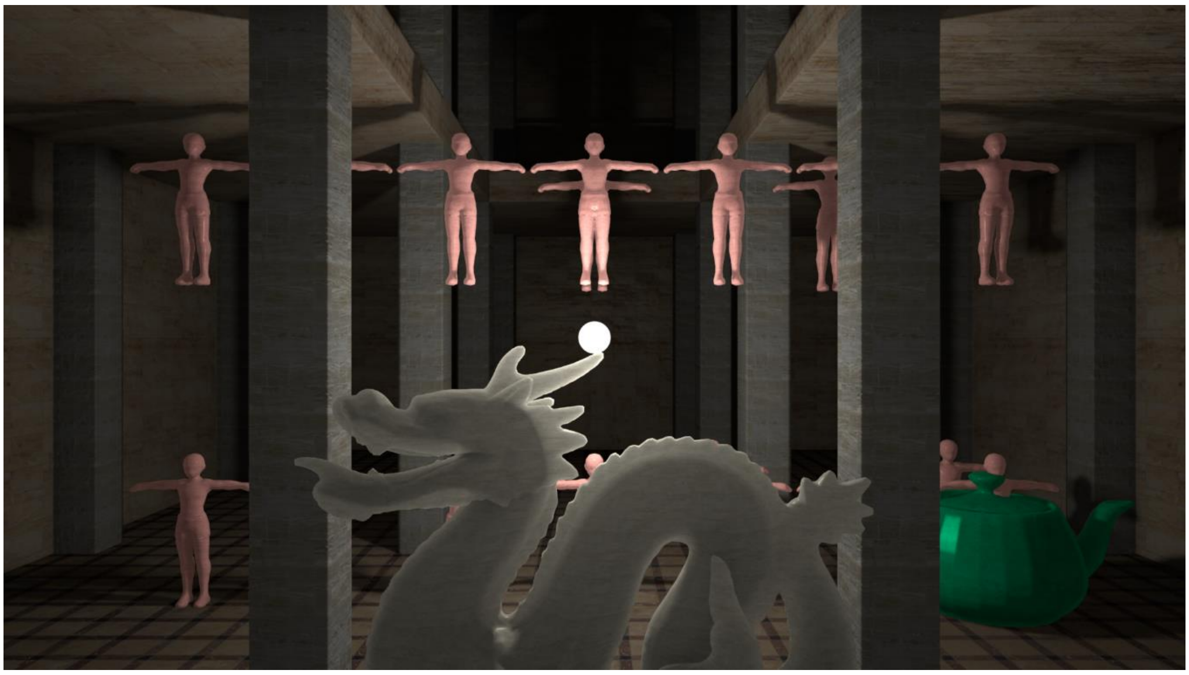


Figure 29: The default scene setup that is used for all performance measurements with Blinn-Phong lighting, as seen here with 20 objects.

after the camera velocity has been calculated. In mTec, object velocities could be calculated in a single step, as the id of each on-screen object could be determined from an id buffer lookup. By caching the object transform matrices from the previous frame, the current velocity could be calculated in a single step without the need for multiple expensive draw calls. This technique for dynamic motion blur has not been implemented in mTec but could prove an interesting direction for future research.

All together the whole post-processing step takes about 0.18 ms of rendering time. These are constant costs, independent of the number of objects or the scene setup. In addition to alleviating the aliasing of edges, TAA also helps in reducing the visibility of artefacts from undersampling of shadows and AO. Examples of this can be seen in Figure 28. These artefacts most commonly appear on straight edges such as the pillars in the mTec test scene. Although TAA reduces the apparent size of these artefacts, it also introduces a visible flickering on such border regions. A possible solution to the flickering of unstable regions is proposed by Wright [14]. In their approach the current pixel is filled with its blurred surroundings if it is deemed “unstable” by TAA. This solution has not been tested in mTec, and represents a future research direction.

After post-processing the resulting color buffer is blitted into the backbuffer, and a buffer swap is performed, thus ending rendering for the current frame.

8.3 Performance Analysis

The main focus of this thesis was the development of efficient ways to render distance fields in real time. This was a long process, in which all the different components of the renderer were rewritten, extended and re-tested. This chapter will give a more thorough look on the overall performance instead of focusing on isolated features. When rendering distance fields it is rather hard to get an objective measurement of the costs of a single object rendered in the scene. The rendering time is highly dependent on the placing of the objects themselves: the closer they are to each other, the more likely it is that they will generate invalid borders regions in the world id field, making a fallback to culled lists necessary. Close objects are more likely to be added to the same culling list, making AO and shadows calculation more expensive. After a certain object count is reached the GPU will hit a memory bottleneck leading to a non-linear increase in per-object costs. For all these reasons no perfectly objective per-object costs can be given in distance field rendering. The performance evaluations presented in this chapter only apply to our tested scene and different scenes and geometries will yield different results, within a certain margin. The aim of the test scene was to create a realistic scenario in order to get meaningful results. There were no geometry adjustments to better fit our optimization techniques. An in-depth look on our testing scene will be given and the results of our extensive performance tests will be highlighted. Finally, possible unexplored avenues for future optimizations will be discussed.

8.3.1 On Scene Composition

The default mTec scene used for all tests and performance evaluations is a closed room with pillars and a few other objects. The geometry of the room itself is entirely composed of implicit surfaces: the pillars are domain-repeated boxes and the walls and roof are created through prisms and various Boolean functions. As they are not actual 3D textures, they are handled in the renderer as a special case and are not included in the culling steps. When talking about “object count” in the scene, only regular mesh-based distance fields are taken into account. The room contains a Stanford dragon and Utah teapot model as well as a variable amount of humanoid figures. Whenever the object count is increased, more and more human models are placed in rows of seven behind each other. Even though the same base model is used, these are treated as separate entities by the renderer, and no data is re-used from already placed models. As far as the renderer is concerned, all the humanoid figures in the scene could be entirely different distance fields. Whenever an instancing scheme is used in order to save memory and increase performance for multiple copies of the same model it is explicitly mentioned in the section itself. The scene geometry and camera position was chosen in a way to present a “worst case” for performance evaluation. The whole room is in full view, and all objects are inside the camera frustum. The human models are pretty close together and create intersections in

culling and world id buffers. When moving the camera through the room, the average framerate is higher than in the fixed position used for measurements.
For further specifics on testing hardware please consult Appendix I. The following table highlights detailed information about the actual mesh SDFs used.

Model	*SDF Dimensions*	*Size (Mb)*	*Number of triangles*
Teapot	200x98x125	4.67	2464
Dragon	128x91x58	1.29	100.000
Human	295x350x50	9.85	2796

Even though the human mesh is a low-poly model, the generated distance field has the highest resolution by far. The correct size for a distance field is often independent of the underlying mesh, and should be chosen carefully on a per-object basis. The required GPU memory for the scene can be calculated as 4.67+1.29+9.87*(Objects-2) Megabytes. A performance test run with 20 objects has therefore a GPU memory requirement of about 183 Mb just for the 3D distance textures, a cost that can be alleviated by using object instancing for identical distance fields.
It might seem that there is a high memory overhead for distance field rendering, as each object is represented as a 3D texture. However, when compared with regular triangle meshes the costs aren’t that much different. The dragon model, for example, takes 1.29 Mb as a 3D texture. This is equal to about 14090 half precision vertices consisting of position and normal. The original dragon mesh model that was used for conversion consisted of 100.000 triangles. Overall the cost of distance fields as 3D textures is comparable to conventional meshes, and the truly relevant factor is the resolution chosen for the distance field.

8.3.2 Distance Field Instancing

It is not uncommon for video game developers to place many thousands objects in a scene that need to be rendered simultaneously. This is mostly the case with particle effects or dense vegetation, where the conventional approach of dispatching a draw call for each object would quickly lead to severe bottlenecks with such a big amount of calls. The solution to this problem came with the support of instancing in GPU hardware. Instancing, which has now been supported for over ten years[24], allows a single mesh to be rendered many times with different transformation data during a single draw call. An example for this would be the rendering of leaves on a tree. These leaves could all be based on the same mesh geometry, but drawn at different positions or modified otherwise through tessellation or deformation.

[24] https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_draw_instanced.txt

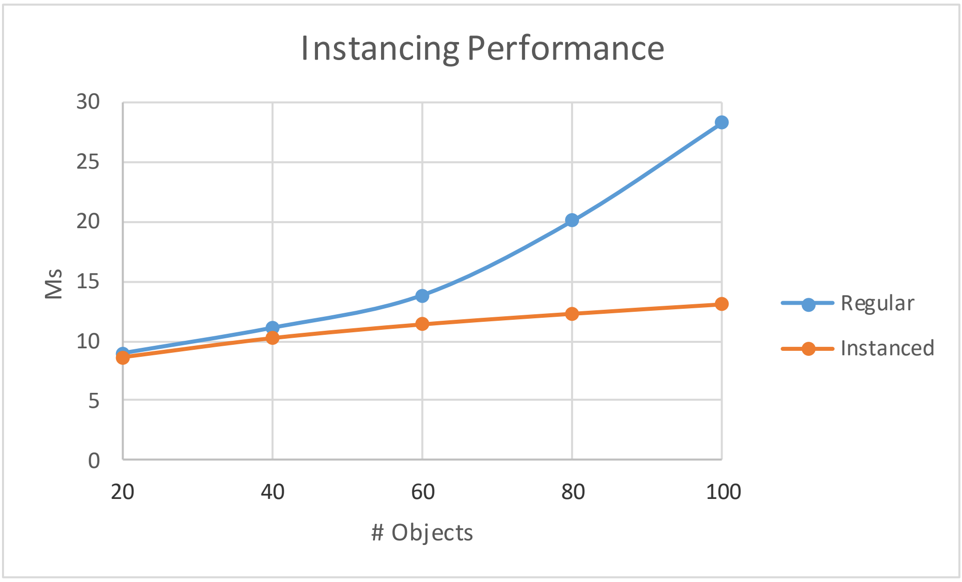


Figure 30: Render times plotted for various object counts.

The same basic principle can be applied to rendering with distance fields: each distance field texture that needs to be rendered has to be present in GPU memory, but what is more problematic is the fact that each texture has to be checked multiple times during sphere tracing. This puts a heavy burden on the GPU texture cache, which needs to re-fetch the relevant texture data continuously and can lead to an exponential increase in render times over a certain object threshold. The resulting loss of performance can be seen very well in Figure 30. An overall improvement can be achieved by keeping the memory footprint of each object as small as possible, packing meshes tightly into distance textures.
Instancing can further help with this issue. When multiple objects share the same distance field, the field can be uploaded to the GPU only a single time. The metadata of each object simply points to the same distance texture, therefore potentially saving a lot of memory.

Objects	*Regular*	*Instanced*
20	8.97	8.63
40	11.1	10.23
60	13.83	11.38
80	20.05	12.27
100	28.27	13.06

Table 5: Rendering times for various object counts. A full image with all effects is rendered for each measurement.

The rendering performance was measured for various object counts and can be seen in Figure 30 and Table 5. The instanced measurements were performed with 3 regular objects (dragon, teapot and human) and a varying amount of instanced versions of the human model.

For low object counts, there is almost no difference between instanced and non-instanced rendering. The performance gap starts to become apparent for object counts above 50. At this point even the most optimized version of the mTec renderer hits the aforementioned threshold at which each additional object leads to an exponential loss of performance. This issue is completely mitigated through object instancing: even when rendering many objects, the per-objects costs remain stable at about 0.05 ms.

8.3.3 Overall Performance

As the focus of our mTec renderer was the development of various optimizations, rendering speed was the topmost factor for evaluating the success of a particular approach. In order to put the final performance into better perspective, the current renderer will be compared to 6 other discrete steps during development. These steps are, in chronological order: naïve fragment shader, deferred compute shader, tiled culling, world SDF, undersampling, tighter buffers and the current iteration. Each following step builds on the optimizations of all previous steps unless noted otherwise. A brief overview will be given which optimizations each step encompasses before presenting the profiling results.

8.3.3.1 Naive Fragment Shader

The first full implementation supporting all distance field features, from shadows to sub-surface scattering was done in a single draw call. It calculated each image the most straightforward way, on a full-screen quad in a fragment shader. This was also the last iteration of mTec that could theoretically be run in WebGL/Shadertoy by switching all the distance textures to implicit geometry.

8.3.3.2 Deferred compute Shader

The first optimization step was the transfer from a single shader to the now used deferred compute shader pipeline. The final shading step was performed in a fragment shader due to the previously mentioned mipmapping issues. This was a simple split of the calculations and no further pre or post processing steps were performed. All algorithms ran in full resolution.

8.3.3.3 Tiled Culling

The tiled culling step encompasses the culling of distance fields to screen space tiles. The culling was limited to sphere tracing and AO, and shadows were still calculated naively. Furthermore the final shading step was optimized to use a single triangle instead of a quad.

8.3.3.4 World SDF

In this optimization step the world distance field calculation was implemented. The world field was used by sphere tracing and all image effects for speedups, but it was static in nature so no object animations could be performed. Even though they might seem similar, the world id field was not included in this step and was implemented much later.

8.3.3.5 Undersampling

The next big optimization step was to decrease the resolution of the rendered shadows and ambient occlusion. Shadows were decreased to half and AO to a quarter of the screen resolution. To improve the visual quality of the final image the buffers were upsampled using bilateral filtering.

8.3.3.6 Tighter Buffers

In order to improve the performance with more objects in the scene, the gBuffer and distance buffers were compacted and streamlined in this step. Furthermore proper culling was added for the shadow calculation. It was here that TAA was implemented, but as mentioned in Appendix I this was a purely visual improvement and was always disabled for performance measurements.

8.3.3.7 The Status Quo

The last big optimizations that were implemented were the world id field and rasterized sphere tracing. The rasterized tracing technique is a bit of a special case as described in chapter 8.4.2. It is disabled by default for the test cases presented here, as the conventional sphere tracing method displayed better performance for the tested object counts. Especially the early iterations of mTec were too slow to render anything beyond 20 objects, as shown in the evaluation below. A more specific comparison between the sphere tracing techniques at higher object counts is presented in the rasterized sphere tracing chapter.

8.3.3.8 Evaluation

For these comparative performance measurements the default scene was slightly modified by placing the human figures only in rows of three and moving them further back in the room hiding a few figures in higher object counts behind the backroom wall. This does not reduce calculation load by itself, but the more spread-out objects make it more unlikely that long culling lists are created. The reason for this change were the very long calculation times for the un-optimized versions of the renderer. The results of profiling can be seen in Table 6, Figure 32, Figure 31 and Figure 33. The performance of mTec has improved dramatically compared to the naive single shader version. Overall, the performance for 5 objects has improved by 500%, while the performance for 20 object has improved by almost 4000%.

Objects	*Single Shader*	*Compute Deferred*	*Tiled Culling*	*World Sdf*	*Undersampling*	*Tighter Buffers*	*Current*
5	36.37	30.14	19.7	15.86	9.45	7.76	7.19
10	70.54	63.06	32.05	29.05	21.1	9.13	7.22
15	134.1	125.35	63.19	94.16	29.67	10.89	7.33
20	298.52	319.82	154.49	160.66	59.94	12	7.53

Table 6: Comparison of optimizations.

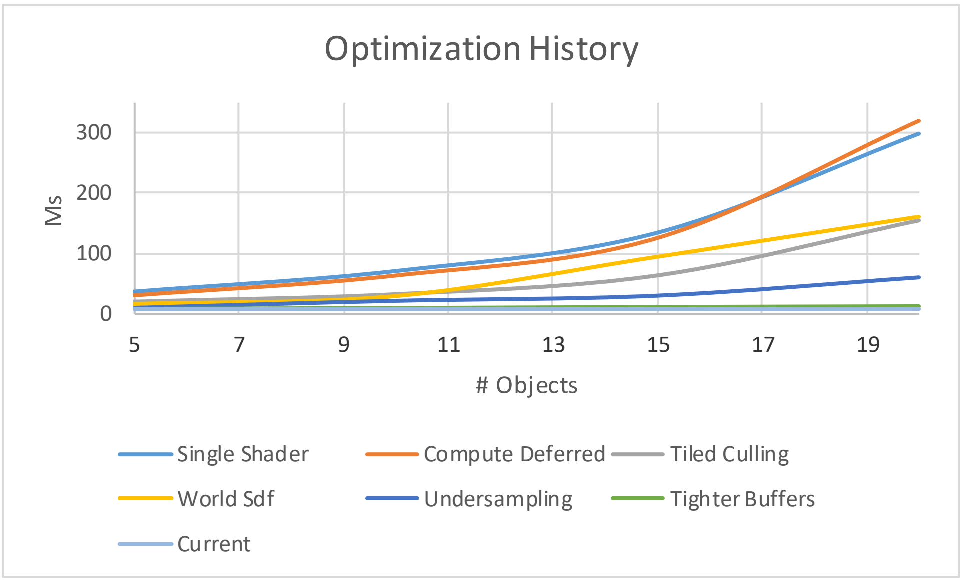
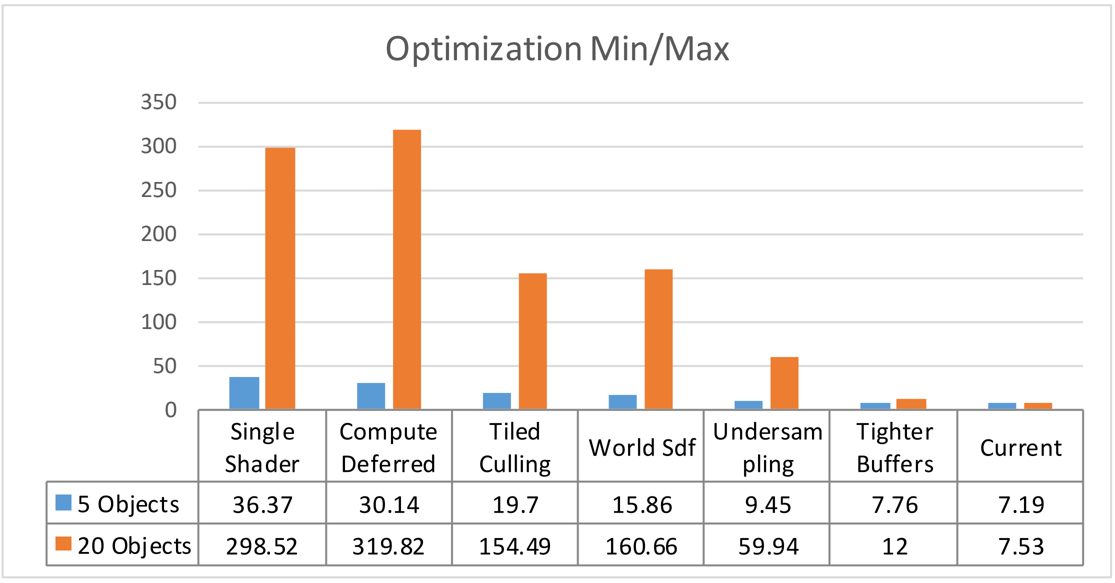


Figure 32: The results from Table 6 visualized as a line graph. The bend at which additional object costs become non-linear is clearly visible.



	Single Shader	Compute Deferred	Tiled Culling	World Sdf	Undersampling	Tighter Buffers	Current
🟦 5 Objects	36.37	30.14	19.7	15.86	9.45	7.76	7.19
🟧 20 Objects	298.52	319.82	154.49	160.66	59.94	12	7.53

Figure 31: The performance results from Table 6 with their min/ max values highlighted.

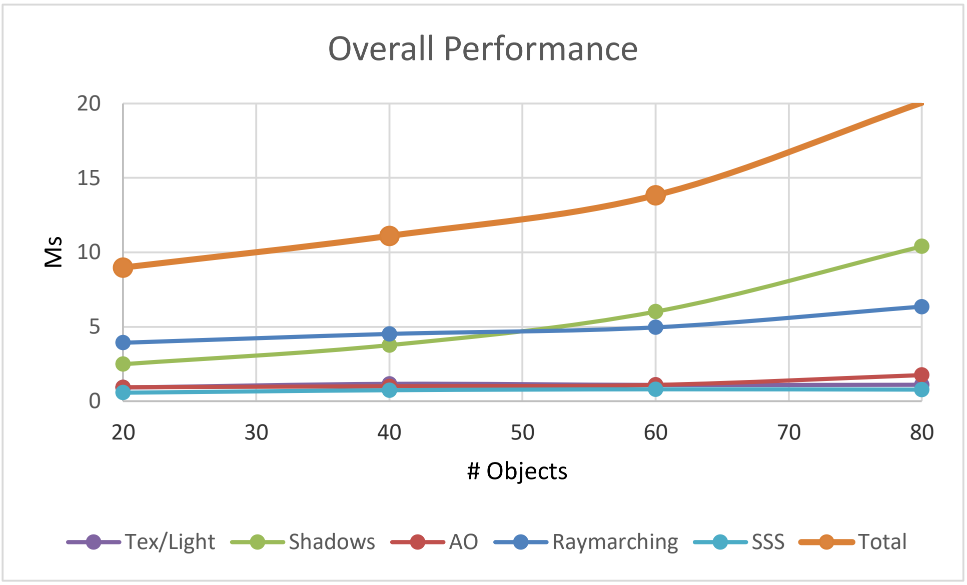


Figure 33: The rendering times of all modules in the current version of mTec, for better comparison. The data is taken from the respective module chapters. The „Total“ entry denotes the actual measured rendering time of the renderer running with all features (except TAA) enabled and not the sum of the other measurements.

8.4 Abandoned Optimizations

During the whole development process of mTec a wide array of other optimizations were implemented, evaluated and eventually abandoned. The techniques described hereafter have proven either less robust or simply slower than what eventually ended up in the main renderer. Nevertheless the ideas behind them might prove very interesting topics for future research.

8.4.1 Rasterized Distance Field Culling

Regular distance field culling is performed by dispatching a compute shader in 1/8 of the screen dimensions and checking for intersections with all the scene distance fields. The same could be achieved through rasterization. Object fields could be wrapped in bounding boxes, and then a draw call could be dispatched for each box. This box would then be rasterized in the rendering pipeline and a fragment shader would be dispatched for all the fragments of the object on screen. The shader program for generating such a culled list is very simple.

```
void main()
 {
      ivec2 coords = ivec2(gl_FragCoord.xy);
      int storePos = coords.x+coords.y*renderWidth*tileLength;
      int prev = atomicAdd(ids[storePos],1);
      culledSDFs[storePos+prev+1] = SDFIndex;
 }
```

The id of the currently drawn object has to be added to the culling list, and the number of objects for the current tile has to be increased through an atomic add. In the end, after all distance fields were drawn, the resulting culling list would be identical to the single pass compute shader approach. This approach was implemented and tried for culling, but in the end abandoned as there were no clear performance benefits over the single pass approach. In the default scene described above the rasterization approach results in a lot of overdraw, as there may be up to 8 completely overlapping boxes that need to be drawn. Regular depth testing cannot be used, as these boxes are just conservative bounds and do not account for holes or concave features of the underlying distance fields. In comparison the simple intersection tests performed by the compute shader were faster than separate draw calls.
Even though this approach has proven slower in mTec and was eventually abandoned, it may be worth evaluating depending on the used scene geometry. When employing general frustum culling this method has a few upsides over the single dispatch approach. The CPU-culled list for the camera frustum would need to be transferred to the GPU for the compute shader approach. When using multiple draw calls, the information could stay on the CPU side.

8.4.2 Rasterized Sphere tracing

Conventionally, depth finding in a distance field scene is performed by sphere tracing a ray for each on-screen pixel. This means that at all scene fields have to be checked over and over again, thrashing the GPUs texture cache. An alternative approach which could alleviate many of the issues exhibited by sphere tracing was developed for mTec. This technique is quite similar to the rasterized culling algorithm described in the previous chapter and was inspired by Evan‘s brick rendering approach [15]. It breaks the single sphere tracing pass down into separate draw calls: in each draw call the bounding box of a single distance field is drawn as actual geometry and traced (Figure 34). Because only the contents of the currently rendered box need to be traced, the tracing algorithm itself can be optimized more aggressively. The ray does not need to start from the camera, but rather from the pixel position in world-space, which is the ray’s intersection with the bounding box. This allows for the skipping of potentially a lot of empty space. Furthermore, the algorithm can be terminated as soon as the ray exits the box. Calculating this second intersection is also much easier than it would be conventionally. There is no need for transforming each

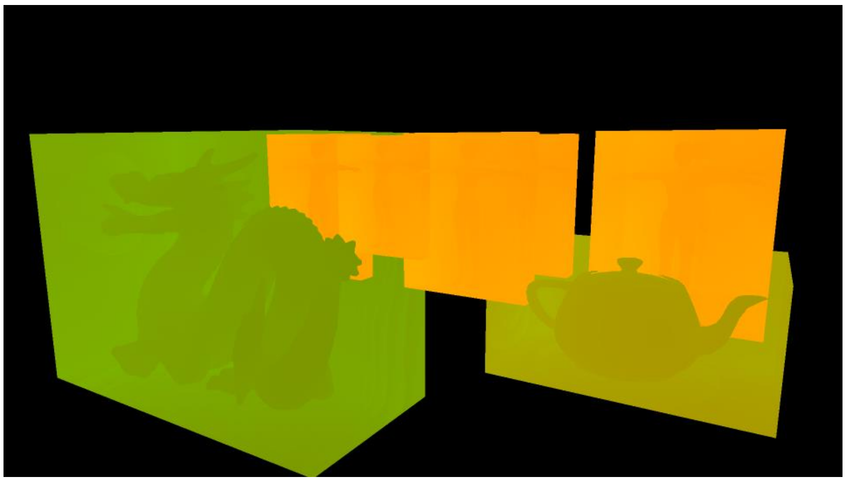


Figure 34: Rasterized bounding boxes containing the sphere traced fields.

point on the ray into the corresponding texture space before sampling the field, as only a single field is traced each draw call. The entire ray can be transformed into texture space at the beginning of the algorithm and the entire algorithm can be run in this space. At the end of the calculation, the resulting surface point simply needs to be transformed back into world space. This reduces the amount of matrix multiplications that have to be performed for each object dramatically. Furthermore, it can be easily detected if the ray has passed through the bounding box. As long as the current point on the ray lies within the [0,1] bounds on all three axes, it is still valid.
The following code is effectively a point-AABB containment test in texture space.

```
vec3 clamped = vec3(clamp(P,0,1));
if(lengthSqr(P-clamped) > 0)
{
      break; //outside of box
}
```

This is much easier to compute than a conventional OBB-Ray intersection test would be.

Objects	*Fullscreen*	*Fullscreen-inst*	*Rasterized*	*Rasterized-inst*
20	3.93	3.93	4.13	4.05
40	4.52	4.33	4.53	4.39
60	4.96	4.63	5.06	4.8
80	6.35	4.93	5.63	4.99
100	7.82	5.3	6.06	5.92

Table 7: Sphere tracing times with various algorithms. For these measurements only depth and normal was calculated for each pixel.

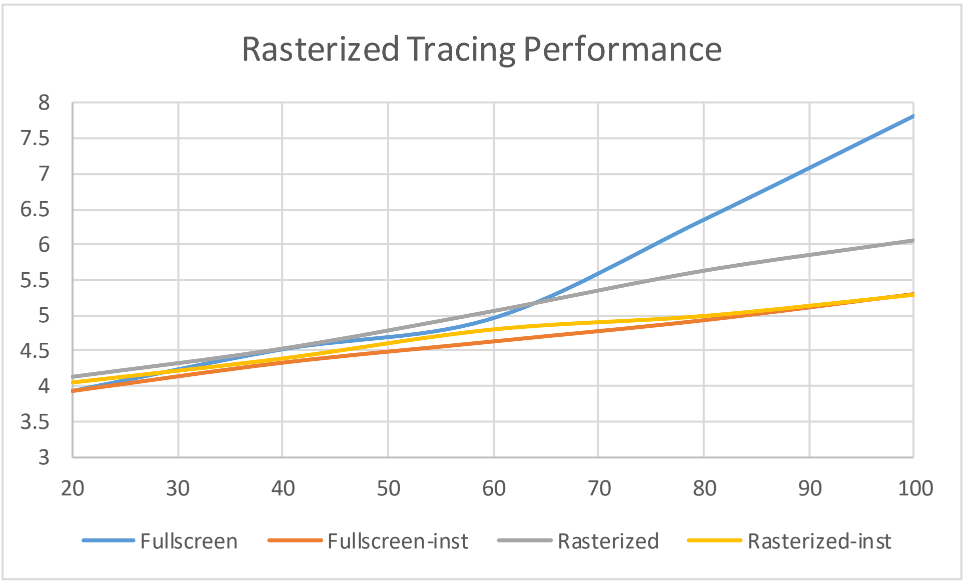


Figure 35: Sphere traced performance from Table 7 plotted on a graph.

Each object is sphere traced by itself, so there can be no interferences from nearby objects, which allows for the maximum step count to be reduced severely without getting the well-known sphere tracing artefacts from rays passing close by an unrelated object before hitting a surface. Artefacts from large, concave objects can still occur, but these are much rarer, and the technique can allow for different step thresholds on a per-object basis. Furthermore, a much nicer cache coherency is achieved by only needing to access a single texture per draw call. This rasterized approach has many interesting implications for other techniques such as the rendering of semi-transparent objects. Transparency is rather hard with distance fields, as each transparent object would need to be traced and shaded in its own separate full-screen step in order to ensure correct blending between multiple semi-transparent objects. When rendering only bounding boxes, semi-transparent objects can be simply added on top of the previously rendered scene in the same fashion as conventional alpha-blended scenes. Another possibility would be hybrid rendering, where the whole scene is a combination of meshes and distance fields. This technique would only require small changes to the renderer itself by simply adding a separate shader for sphere tracing.

There are a few issues with this technique which are the main reason that full-screen sphere tracing is still used as the main depth finding technique in mTec. The first and foremost issue, which has the biggest impact on performance is overdraw. In the mTec sample scene there are a lot of objects directly behind each other which requires these parts of the scene to be rasterized and drawn over and over again, even though the results might be discarded. In conventional renderers this is handled by a depth pre-pass and rejection of occluded fragments through early depth testing. Unfortunately this is much harder with rasterized sphere tracing. When a depth buffer and depth testing is enabled, the GPU wants to write the fragment's original depth into the buffer. In our case this is the

surface of the bounding box and not the object contained within. The value that will be written to the depth buffer can be overridden manually by assigning a value to the built-in gl_FragDepth variable, which allows us to set the correct depth value after sphere tracing. This manual setting of the depth, however, leads to OpenGL automatically disabling early depth testing. The GPU cannot be „lied“ to, by making it pass an early depth test and then modifying the depth value which could lead to the rejection of the fragment. Fortunately an extension exists which can be used to tackle this problem. This extension is ARB_conservative_depth: it allows the user to specify whether the depth manually set in the shader will only be smaller, larger or equal to the initial value determined by rasterization. By specifying this inequality the GPU can perform better optimizations of the code which leads to a better overall performance.
Another issue arises when the bounding box intersects the camera near plane. In that case the triangles are clipped, and no sphere tracing is performed. This problem can be alleviated by enabling depth clamping. With depth clamping enabled triangles will not get clipped at the near or far plane, but the problem still persists if the camera happens to be inside the rendered volume. Further research will be necessary into the issue in order to find a robust method to ensure that the boxed objects are rendered properly at all times. A possible solution could be for the camera to detect if it is inside a distance volume and alter the sphere tracing algorithm to reconstruct the correct initial marching positions from the back faces of the bounding box.
The main reason why this approach is slower than our full-screen pass lies in the composition of our scene. The explicit distance objects cover only about 30% of the scene from the default viewing point, while the rest consists of the implicitly calculated room and columns. Even after all the object bounding boxes have been sphere traced, another full-screen pass has to be performed just to sphere trace the implicit geometry covering the room. This results in the complete loss of all performance previously gained by this technique. When rendering just the explicit 3D textures, the whole algorithm takes about 2.05 ms per frame, compared to 3.83 of the single pass. The single pass however, has very similar render times with and without the background as each pixel has to be traced anyway. So when rendering the whole scene, rasterized sphere tracing takes 4.13 ms while the single pass only needs 3.93 ms for 20 objects. This may seem like a knock-out criterion for this technique, but extensive performance measurements (Table 7) reveal a different picture. Even though rasterized tracing starts out slower than the optimized full screen trace, it overtakes the conventional technique at about 60 rendered objects. When these results are visualized in Figure 35, it can be easily seen why this is the case: the regular trace starts to hit the texture cache threshold and costs become exponential while rasterized tracing does not suffer from the same issue. Each traced field is a separate draw call and only a single 3D distance field has to be kept in the texture cache. So while this novel technique has a higher overhead to start with, it yields much better results for dense scenes. When using object instancing, both algorithms produce very similar rendering times. This leads to the conclusion that rasterized sphere tracing is a very promising

approach for scenes with high object counts, but more performance can be gained still by re-using as many objects in the scene as possible.
Another good fit for boxed tracing might be hybrid renderers, where most screen elements are meshes mixed with a few distance field objects. In such cases rendering would be much faster and allow for easy integration into pre-existing pipelines.
There are still many avenues for the optimization of this technique to be explored. Our full-screen sphere tracing pass would be much slower than rasterized tracing if it were not for culling, world distance and id fields. The rasterization approach does not use any of these acceleration structures as the underlying algorithms are fundamentally different. Therefore any acceleration techniques that could be fitted to this algorithm could make it significantly faster even for small object counts in pure distance field engines.
To summarize, rasterized sphere tracing is an alternative algorithm to classical full screen tracing. It has some very interesting properties which can mitigate the shortcomings of more classical tracing approaches. Nevertheless there are still some issues which require further research which is why the technique is supported by mTec, but disabled by default.

9 Discussion

There are still many unsolved problems in distance field rendering. Sphere tracing, one of the most essential algorithms for utilizing SDFs is not 100% robust, and many artefacts still pop up and have to be addressed individually, depending on the rendered scene and the objects themselves. The novel rasterized sphere tracing technique presented in this thesis can help in mitigating some of these issues, but has some quirks by itself. Widely accepted techniques for distance field animations or texturing do not exist yet, and the techniques presented in this thesis are functional, but not robust enough or too narrow for the production of modern videogames. These are still very much open research topics in the distance field community. The focus of this thesis lies in the development of acceleration techniques for the most popular distance field rendering effects. It has been shown that by employing a few common techniques, the rendering performance can be increased significantly. Still, there is a lot more room for improvement of these and other techniques. A lot of performance could be gained by performing micro-optimizations of the shaders developed for mTec. A more in depth analysis of the shaders could identify GPU bottlenecks and refactoring of the algorithms to take into account hardware features such as coalesced memory access or GPU load-balancing could result in vast performance gains. The rendering is very much GPU bound as many different shaders are run every frame. Even small improvements in the amount of shader operations or registers used can have a relevant impact on performance. For larger scenes, well-established techniques such as frustum culling could be utilized to increase the rendering and tile-culling performance. Unfortunately many of the common spatial acceleration structures used for CPU algorithms are very hard to properly utilize in GPU calculations. Most hierarchical tree structures require bookkeeping of some sorts during traversal, but shader units themselves have a rather small amount of registers which limits the data that can be kept in memory efficiently. In order to accelerate sphere tracing calculations a spatial data structure would need to have a very low memory overhead and require as little texture reads as possible, as those are already a choke point.

There is a wide variety of future research directions for the mTec renderer, from simple optimizations to entirely new rendering techniques. The implications of distance fields in combination with temporal reprojection and motion blur are intriguing, and would definitely warrant deeper research. Temporal amortization in particular could pose a potential way of increasing the general robustness of distance field rendering.

The application of distance fields for efficient real-time rendering is still a very underdeveloped field. There are very many papers on the properties, calculation and essential techniques of SDFs, but very few that bring these techniques into a modern, performance driven context. This thesis should give a solid basis on the current status-quo of distance field rendering, and highlight possible avenues for future research into this fascinating topic.

As it stands fully-fledged distance field rendering is still not robust or fast enough for large-scale commercial video game productions.
Still, the simplicity of these techniques as compared to today’s industry standards make them compelling for rendering purposes. Single techniques such as ambient occlusion have a much lower performance impact and are already in use in at least one big commercial engine. As time progresses, algorithms and rendering hardware will improve, making efficient hybrid or full-on distance field rendering in game development only a matter of time. This author strongly believes that SDFs will find their home in game engines in the days to come.

10 Appendix I: On Testing

Unless noted otherwise, all performance measurements were performed on a Windows 7 PC, with an Intel Xenon E5606 Quad core CPU, an Nvidia Geforce GTX 980 GPU with the 347.62 display driver, and 8 GB of RAM. A screen resolution of 1920x1080 was used for all rendering tests. The frame rendering time is measured using the QueryPerformanceCounter function from the Windows API. In order to get more reliable results, the rendering times are averaged over a thousand successive frames. All results are given in milliseconds (ms), and exclude the post-processing step/TAA which has a fixed cost of about 0.18 ms, as explained in the post-processing chapter.

References

[1] Jones, M. W., Baerentzen, J. A., & Sramek, M. (2006). 3D distance fields: A survey of techniques and applications. *IEEE Transactions on visualization and Computer Graphics*, *12*(4), 581-599.

[2] Erleben, K & Dohlmann, H 2008, Signed Distance Fields Using Single-Pass GPU Scan Conversion of Tetrahedra. in H Nguyen (ed.), *GPU Gems 3.* Addison-Wesley, Upper Saddle River, N.J., pp. 741-763.

[3] Varadhan, G., Krishnan, S., Kim, Y. J., Diggavi, S., & Manocha, D. (2003, June). Efficient max-norm distance computation and reliable voxelization. In *Symposium on geometry processing* (pp. 116-126).

[4] Danielsson, P. E. (1980). Euclidean distance mapping. *Computer Graphics and image processing*, *14*(3), 227-248.

[5] Hart, J. C., Sandin, D. J., & Kauffman, L. H. (1989, July). Ray tracing deterministic 3-D fractals. In *ACM SIGGRAPH Computer Graphics* (Vol. 23, No. 3, pp. 289-296). ACM.

[6] Zuiderveld, K. J., Koning, A. H., & Viergever, M. A. (1992, September). Acceleration of ray-casting using 3-D distance transforms. In *Visualization in Biomedical Computing* (pp. 324-335). International Society for Optics and Photonics.

[7] Daniel E. Koditschek, "Robot Planning and Control Via Potential Functions",*The Robotics Review* , 349-367. January 1989.

[8] Gibson, S. F. F. (1998, October). Using distance maps for accurate surface representation in sampled volumes. In *Volume Visualization, 1998. IEEE Symposium on* (pp. 23-30). IEEE.

[9] Frisken, S. F., Perry, R. N., Rockwood, A. P., & Jones, T. R. (2000, July). Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (pp. 249-254). ACM Press/Addison-Wesley Publishing Co..

[10] Quilez, I. 2008. Rendering Worlds with Two Triangles with raytracing on the GPU in 4096 bytes. http://iquilezles.org/www/material/nvscene2008/rwwtt.pdf ,Retrieved 30.11.2015

[11] Fuhrmann, A., Sobotka, G., & Groß, C. (2003, September). Distance fields for rapid collision detection in physically based modeling. In *Proceedings of GraphiCon 2003* (pp. 58-65).

[12] Green, C. (2007, August). Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 courses* (pp. 9-18). ACM.

[13] Carvalho, C. (2002). The gap between processor and memory speeds. In *Proc. of IEEE International Conference on Control and Automation*.

[14] Wright, D. (2015). Dynamic Occlusion with Signed Distance Fields. Advances in Real-Time Rendering, SIGGRAPH 2015. http://advances.realtimerendering.com/s2015/DynamicOcclusionWithSignedDistanceFields.pdf . Retrieved 19.04.2017

[15] 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, SIGGRAPH 2015. http://media.lolrus.mediamolecule.com/AlexEvans_SIGGRAPH-2015.pdf . Retrieved 19.04.2017

[16] Akleman, E., & Chen, J. (1999, March). Generalized distance functions. In Shape Modeling and Applications, 1999. Proceedings. Shape Modeling International'99. International Conference on (pp. 72-79). IEEE.

[17] Strain, J. (1999). Fast tree-based redistancing for level set computations. *Journal of Computational Physics*, *152*(2), 664-686.

[18] Nooruddin, F. S., & Turk, G. (2003). Simplification and repair of polygonal models using volumetric techniques. *IEEE Transactions on Visualization and Computer Graphics*, *9*(2), 191-205.

[19] Jones, M. W. (1995). 3D distance from a point to a triangle. Department of Computer Science, University of Wales Swansea Technical Report CSR-5.

[20] Baerentzen, J. A., & Aanaes, H. (2005). Signed distance computation using the angle weighted pseudonormal. *IEEE Transactions on Visualization and Computer Graphics*, *11*(3), 243-253.

[21] Max, N. (1999). Weights for computing vertex normals from facet normals. *Journal of Graphics Tools*, *4*(2), 1-6.

[22] Borgefors, G. (1986). Distance transformations in digital images. *Computer vision, graphics, and image processing*, *34*(3), 344-371.

[23] Grevera, G. J. (2004). The "dead reckoning" signed distance transform. *Computer Vision and Image Understanding*, *95*(3), 317-333.

[24] Kessler. (2010). Go With The Flow: Fluid and Particle Physics in PixelJunk Shooter. Game Developers Conference 2010.

[25] Swoboda, M. Advanced Procedural Rendering in DirectX 11.

[26] Tomczak, L. J. (2012). GPU Ray Marching of Distance Fields. *Technical University of Denmark*.

[27] Hart, J. C. (1996). Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, *12*(10), 527-545.

[28] Perlin, K., & Hoffert, E. M. (1989, July). Hypertexture. In *ACM SIGGRAPH Computer Graphics* (Vol. 23, No. 3, pp. 253-262). ACM.

[29] McReynolds, T., Blythe, D., Fowle, C., Grantham, B., Hui, S., & Womack, P. (1997). Programming with OpenGL: advanced rendering. In *SIGGRAPH* (Vol. 97, pp. 144-153).

[30] Blinn, J. F. (1978, August). Simulation of wrinkled surfaces. In *ACM SIGGRAPH computer graphics* (Vol. 12, No. 3, pp. 286-292). ACM.

[31] Geiss, R. (2007). Generating complex procedural terrains using the GPU. *GPU gems*, *3*, 7-37.

[32] Lorensen, W. E., & Cline, H. E. (1987, August). Marching cubes: A high resolution 3D surface construction algorithm. In *ACM siggraph computer graphics* (Vol. 21, No. 4, pp. 163-169). ACM.

[33] Jamriška, O. (2010, May). Interactive ray tracing of distance fields. In *The 14th Central European Seminar on Computer Graphics* (Vol. 2).

[34] Kajiya, J. T. (1986, August). The rendering equation. In *ACM Siggraph Computer Graphics* (Vol. 20, No. 4, pp. 143-150). ACM.

[35] Akenine-Möller, T., Haines, E., & Hoffman, N. (2008). *Real-time rendering*. CRC Press.

[36] Bavoil, L., & Sainz, M. (2008). Screen space ambient occlusion. *NVIDIA developer information: http://developers. nvidia. com*, *6*.

[37] Grosch, T., & Ritschel, T. (2010). Screen-space directional occlusion. *GPU Pro*, 215-230.

[38] Evans, A. (2006, July). Fast approximations for global illumination on dynamic scenes. In *ACM SIGGRAPH 2006 Courses* (pp. 153-171). ACM.

[39] Barre-Brisebois, C., & Bouchard, M. (2011). Approximating translucency for a fast, cheap and convincing subsurface-scattering look. In *Game developers conference* (Vol. 6).

[40] Pedersen, L.,J.,F., Temporal Reprojection Anti-Aliasing in Inside, Game Developers Conference 2016, https://github.com/playdeadgames/temporal/blob/master/GDC2016_Temporal_Reprojection_AA_INSIDE.pdf Retrieved 19.04.2017

[41] Sigg, C., Peikert, R., & Gross, M. (2003, October). Signed distance transform using graphics hardware. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)* (p. 12). IEEE Computer Society.

[42] Sud, A., & Manocha, D. (2003). Fast distance field computation using graphics hardware. *Tech. Rep.*

[43] Rong, G., & Tan, T. S. (2006, March). Jump flooding in GPU with applications to Voronoi diagram and distance transform. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games* (pp. 109-116). ACM.

[44] Herzog, R., Eisemann, E., Myszkowski, K., & Seidel, H. P. (2010, February). Spatio-temporal upsampling on the GPU. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games* (pp. 91-98). ACM.

[45] Keinert, B., Innmann, M., Sänger, M., & Stamminger, M. (2015). Spherical fibonacci mapping. *ACM Transactions on Graphics (TOG)*, *34*(6), 193.

[46] McGuire, Morgan. "The Graphics Codex." (2013).

Table of Figures

Table of Tables

Table of Abbreviations

SDF	Signed Distance Field
DF	Distance Field
FPS	Frames Per Second
CSG	Constructive Solid Geometry
CDT	Chamfer Distance Transform
ULP	Unit in the Last Place
TAA	Temporal Anti-Aliasing
SSS	Sub-Surface Scattering
AO	Ambient Occlusion
SSAO	Screen-Space Ambient Occlusion
GI	Global Illumination
ms	Milliseconds
SIMD	Single Instruction Multiple Data
PBR	Physically Based Rendering

Table of Assets

Assets	*Source*	*Url*
Dragon Mesh	The Stanford Scanning Repository	http://graphics.stanford.edu/data/3Dscanrep/
Teapot Mesh	The Utah Teapot	http://goanna.cs.rmit.edu.au/~pknowles/models.html
Human Mesh	Cgtrader	https://www.cgtrader.com/dev-piplay
Floor Texture	Daniel Mccann	http://mccannd.blogspot.com
Wall Texture	Substance share	https://share.allegorithmic.com/libraries/1169
Pillar Texture	Substance share	https://share.allegorithmic.com/libraries/1425
Human Texture	freepbr	http://freepbr.com/materials/greasy-worn-pbr-metal-material-1/
Teapot Texture	freepbr	http://freepbr.com/materials/rusted-iron-pbr-metal-material/
Dragon Texture	freepbr	http://freepbr.com/materials/polished-speckled-marble-top-pbr-material/