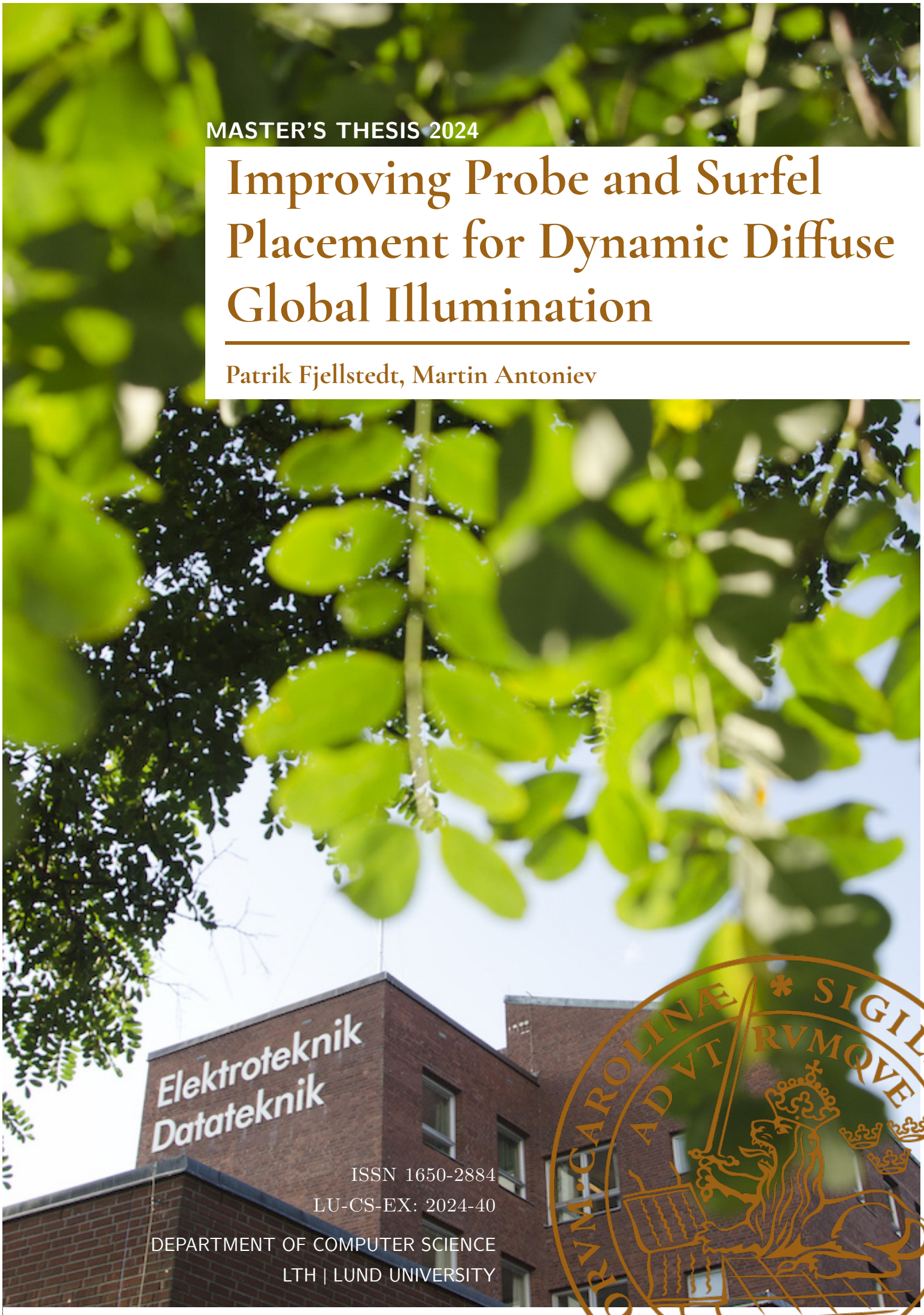


MASTER'S THESIS 2024

Improving Probe and Surfel Placement for Dynamic Diffuse Global Illumination

Patrik Fjellstedt, Martin Antoniev

ISSN 1650-2884
LU-CS-EX: 2024-40

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2024-40

Improving Probe and Surfel Placement for Dynamic Diffuse Global Illumination

Förbättrad placering av ljussonder och ytelement för dynamisk diffus global belysning

Patrik Fjellstedt, Martin Antoniev

Improving Probe and Surfel Placement for Dynamic Diffuse Global Illumination

Patrik Fjellstedt
pa2747fj-s@student.lu.se

Martin Antoniev
ma3468an-s@student.lu.se

June 25, 2024

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Michael Doggett, michael.doggett@cs.lth.se
Calle Lejdfors, calle.lejdfors@gmail.com

Examiner: Per Andersson, per.andersson@cs.lth.se

Abstract

The computational complexity imposed by accurately estimating, calculating and simulating dynamic diffuse global illumination (DDGI) in real-time computer graphics has led to the development of a multitude of different approaches. This thesis extends upon an existing system which uses statically placed light probes and surface elements (surfels) to discretize and calculate the scene DDGI, and presents a method for dynamically distributing and placing the light probes in the scene based on its complexity. The new system uses an octree data structure whose branches are iteratively built and rebuilt depending on the geometric density and changes in the scene, and where the light probes are placed in its leaf nodes. Each probe evenly places surfels around itself using a Fibonacci lattice to determine both indirect lighting contributions, which are then used to calculate the scene DDGI, and whether to rebuild the octree at the node in which the light probe resides. The new system was able to significantly reduce the number of probes in a test scene from 9261 to 30 while maintaining visual quality, albeit at the expense of an increased frame time.

Keywords: Octrees, Probes, Surfels, Real-Time, Computer Graphics, Ray Tracing

Acknowledgements

We would like to thank our supervisors Michael Doggett and Calle Lejdfors for their guidance, support and invaluable feedback throughout the project. We would also like to thank the author of the original work, Elmer Dellson, who provided us with a starting platform from which we could build upon. Finally, we would like to thank our friends and family for their support and encouragement.

Contents

Chapter 1
Introduction

Lighting a three-dimensional scene in computer graphics in a convincing way remains a long-standing challenge. One critical aspect of this challenge is global illumination, which aims to simulate the complex interactions of light as it bounces off of surfaces in the scene. Global illumination is essential for creating realistic images, as it captures the indirect lighting effects that are not captured by direct lighting alone. In this thesis we focus on the problem of real-time, scene-adaptive, global illumination through diffuse reflections, collectively known as dynamic diffuse global illumination (DDGI)

Building upon an existing system that uses light probes (probes) to capture light information within a region of space, combined with surface elements (surfels) to approximate indirect lighting, this thesis aims to improve the placement of the probes and surfels in the scene. The existing system employs a dense and regular grid to place the light probes, which can be inefficient in terms of the number of probes.

We propose an improvement by distributing the probes hierarchically with an octree data structure, based on the amount of geometry around each probe. This allows for a more adaptive and sparse placement of probes based on the scene complexity without manual intervention. In addition to the octree we also incorporate a Fibonacci lattice and compare this with the latitude-longitude lattice used in the previous system, as a method for placing the surfels.

By dynamically adjusting the probe placement based on the scene complexity, we aim to reduce the number of probes placed in the scene while maintaining a similar, or producing a better visual quality compared to the previous system.

1.1 Research Questions

This thesis aims to answer the following research questions:

- How well does the new system for probe and surfel placement reduce the number of

placed probes while retaining a similar, or producing a better, visual quality compared to the previous system in a more complex environment?

- How significantly does the choice of spherical lattice affect the surfel placement?

1.2 Project Scope

In order to investigate our research questions posed in section 1.1, we limited our project scope. Initially we planned to extend the complexity of the scene further by introducing additional lights and more advanced materials, however due to time constraints we decided to focus on the diffuse part of surfaces and only introduced a more geometrically complex object. We also did not consider more than the second order light contributions.

1.3 Contribution

The contributions to the state of knowledge that this thesis provides can be summarized as follows:

- A method for dynamically and hierarchically determine where and when to place light probes in a three-dimensional scene in a scalable way.
- An implementation and representation of an octree, a recursive data structure, for easy interpretation on the graphics processing unit.

Chapter 2
Background and Theory

This chapter covers relevant background and theory used in this thesis.

2.1 Real-Time Computer Graphics

Real-time computer graphics is a subfield of computer graphics that imposes a hard constraint on the rendering time of images, such that they are interactive. This typically means that the images have to be generated at 30 frames per second or higher and is used in applications such as video games. In order to achieve this type of performance an effective combination of hardware and software must be used. This also leads to a variety of techniques and optimizations that might not be necessary in other aspects of computer graphics, such as offline rendering, and often means a trade-off between different factors such as visual quality and performance.

The fundamental goal when producing an image is to calculate the color of each pixel. There are a multitude of ways to achieve this, such as rasterization, ray tracing, path tracing, and many more. In this thesis we are only concerned with ray tracing, which is a method that has gained popularity in recent years, in no small part due to the access of hardware capable of ray tracing becoming more available. In this method rays are cast from the camera into the scene for each pixel and are traced until they hit a surface or reach a maximum depth.

2.2 The Rendering Equation

The rendering equation was formulated concisely by Kajiya in 1986 [7], and has since become somewhat of a cornerstone for computer graphics when it comes to rendering images realistically. There are different ways of writing the equation, but perhaps the simplest and most straightforward is as it appears in the original paper, shown in equation (2.1).

$$I(x, x') = g(x, x')\left[\epsilon(x, x') + \int_S \rho(x, x', x'')I(x', x'')dx''\right] \tag{2.1}$$

The rendering equation is based on the physical law of conservation of energy, and states that the incoming light for a surface point is the sum of the emitted light and scattered light from all other surfaces toward the surface point [7]. The quantity $I(x, x')$ relates to the light intensity at point x from point x', and $g(x, x')$ is a "geometry" term which is zero if x and x' are not mutually visible or r^{-2} otherwise, where r is the distance between the two points. The quantity $\epsilon(x, x')$ relates to the light intensity at point x, emitted from point x', whereas the quantity $\rho(x, x', x'')$ relates to the light intensity at point x, scattered at point x' from point x''.

Scattered light reaching point x may be scattered at any point x' on all surfaces, S, and originate from any other point x''. This makes the rendering equation recursive and, consequently, unfeasible to solve in real-time for every point on every surface in a scene. Instead, certain simplifications are applied to the recursive part of the rendering equation, such as limiting the number of considered scatterings and directions. Furthermore, the two terms of the rendering equation, $g(x, x')\epsilon(x, x')$ and $g(x, x') \int_S \rho(x, x', x'')I(x', x'')dx''$, can be expressed in terms of direct and global illumination.

2.2.1 Direct Illumination

Direct illumination refers to the light reaching a surface, without being obstructed, directly from a light source [1], and corresponds to the first term in equation (2.1). In the context of ray tracing, this means that when a ray hits a surface, and the hit point is visible from the perspective of the light source, the hit point on the surface is directly illuminated by the light source. Equivalently, light rays from the light source which reflect once off of a surface and reach an observer, directly illuminate the surface from the perspective of the observer.

2.2.2 Global Illumination

Global illumination, in contrast to direct illumination, encompasses all light which reaches a surface indirectly, and corresponds to the second term in equation (2.1). As such, it can be considered to describe all light reaching a surface except for that described by direct illumination. In particular, global illumination accounts for light from a light source which has been reflected at least once before reaching a surface that is being observed. Equivalently, global illumination is light which has been reflected at least twice before reaching the observer [1].

2.3 Scene Discretization

A *scene*, in the context of computer graphics, encompasses everything required in a three-dimensional world for an image to be rendered in a desirable way [1]. For a scene, or parts thereof, to be rendered efficiently in real-time, certain discretizations and simplifications can be implemented and applied to make difficult calculations and representations simpler and more manageable.

2.3.1 Octrees

An *octree* is a recursively defined tree data structure that can be used to hierarchically partition and organize three-dimensional space into *octants*, where an octant has the primitive shape of a cube [9]. Each *node* in an octree has either zero or exactly eight *children*, known as each other's *siblings*. A node which has no children is a *leaf* node, while a node which has children is a *parent* node. All nodes in the octree have a parent except for the *root* node which is at the top of the hierarchy. The *depth* of a node is the distance to the root in number of edges, giving the root node a depth of zero.

Figure 2.1 shows an example of a hierarchical partitioning of three-dimensional space into octants and a corresponding possible octree to represent the partitioning. The figure illustrates how the octant for the root node subsumes all other partitions, as well as how each child covers one eight the volume of its parent.

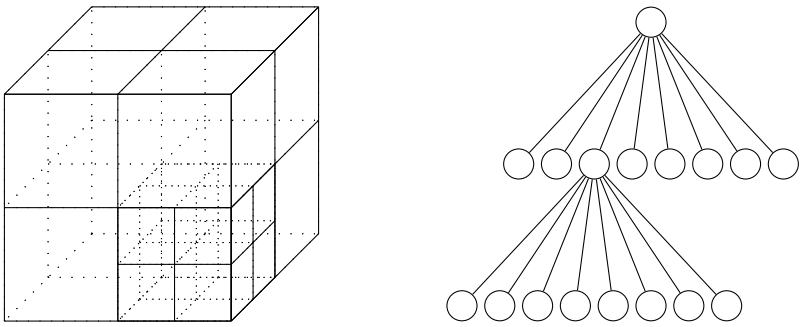


Figure 2.1: An example partitioning of three-dimensional space into octants and a possible octree representation for the partitioning.

2.3.2 Light Probes

In simple terms, a *light probe* (probe) is a data structure which can be viewed as an omnidirectional camera that captures and encodes how light interacts and passes through a particular point in a three-dimensional scene [16, 5]. There are different ways of encoding the aforementioned light information, and certain representations are better suited over others, depending on their usage and application.

2.3.3 Surface Elements

A *surface element* (surfel) can be thought of as small rendering primitives which capture localized information about a surface [13]. A surfel data structure can, for instance, store its radius, world-space position and normal, and the color of the surface it is placed on.

2.3.4 Axis-Aligned Bounding Boxes

An *axis-aligned bounding box* (AABB) is a type of bounding volume whose edges are parallel, and faces are perpendicular, to the coordinate axes of the space. An AABB in three-dimensional Euclidean space can be described by the pair of triples $(x_{\min}, y_{\min}, z_{\min})$ and

$(x_{max}, y_{max}, z_{max})$, where the first and second triple are the minimum and maximum bounds, respectively, of the AABB.

The benefit of using AABBs is that intersection tests against them are computationally inexpensive [1]. They are therefore typically assigned to dynamic objects in the scene to serve as coarse approximations of the size for the objects they enclose, and are then used to, for example, test for collision against other objects or the scene geometry itself.

2.4 Spherical Lattices

A point on a sphere in spherical coordinates is described by the triple (r, θ, ϕ), where $r \in [0, \infty)$ is the radius of the sphere, and where $\theta \in [0, \pi]$ and $\phi \in [0, 2\pi)$ are the latitudinal and longitudinal angles, respectively. Transforming a point from spherical to Cartesian coordinates can be done by using equation (2.2).

$$(x, y, z) = (r \sin(\theta) \cos(\phi), r \sin(\theta) \sin(\phi), r \cos(\theta)) \tag{2.2}$$

2.4.1 The Latitude-Longitude Lattice

The points on a latitude-longitude lattice are the intersections on the grid spanned by a number of parallels and meridians such that they are consecutively separated by an equal latitudinal and longitudinal angle, respectively [4]. An example of a latitude-longitude lattice consisting of 200 points, constructed from 10 parallels and 20 meridians, is shown in figure 2.2. Note the grid-like structure and the concentration of points near the poles, as well as the sparsity of points closer to the equator.

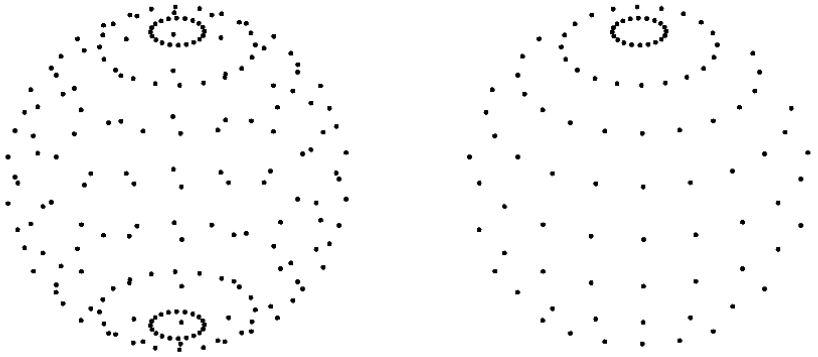


Figure 2.2: A latitude-longitude lattice with 200 points, spanned by 10 parallels and 20 meridians. The left image shows all points and the right image only shows the points which are front-facing.

2.4.2 The Fibonacci Lattice

The points on a Fibonacci lattice are placed on a spiral such that the longitudinal angle between consecutive points is the golden angle [4]. The spiral is wound onto the surface of the unit sphere, from pole to pole, by letting the distance along the polar axis between

consecutive points vary in steps that are inversely proportional to the total number of points in the lattice [4, 14].

Equation (2.3) summarizes the above description of a Fibonacci lattice with $n \in \mathbb{N}^+$ points where g denotes the golden angle. A point on the Fibonacci lattice on the unit sphere in Cartesian coordinates is given by equation (2.4), where the trigonometric relation $\sin(\arccos(z)) = \sqrt{1-z^2} \iff z \in [-1, 1]$ has been used to substitute for $\sin(\theta)$, where $\theta = \arccos(z) \iff r = 1$.

$$(z_i, \phi_i) = \left(1 - \frac{2}{n}\left(i - \frac{1}{2}\right), g\left(i - \frac{1}{2}\right)\right), \ \forall i \in [1, n] \tag{2.3}$$

$$(x_i, y_i, z_i) = \left(\sqrt{1-z_i^2}\cos(\phi_i), \sqrt{1-z_i^2}\sin(\phi_i), z_i\right) \tag{2.4}$$

An example of a Fibonacci lattice constructed using equations (2.3) and (2.4) consisting of 200 points is shown in figure 2.3. Note the subtle spiral-like structure and more evenly distributed points as compared to the latitude-longitude lattice shown in figure 2.2.

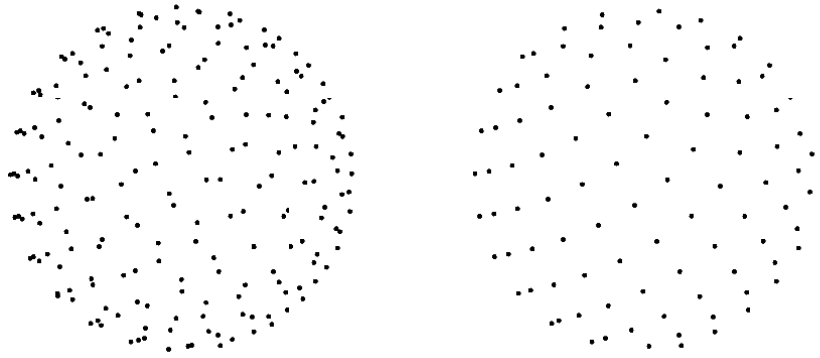


Figure 2.3: A Fibonacci lattice with 200 points viewed from the same angle as in figure 2.2. The left image shows all points and the right image only shows the points which are front-facing.

2.5 Shader Programs

A *shader program* (shader) is a programmable computer program that is executed on the graphics processing unit (GPU), as opposed to the central processing unit (CPU), as a stage in a pipeline [1]. There are many types of shaders, each belonging to a specific pipeline and with their own specialized purpose. For understanding this thesis, however, only the two following shader types are of interest.

2.5.1 Closest Hit Shaders

In the context of ray tracing and the ray tracing pipeline, after a ray has been dispatched into the scene and finished its traversal, a *closest hit shader* may be invoked for the ray. All the geometry that was intersected by the ray is enumerated, and the closest hit shader is executed for the intersection that was nearest to the ray origin [11].

The closest hit shader is single-threaded and typically used to evaluate the color of the *hit point* by, for example, testing if it is in shadow, checking the color of the hit surface or contributing light from global illumination.

We adopted the terminology used by the original author to call the ray that was dispatched as the *primary ray*.

2.5.2 Compute Shaders

It is possible to perform general-purpose calculations on the GPU that are not strictly related to graphics. For this, there is a special and simpler pipeline, designed to be run in parallel with the ray tracing pipeline by utilizing the large number of parallel processors on the GPU [10]. The programs that are executed by this pipeline are called *compute shaders*.

2.6 Previous System

The system developed in this thesis builds upon and extends an existing system by a former master's student [2] for calculating DDGI. The following subsections give brief overviews of the design and key aspects of that system as they pertain to this thesis. For a more comprehensive description of the previous system, the reader is encouraged to take a look at the aforementioned thesis.

2.6.1 Probes

The probes are naively and manually placed in a high-density, regular axis-aligned grid such that neighboring probes along any axis are equidistant from one another, and so that the entire scene is covered by the grid. Each probe is in turn represented by an octahedron which discretizes a sphere into eight distinct outgoing light directions from the probe, parallel to the facet normals of the octahedron.

All probes are stored in a buffer, and the position in the scene for a specific probe is determined by its offset into the probe buffer. This consequently results in there always being a fixed number of probes in the scene where they all have static positions for the duration of the application.

2.6.2 Surfels

All probes have the same fixed number of surfels, and each surfel belongs to exactly one probe and light direction for that particular probe. The surfels are stored in a buffer and, in much the same way as for the probes, the probe and light direction that they belong to is determined by their offset into the surfel buffer. This, again, means that there is always a fixed number of surfels in the scene. The surfels are also placed around the probe which they belong to using a latitude-longitude lattice.

2.7 Related Work

Majercik et al. [8] describe, among other things, that the use of probes and voxelized representations of the scene can be used for real-time global illumination and present trade-offs in using them. They also coin the term DDGI and present a method in which irradiance probes are extended to full irradiance fields in order to compute global illumination in scenes with dynamic objects and lighting.

The work of Dellson [2] presents a method for using light probes in combination with surfels to calculate DDGI in real-time through the use of a dense grid of probes. The work was inspired by a precomputed radiance transfer (PRT) system which utilizes light probes in the game *Tom Clancy's The Division* (2016) [15] and global illumination based on surfels (GIBS) which utilized screen-space surfels [6].

Zhang [16] presents a rendering implementation that uses static and dynamic surfels to improve real-time rendering efficiency and is an extension of the GIBS system. In order to accelerate the querying of surfels an octree grid was used.

Chapter 3
Implementation

This chapter describes, in considerable detail, how the new DDGI system works. Since the main focus of this thesis is to improve the probe and surfel placements of an existing system, we focus our attention on describing how new components were implemented, namely the octree as a hierarchical data structure, and how parts of the old system were reused, restructured and adapted to fit the dynamic nature of the new system.

The major components of the implemented system are namely the buffer resources, which store all data used by the system, and the shader programs, which access, manipulate and transform the data in the buffers.

3.1 Buffer Resources

In addition to the probe and surfel buffers used by the previous system that were mentioned in section 2.6, the new system uses three more buffers. The most notable of the new buffers is the octree buffer, which stores the scene octree and describes its shape, the details of which are explained in section 3.1.1. The two other newly introduced buffers are used as acceleration structures for quickly finding and referencing elements in other buffers and are explained in sections 3.1.4 and 3.1.5.

The function and purpose of the buffer that stores the AABBs is unchanged from the previous system and is used similarly in the new system, namely to detect changes to the geometry in the scene. The elements of the previous probe and surfel buffers are, however, restructured to accommodate for dynamically allocating and placing them in the scene. More regarding the new probe and surfel buffers is described in sections 3.1.2 and 3.1.3, respectively. Figure 3.1 shows a summary of the mentioned buffers, where the arrows are meant to illustrate how an element in one buffer can refer to an element in a different or the same buffer.

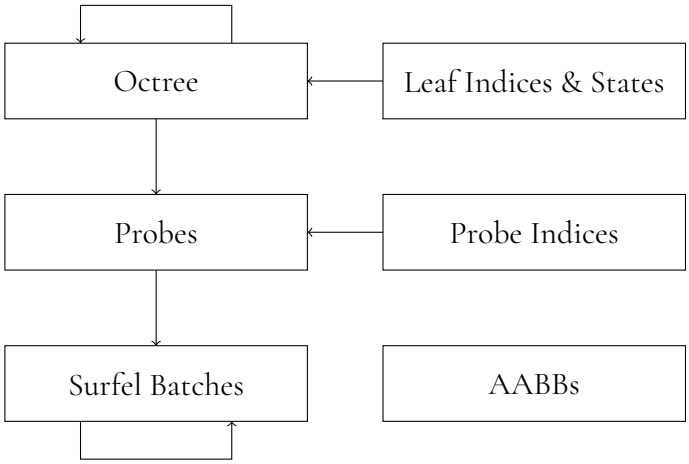


Figure 3.1: Key buffers maintained by the system and accessed by the different shaders. The arrows indicate how elements in one buffer can refer to elements in a different or the same buffer.

3.1.1 Octree

The octree buffer stores the nodes in the octree as signed integers and also describes the shape of the octree from the data stored in the buffer. The nodes are stored in *node clusters*, which are logical groupings of eight sibling nodes. Each node cluster also contains an additional element at the front of the cluster, used in part for referring to the parent node of the siblings, since they all have the same parent. A node cluster therefore occupies nine elements in the octree buffer.

The root node is treated specially since it does not have a parent nor any siblings, and consequently cannot be grouped into a node cluster. As for all other nodes, the root node only occupies one element in the octree buffer which we have defined to always be the element at the front of the buffer, at the zeroth index. Consecutive groups of nine elements after the root node are each node clusters.

Storing the root node, which occupies one ninth the size of a node cluster, in the same buffer where the node clusters are the logical units, makes the buffer inhomogeneous. This is the reason that the elements of the octree buffer are signed integers instead of node cluster data structures. Consequently, this introduces an undesirable side effect when the number of elements in the octree buffer is not $1 + 9n$ where $n \in \mathbb{N}$. In these cases, the elements at the back of the buffer will form an *incomplete* node cluster, of which there can be at most one. All other node clusters, if any, are *complete* node clusters. Caution is therefore exercised when building the octree to never have a node be the parent of the incomplete node cluster, if one exists, since this would result in an invalid octree representation.

Taking the above precaution into consideration, this implementation for representing and storing an octree in a buffer in the described way, allows for easy scalability and also guarantees that an octree buffer of any positive element count can be used to represent a valid octree, since the root node will always exist.

The following sets of indices are now introduced to aid in organizing and describing the

elements in the octree buffer:

$$\begin{aligned} R &= \{0\} \\ M &= \{1, 10, 19, \ldots\} \\ N &= \{[2, 9], [11, 18], [20, 27], \ldots\} \end{aligned}$$

The member of the singleton set R is the *root index* and is always zero as defined above. The members of the set M are *meta indices* which denote the starting index for the node clusters. Finally, the members of the set N are the *node indices* which denote the offsets of all nodes that are grouped into node clusters. All the nodes in the octree buffer are hence found at the indices in the set $R \cup N$.

The values stored at the indices from the set M have the following interpretations depending on the sign:

- Zero: The node cluster at this index is free.
- Negative: The node cluster at this index is allocated and the bitwise complement of the value stored at this index is the index of the parent node.

Similarly, the values stored at the indices from the set $R \cup N$ have the following interpretations depending on the sign:

- Zero: The node at this index is a leaf node.
- Negative: The node at this index is a leaf node and the bitwise complement of the value stored at this index is the index of the probe in the probe buffer which occupies this node.
- Positive: The node at this index is a parent node and the value stored at this index is the index of the node cluster containing the children.

When a node requests for a node cluster to be allocated for it, the octree buffer is linearly scanned at the indices in the set M until a free node cluster is found, disregarding the incomplete node cluster at the back if it exists. The found free node cluster is then linked to the parent using a compare-and-swap (CAS) atomic operation. If the CAS succeeded, the value stored at the parent is set to the index of the node cluster. Otherwise, the octree buffer is scanned anew.

Figure 3.2 depicts an example octree buffer with the root node at the top left at index zero, two complete node clusters and one incomplete node cluster. The root is the parent of the node cluster in the middle, wherein the third node is the parent of the top node cluster and the seventh node is a leaf with a probe in it.

3.1.2 Probes

The probes in this thesis are represented in the same way as in the previous system, that is, as octahedrons, and the notion of surfel ownership is also retained as was described in sections 2.6.1 and 2.6.2. To accommodate for the new system needs, the data structure definition for a probe is extended and shown in listing 3.1.

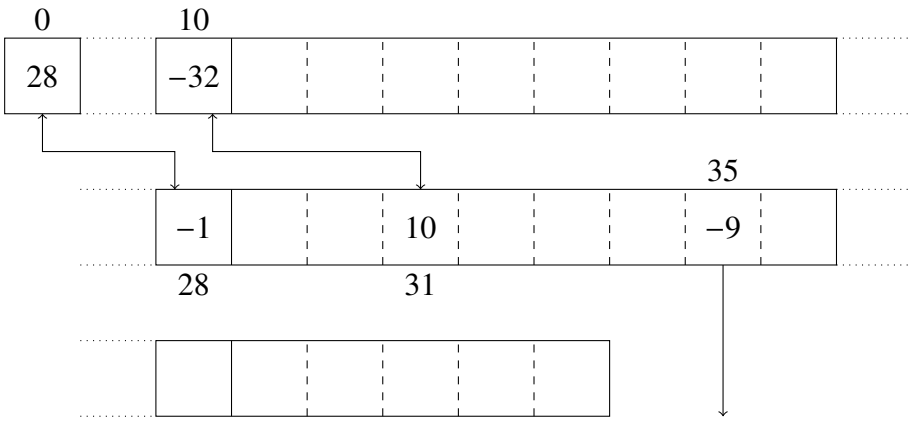


Figure 3.2: An example of an octree buffer showing the root node at the top left, two complete node clusters and one incomplete node cluster. The double-headed arrows are intra-buffer links and indicate parent-children relationships. The rightmost arrow is an inter-buffer link that points into the probe buffer, not shown in the figure.

In addition to the array of colors for the eight outgoing light directions, the structure now also has a signed integer whose interpretation is described below. The position of the probe in the world and depth in the octree is also stored in the probe data structure. The reasoning for this is that it avoids traversing the octree to calculate these values, which is a slow operation.

Listing 3.1: Data structure definition for probes with members for the multipurpose signed integer, array of outgoing light colors, world-space position, and depth in the octree.

```
struct probe
{
    int    link;
    float4 colors[COLORS_PER_PROBE];
    float3 position;
    int    depth;
};
```

The signed integer member of the probe data structure has the following interpretations depending on its sign:

- Zero: This probe is a free element in the probe buffer.
- Negative: This probe is allocated and the value of the bitwise complement of this member is the index of the first surfel batch in the surfel batch buffer in a chain of surfel batches owned by this probe.

Probes are allocated similarly to node clusters. Namely, the probe buffer is linearly scanned until a free probe is found and then allocated using a CAS atomic operation on the multipurpose integer member.

3.1.3 Surfels and Surfel Batches

The surfel data structure, shown in listing 3.2, has members for the position of the surfel in the world, as well as for the normal and diffuse color of the surface it is placed on, which is the same as in the previous system. In addition, the data structure now also has a member for the lit color of the surfel, which allows for the new system to separate the lighting of the surfels into multiple threads before accumulating the light into the probe to which the surfel belongs. This is further described in section 3.2.2.

Listing 3.2: Data structure definition for surfels with members for the world-space position and normal, and surface diffuse and lit colors.

```
struct surfel
{
    float3 position;
    float3 normal;
    float4 color_diffuse;
    float4 color_lit;
};
```

The surfel buffer has been restructured to no longer store surfels individually but rather in *surfel batches*, which are chunks of a fixed number of surfels. Listing 3.3 shows the data structure definition for a surfel batch where the surfels are stored in the array member. The interpretation of the signed integer member is described below.

Listing 3.3: Data structure definition for surfel batches with members for the multipurpose signed integer and array of surfels.

```
struct surfel_batch
{
    int    link;
    surfel surfels[SURFELS_PER_BATCH];
};
```

The signed integer member of the surfel batch data structure has the following interpretations depending on its sign:

- Zero: This surfel batch is a free element in the surfel batch buffer.
- Negative: This surfel batch is allocated and the value of the bitwise complement of this member is the index of the next surfel batch in a chain of surfel batches.
- Positive: This surfel batch is allocated and is the last surfel batch in a chain of surfel batches.

Multiple surfel batches can thus be chained together, allowing for the probes to have a different number of surfels as opposed to always having the same number of surfels, as was described in section 2.6.2.

Surfel batches are allocated in the same way as for the probes, namely by linearly scanning the surfel batch buffer and performing a CAS atomic operation on the multipurpose integer member for the found free surfel batch.

3.1.4 Leaf Indices and States

All operations on the octree occur at the leaf nodes. In particular, node cluster allocations and deallocations make the branches in the octree grow or shrink at the affected leaf nodes, and the probe lighting calculations are also performed in the leaf nodes, since that is where the probes reside.

The leaf buffer is a convenient supplementary to the octree buffer which records both the indices and states of all leaf nodes, and is used to operate on, and maintain the integrity of the octree. Both the memory and light update shaders, described in sections 3.2.1 and 3.2.2, access the leaf buffer each frame, and are split so that they only consume, or process, elements in the buffer which are in a "memory" or "light" state, respectively. Both shaders can, however, produce elements of both states.

To facilitate this, the leaf buffer consists of three segments of equal size which are treated like a circular buffer. The buffer has associated with it a *pull origin*, p, assigned each frame according to $p \leftarrow p + 2 \mod 3$, and the update shaders have associated with them a *pull offset*, q, which remains constant. The memory update shader has a pull offset of zero, and the light update shader has a pull offset of one.

The two shaders pull elements from the segment at index $p + q \mod 3$, and push elements of the opposite and same type into the segment at index $p+q+1 \mod 3$ and $p+q+2 \mod 3$, respectively. Figure 3.3 shows how the two update shaders access the segments of the leaf buffer over three frames, making one complete cycle of the buffer.

In addition, each segment is treated as a set of key-value pairs, where the keys are the leaf node indices, and where the values are the leaf node states. By treating each individual segment as a map like this, we can ensure that work is not duplicated for any leaf node under a frame, and that the octree integrity is maintained.

3.1.5 Probe Indices

The probe index buffer is supplementary to the probe buffer, and is used by the closest hit shader to accelerate the lookup of all probes in the scene, as is further explained in section 3.2.4. This is done by storing the indices of all allocated probes at the front of the probe index buffer followed by one or more invalid, negative, indices. As such, the number of elements for this buffer is one more than that of the probe buffer, to cover the extreme case where all the probes have been allocated. The probe index buffer is populated by a relatively small and simple compute shader program, of which its functional details are described in section 3.2.3.

3.2 Shader Programs

There are four shader programs used by the new system which are of interest, where three of them are compute shaders and the fourth is a closest hit shader. The shaders are run in parallel, as mentioned in section 2.5, and each shader accesses one or more buffer resources when it is executing. To prevent the shaders from reading incomplete data or corrupting data, the accesses to the buffers are synchronized by the use of resource barriers between each shader invocation.

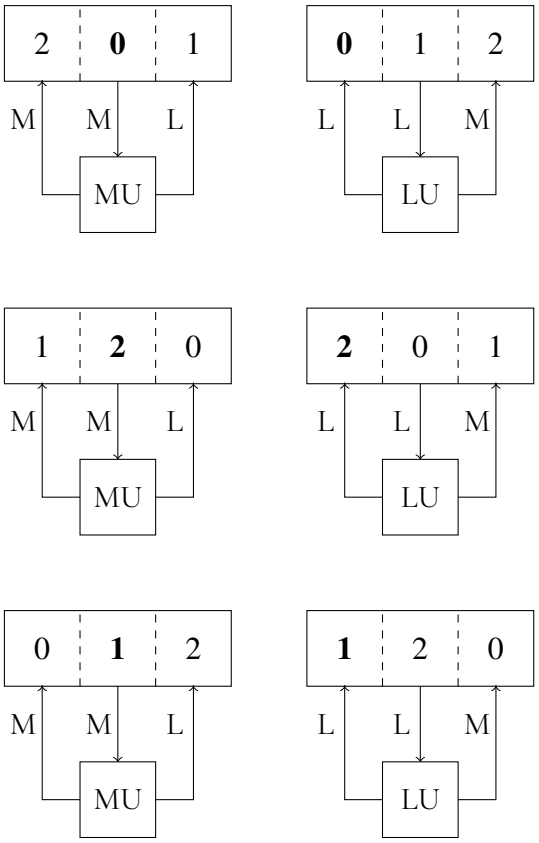


Figure 3.3: Accesses to the leaf buffer for both update shaders over three frames, making one complete cycle. The pull origin index is marked in boldface, and the notations "MU" and "LU" denote the memory and light update shader, respectively. The notations "M" and "L" denote elements which are in a state that are processed by the memory and light update shader, respectively. The arrows indicate elements being pulled from, or pushed into, segments of the leaf buffer by the two update shaders.

The use of barriers in this way imposes an ordering on the memory accesses and ensures that all reads and writes made by a shader for a particular buffer resource are completed before the next shader reads or writes to the same buffer. Figure 3.4 shows the four shaders of interest where the arrows indicate both the conceptual execution order and the logical flow of data between the shaders as a result of using resource barriers.

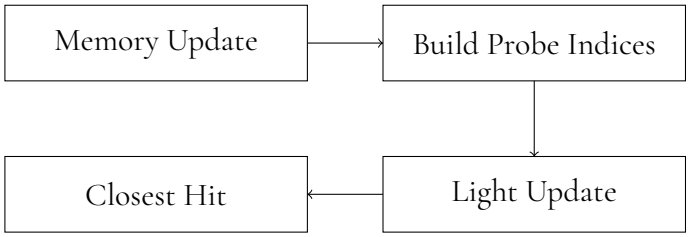


Figure 3.4: Shader invocation order for each frame. The arrows indicate the conceptual flow of execution.

3.2.1 Memory Update

Each thread in the memory update compute shader processes at most one node each frame having one of the states listed below.

M0 Child is allocating: A node in this state is a parent which has at least one child that is attempting to allocate a node cluster. As such, no further processing is done of this node, and it is not pushed back into the leaf buffer. This state overrides all other states processed by this shader, and prevents a parent from deallocating a child which is simultaneously allocating.

M1 Allocate probe and surfel batches: A node in this state is attempting to allocate a probe and associated surfel batches. If the allocations succeeded the node is pushed with a new state of L1. Otherwise, it remains without a probe and is pushed with a state of L3 to test for intersections again.

M2 Free probe and surfel batches: A node in this state frees its probe and associated surfel batches and, if it is not the root, pushes its parent with a state of M4 to prompt further deallocation.

M3 Allocate node cluster, probes and surfel batches: A node in this state is attempting to become a parent by allocating enough memory for a node cluster, and a probe and associated surfel batches for each of the eight children. If the allocations succeeded, the probe and associated surfel batches of the parent are freed, and the eight children are pushed with a state of L1. Otherwise, the node retains its probe and associated surfel batches and is pushed with a state of L2.

M4 Free node cluster: A node in this state is a parent which has been prompted by at least one of its children to deallocate its node cluster. The parent node simply checks if all children are without probes and, if so, proceeds with the deallocation of the node cluster. The children are then pushed with a state of L0, this node is pushed with a

state of L3, and, if this node is not the root, the parent of this node is pushed with a state of M4 to prompt further deallocation. No operation is performed if any of the children had a probe.

3.2.2 Light Update

Each thread group in the light update compute shader processes at most one node each frame having one of the states listed below, after first testing the node cube against all AABBs in the AABB buffer. If there was any intersection and the node does not have a probe, the node is pushed with state M1 and the shader exits. If instead the node has a probe, its state is transitioned to L1 so that the surfels can be replaced.

L0 Parent has freed: The parent of this node has freed the node cluster which the leaf belonged to. As such, no further processing is done of this node, and it is not pushed back into the leaf buffer. This state overrides all other states processed by this shader, and prevents a node from traversing the octree, since this is no longer a valid operation due there not existing a path to the root.

L1 Place surfels and count hits: The surfels of the probe in this leaf node are placed or replaced, and the hit fraction is counted to determine what state to push the node with after lighting the surfels and probe. The surfel placement work is distributed evenly among the threads in the thread group, so that all threads get to place approximately the same number of surfels. The surfels are placed around the probe using a Fibonacci lattice, calculated using equations (2.3) and (2.4), and so that each surfel is not placed outside the octant cube for the node. Once all surfels have been placed, the hit fraction, f, is calculated by a single thread and the state of the node is determined based on a lower and upper surfel hit fraction, f_{lower} and f_{upper}, and the current and maximum depths, d and d_{max}, in the following way:

- If $f < f_{\text{lower}}$ then the state is M2.
- If $f \geq f_{\text{upper}}$ and $d \neq d_{\text{max}}$ then the state is M3 and the parent node is pushed with state M0 if this node is not the root.
- Otherwise, the state is L2.

The node is not yet pushed into the leaf buffer, but rather proceeds with the work described under state L2 below.

L2 Light surfels and probe: A node in this state, or coming from state L1, lights all surfel hits belonging to the probe that resides in this node, before accumulating the light in the probe itself. The work is distributed among the threads in the thread group in the same way as for the work described under state L1. Once all surfel hits have been lit, a single thread accumulates the contributions of all lit surfels into the outgoing light directions of the probe and scales the contributions accordingly. If this state was reached from state L1, the node is pushed with the state determined in that state. Otherwise, the node is pushed with state L2.

L3 Just check AABBs: Leaf nodes with this state do not have a probe and thus do not perform any lighting calculations. The only work done by this shader in this case is to check against the AABBs and take proper action as described above.

3.2.3 Build Probe Indices

A simple, multithreaded, compute shader is used to populate the probe index buffer and is invoked after the memory update shader, when all probes for a frame have been placed. The shader works by first resetting all indices in the probe index buffer to an invalid, negative, index. It then scans all elements in the probe buffer and inserts the indices of all allocated probes into the probe index buffer, from the front toward the back.

3.2.4 Closest Hit

The closest hit shader is where the hit points in the scene for the dispatched rays are colorized, both with direct illumination if the hit point is not in shadow, and with global illumination using the light data stored in the probes.

For calculating the contributing global illumination for a hit point, the closest hit shader iterates over the elements in the probe index buffer until the first invalid value is reached. In each iteration, the current probe index is then used to directly reference a probe that exists in the scene in the probe buffer. The probe is tested for visibility by casting a ray from the hit point to the probe, using the stored probe position in the probe data structure.

If the probe is visible it is treated as a point light and contributes light to the hit point from its outgoing colors that are in the direction toward the hit point, and each contribution is scaled by the scalar product between the hit point surface normal and outgoing light direction. The light is also attenuated based on the distance between the hit point and probe and the depth in the octree of the probe, so that probes that are distant and cover a smaller volume contribute less intense light.

3.3 Technical Challenges

The implementation of the new DDGI system was not without its challenges. The previous system was able to use an efficient way to light probes and then interpolate between them during the primary ray hit. If a probe was unable to place a surfel, it could instead look up the color from a neighboring probe, giving the system a way to propagate light efficiently [2]. However, this was only possible because it used a *dense* probe grid, and since our system is *sparse* there is no guarantee that an immediate neighboring probe exists in a given direction.

This turned out to be a difficult problem to solve, and although some time was spent looking into possible solutions, we were unfortunately unable to find an efficient one. Instead, we settled for a solution where we would loop over all active probes in the scene and calculate their contribution to a hit point. This meant that during the primary ray hit we had to determine if it could see each probe, and in order to check for visibility we computed a ray trace from the hit point to the probe. This is a costly operation and does not scale well with the number of active probes.

Chapter 4
Evaluation

This chapter aims to evaluate our implementation through empirical visual quality, execution times, and how the probe counts and surfel hit fractions varied during the measurement tests. In section 4.1 we will describe the approach we took to evaluate our implementation, in particular, the scenes and configurations used. In sections 4.2 to 4.4 we dive into the different measurements gathered. Finally, in section 4.5 we will discuss the results. The specifications of the PC used can be seen in table 4.1.

Component	Specifications
GPU	NVIDIA GeForce GTX 4070 Ti
CPU	Intel Core i7-12700, 2100M MHz
RAM	16 GB
OS	Windows 11

Table 4.1: Specifications of PC used in the evaluations.

4.1 Evaluation Approach

The test method is very similar to that of the previous work [2], and any notes about reusing parts of this method or extending them will relate to it.

4.1.1 Configurations

We decided to limit our tests to these configurations:

- Maximum octree depths of 2 and 3.
- Number of surfels per probe of 64 and 128.

- Minimum and maximum surfel hit fractions of $\frac{1}{4}$ and $\frac{1}{2}$, respectively.

Our initial tests showed that lower depth values resulted in poor image quality and depths greater than three did not increase the image quality, but rather significantly increased the time spent per frame. As mentioned in section 2.4, the latitude-longitude lattice is constructed from a number of parallels and meridians. This meant that we could compare the two surfel placement methods by giving the latitude-longitude lattice the same number of parallels as meridians in the case of 64 surfels per probe, i.e., 8 each.

We also chose 128 surfels per probe to see how the latitude-longitude lattice would compare to the Fibonacci lattice in the case where the number of parallels and meridians differed. In this case we kept 8 parallels and extended the number of meridians to 16. Additionally, these two configurations were also chosen because they were used in the probe grid approach, and we wanted to see how our implementation would compare to it.

Finally, we do acknowledge that the range chosen for the octree to further subdivide is a parameter that could have been tested further, but due to time constraints we decided to limit the number of configurations to the ones listed above.

4.1.2 Scenes

In order to evaluate our octree implementation we decided to reuse the CornellBox scene from the previous project [2]. This scene consists of a simple room with four walls, a ceiling and a floor. Two of the walls are painted in different colors, the left wall being green and right one blue, and the room is lit by a single light source from the ceiling. In addition to this, there are two boxes, one in the back to the left and one in the front to the right. The boxes, the remaining walls, ceiling and floor are gray. The scene is shown in figure 4.1. Note that in order to see into the room, the wall that should be in front of the camera is removed with back face culling, but it is still present in the scene.

In addition to the original CornellBox scene, we added a skull model to give the scene a more geometrically complex object [3]. This would allow for some frame of reference between the original probe grid and our octree implementation for the DDGI system, as well as showing the effect of the system in a more complex scene.

4.1.3 Test Method

The test method ran the program for 600 frames with an animation that started at frame 200 and ended at frame 350. During this animation, the objects in the scene would rotate 45 degrees around their vertical axis. This allowed us to see how the system handled changes in the scene and how it behaved before and after. In our case, we were interested in how much time was spent per frame in the different shader passes, and how the different lattices choices would affect the results.

In order to take measurements regarding time spent per frame, the system calls the `EndQuery` function before and after dispatching the different shaders to gather timestamps. This is then followed by a call to `ResolveQueryData` in order to retrieve them out of the query heap [12]. After the frame is complete, the measurements are retrieved through a callback method and stored for export to a CSV file for further analysis.

To see the effect of the different surfel placement methods, we decided to measure how many surfels and probes were placed each frame. In the case of the surfel placements, we decided to measure it as a percentage of the total number surfels that the probe could place in the scene.

4.2 Visuals

Figures 4.1 to 4.6 show the output images of the previously mentioned scenes when the probe grid and octree are used. The images are rendered with 64 surfels per probe and using the latitude-longitude surfel placement strategy. When using the octree we have used a depth of 2. We did not perceive any significant visual differences between the surfel placement strategies during our tests, and in order to save space we have chosen to only show the images from depth 3 and sample size of 64 as our results showed the largest amount of differences in this case. The comparison between the two lattices can be seen in figures 4.7 and 4.8.

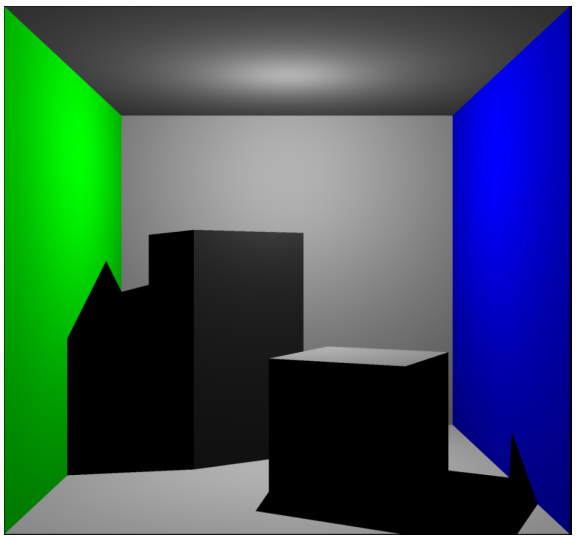
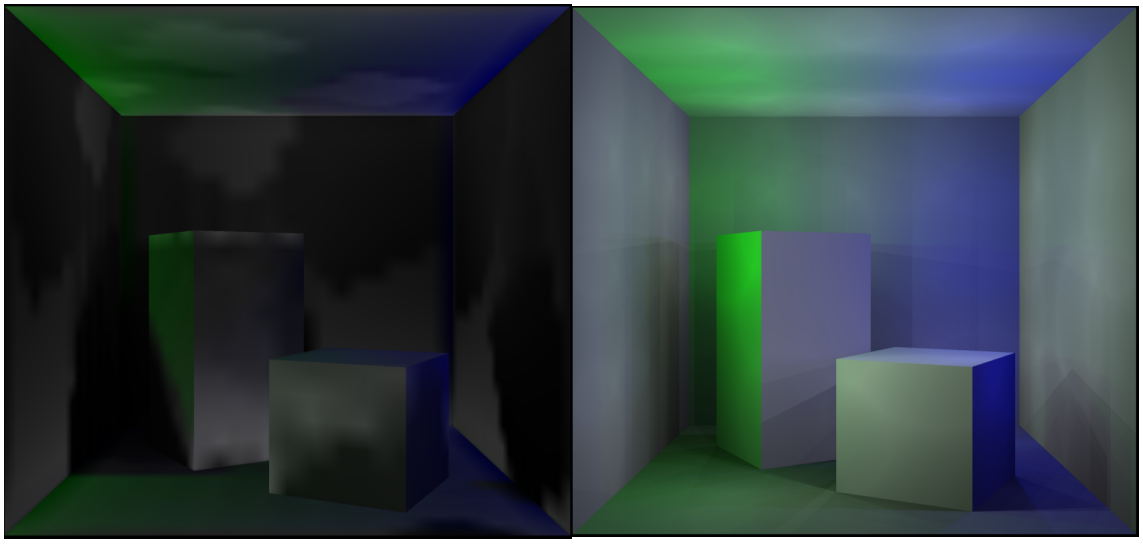
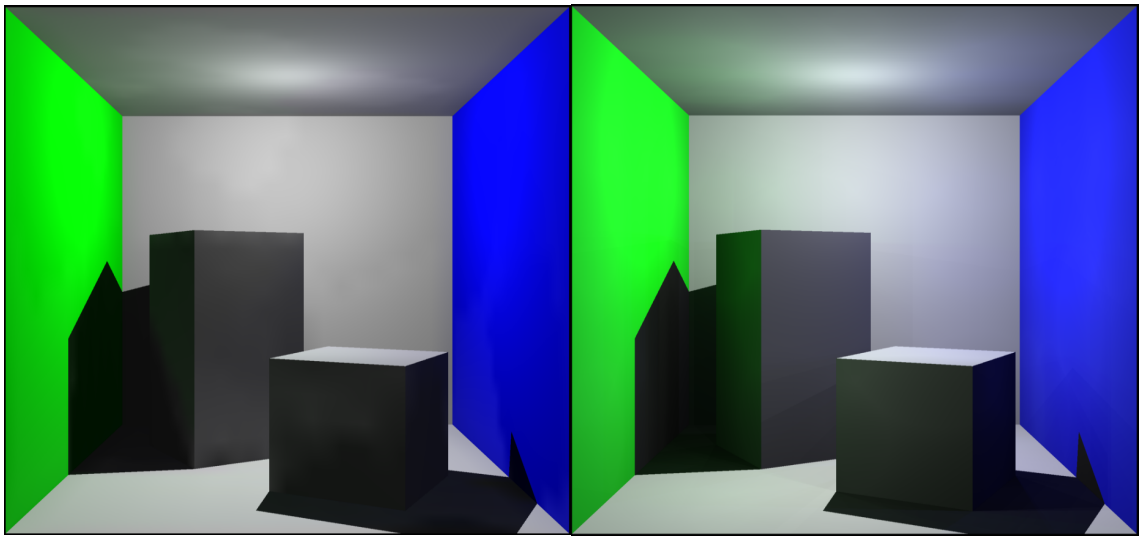


Figure 4.1: DDGI off in CornellBox.



(a) Probe Grid **(b)** Octree

Figure 4.2: DDGI only in CornellBox, brightened by 150%.



(a) Probe Grid **(b)** Octree

Figure 4.3: Final image of CornellBox.

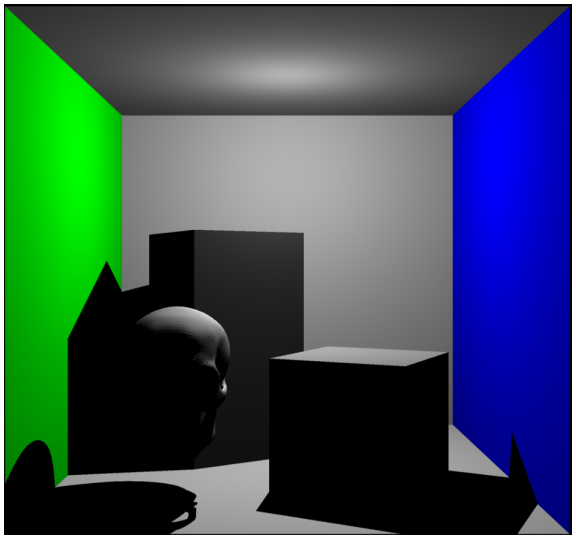
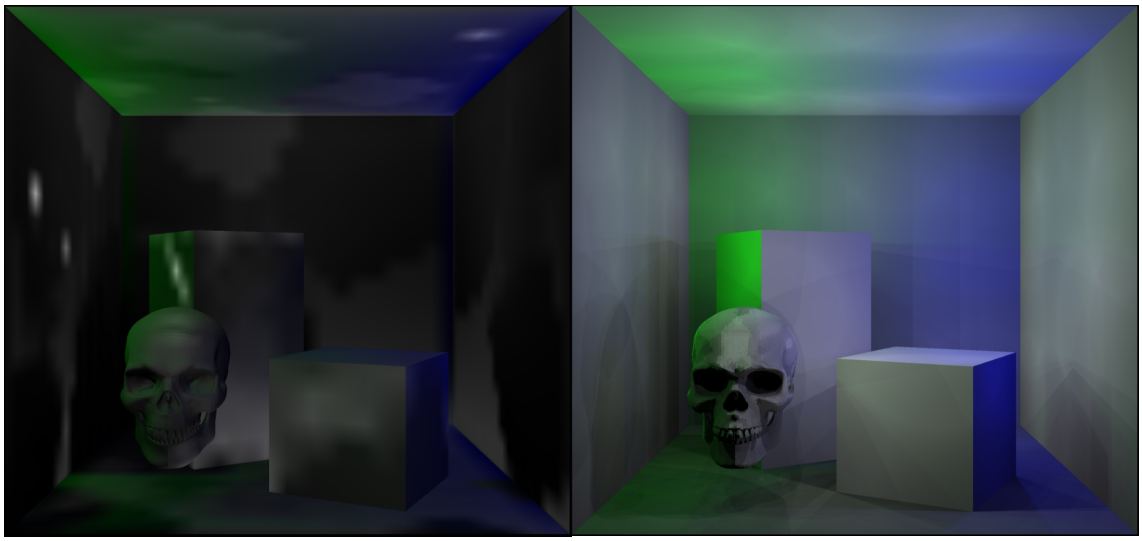
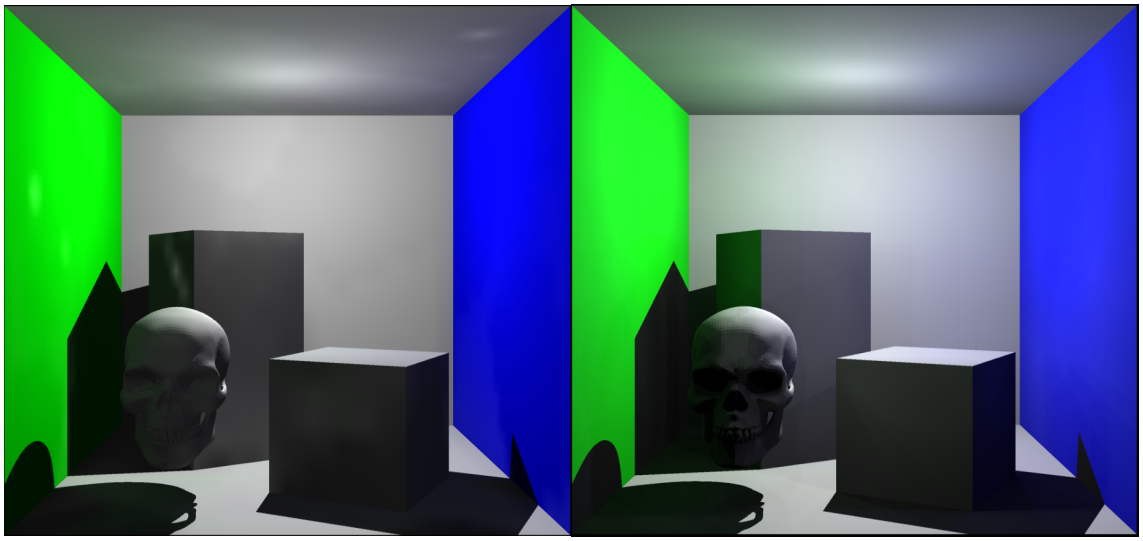


Figure 4.4: DDGI off in CornellBoxWithSkull.



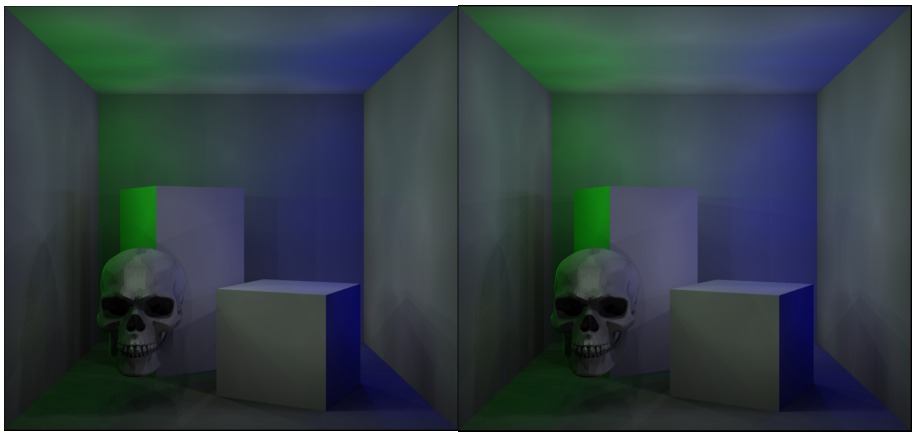
(a) Probe Grid **(b)** Octree

Figure 4.5: DDGI only in CornellBoxWithSkull, brightened by 150%.



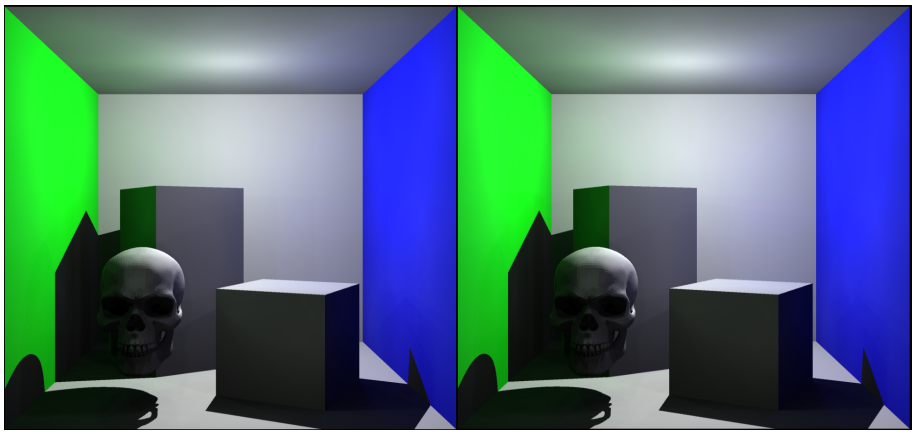
(a) Probe Grid **(b)** Octree

Figure 4.6: Final image of CornellBoxWithSkull.



(a) Lat-Long **(b)** Fibonacci

Figure 4.7: DDGI only in CornellBoxWithSkull for the different lattices, brightened by 150%.



(a) Lat-Long **(b)** Fibonacci

Figure 4.8: Final image of CornellBox with Skull for the different lattices.

4.3 Execution Times

Here follows a set of graphs showing the execution times as measured from the different scenes. In order to save space we only include measurements made with 64 surfels per probe and at octrees with maximum depths of 2 and 3. The graphs show the different compute shader execution times for the different configurations.

4.3.1 Cornell Box

In figures 4.9 to 4.14 we see the execution times for the CornellBox scene measured in milliseconds. In figure 4.9 we see a small bump after around 20 frames into the measurement. We believe that this is due to the system starting up, but the actual cause is unknown, and this behavior is shared across the different measurements. In figure 4.10 we see that outside of the animation the execution times are similar. Once the animation starts we get a jump in the primary ray shader and this behavior is shared for the following measurements. Therefore, we will show a zoomed in version of the animation for the remaining figures.

When we increased the depth to 3 we saw that the overall execution time increased from around 13 ms in the static case to 22 ms for the primary ray shader. During animation, we see a similar jump in the primary ray shader as we did at depth 2. At this depth there are also spikes in the memory update pass that are not present at depth 2.

Finally, in figures 4.13 and 4.14 we see the average execution times for the different implementations.

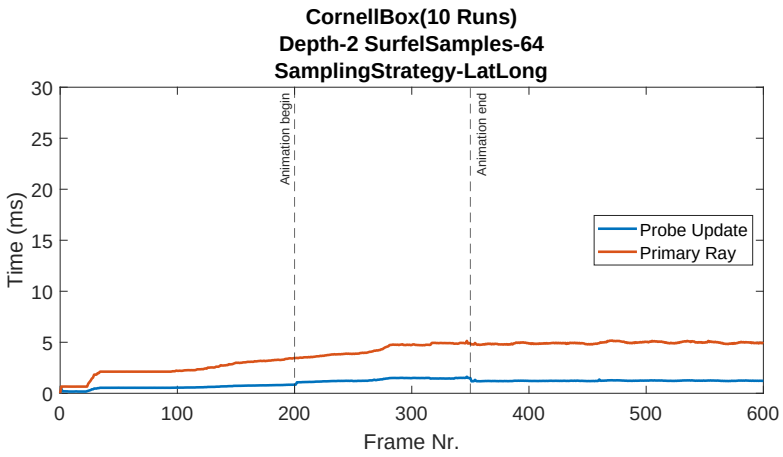


Figure 4.9: Frame time for probe grid using a latitude-longitude lattice.

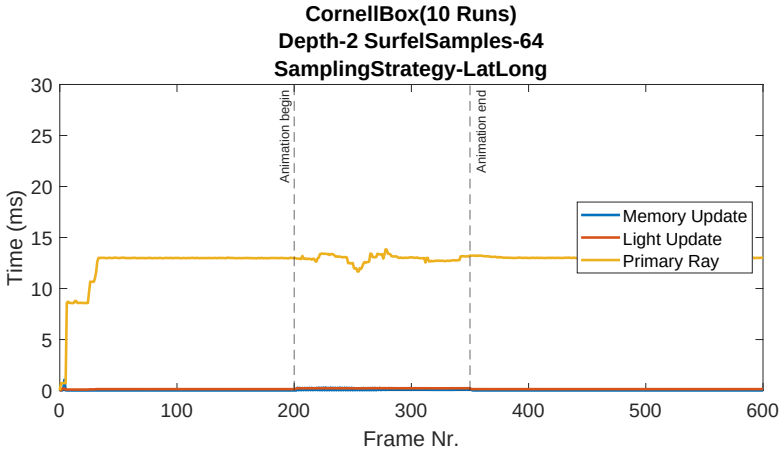


Figure 4.10: Frame time for octree depth 2 using a latitude-longitude lattice.

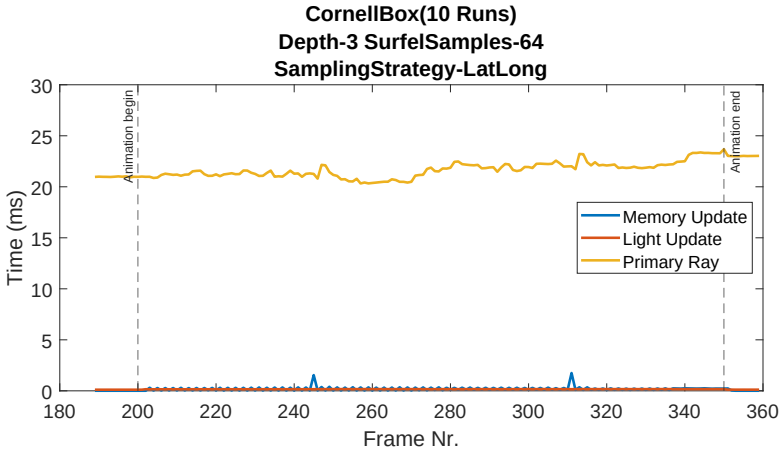


Figure 4.11: Frame time for octree depth 3 using a latitude-longitude lattice, zoomed in.

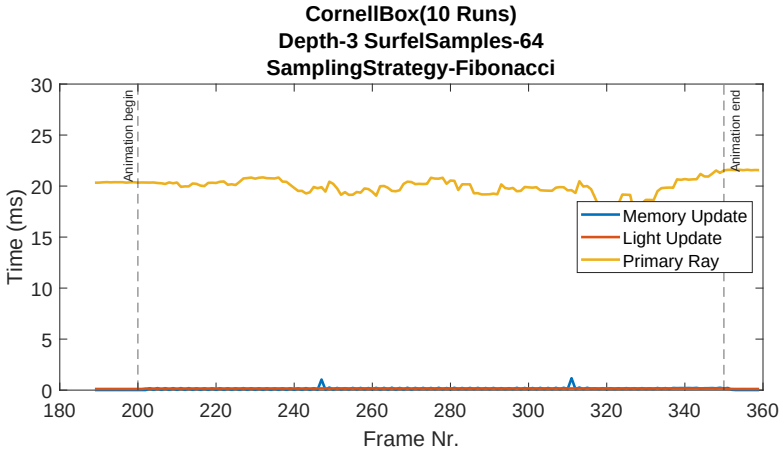


Figure 4.12: Frame time for octree depth 3 using a Fibonacci lattice, zoomed in.

Figure 4.13: Average execution times of probe grid in CornellBox.

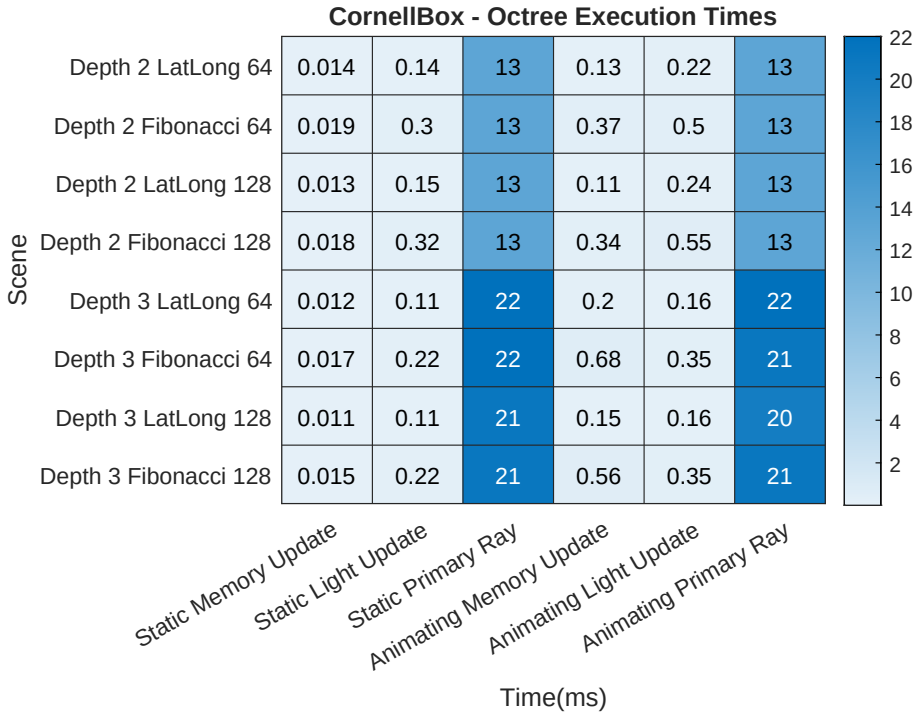


Figure 4.14: Average execution times of octree in CornellBox.

4.3.2 Cornell Box with Skull

Figures 4.15 to 4.20 show the execution times when a skull is added to the CornellBox scene. In figures 4.17 and 4.18 we see that they have spikes in the memory update pass that are not present at depth 2, similar to that of the scene without the skull.

Finally, in figures 4.19 and 4.20 we see the average execution times for the different implementations. From the measurements it is also clear that the performance was not impacted too heavily by the addition of the skull model.

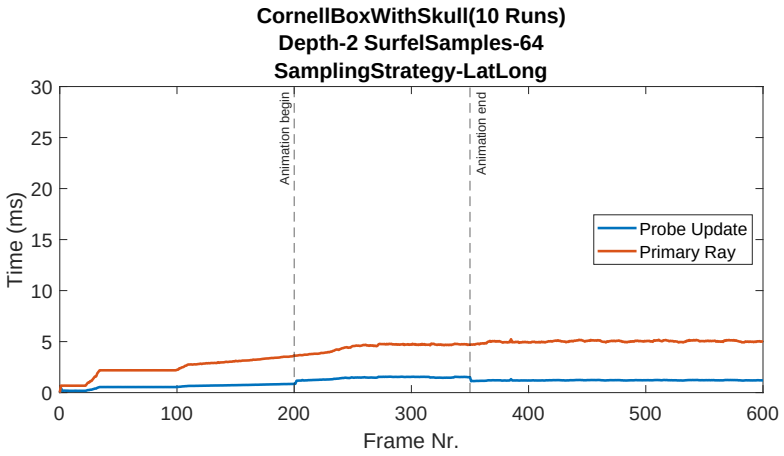


Figure 4.15: Frame time for probe grid using a latitude-longitude lattice.

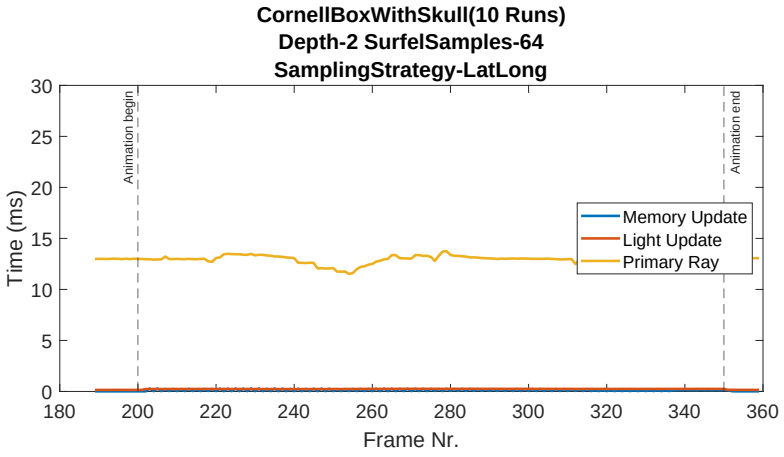


Figure 4.16: Frame time for octree depth 2 using a latitude-longitude lattice, zoomed in.

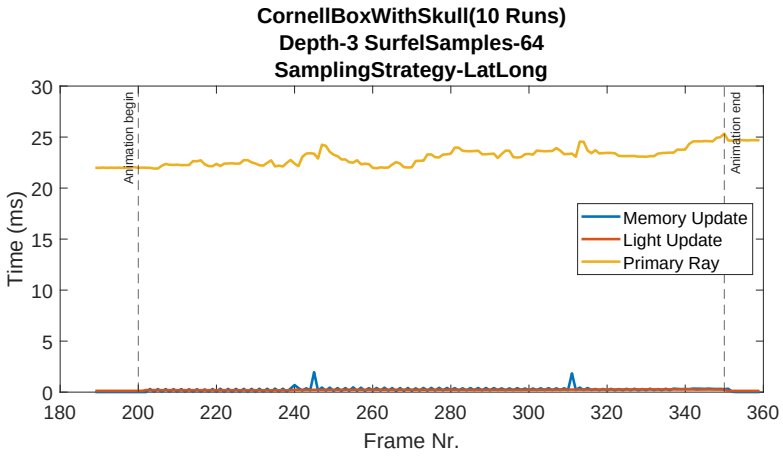


Figure 4.17: Frame time for octree depth 3 using latitude-longitude lattice, zoomed in.

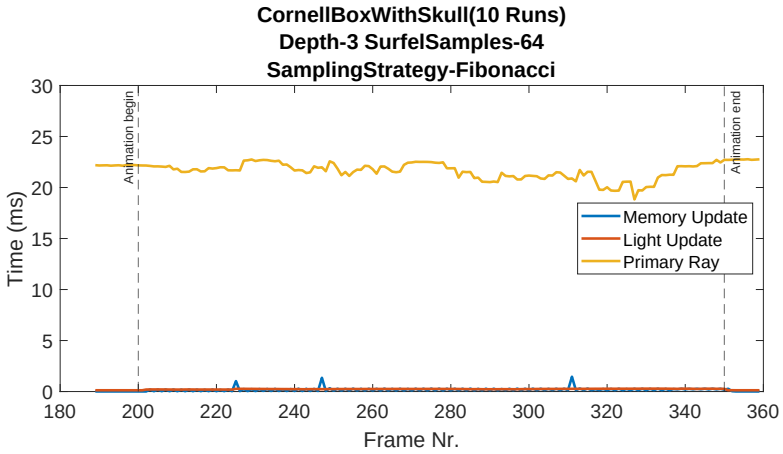


Figure 4.18: Frame time for octree depth 3 using a Fibonacci lattice, zoomed in.

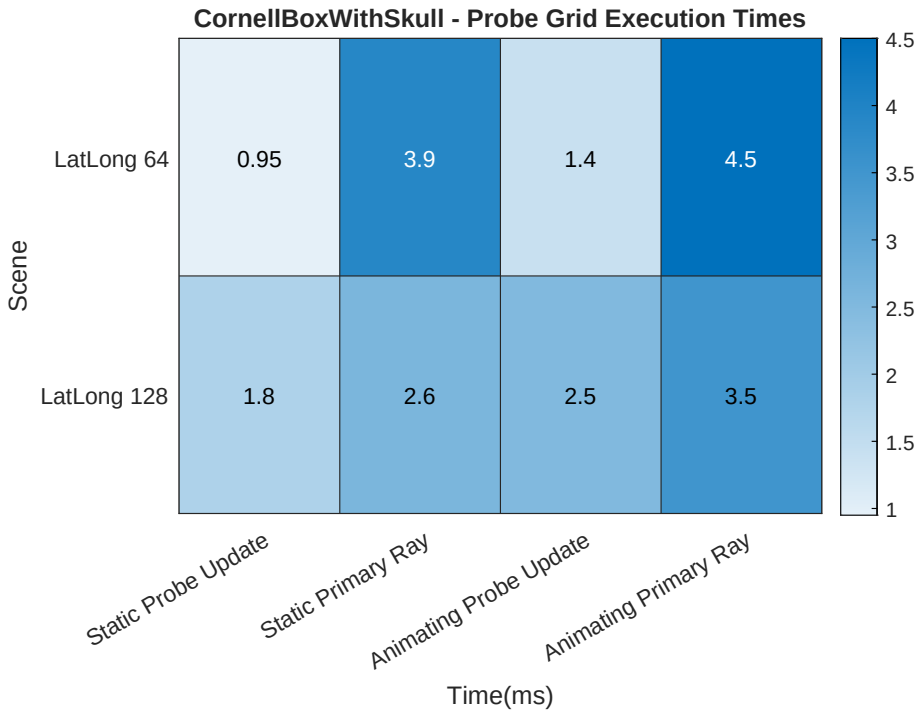


Figure 4.19: Average execution times of probe grid in CornellBoxWithSkull.

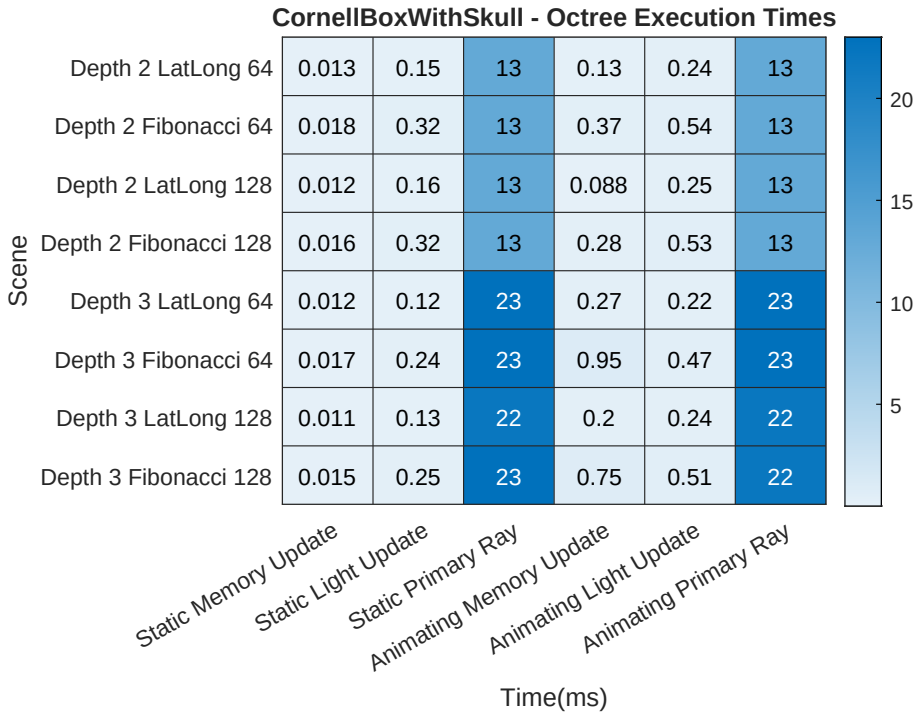


Figure 4.20: Average execution times of octree in CornellBoxWithSkull.

4.4 Probe Counts and Surfel Hits

In this section we look at how the number of probes and surfel hit fractions varied during the measurement tests and as we alternated between the latitude-longitude and Fibonacci lattices. The results from the different measurements are quite similar, and so to save space we will only show figures for the same cases as in section 4.3.2. Since the probe grid has a constant number of probes (9261) we will only show the results for the octree.

It should be noted that the y-axis of the different graphs are affected by certain aspects. For the probe count, the maximum number of probes that can be in the scene for the different depths is a power of 8, which in the worst case means that it devolves into a probe grid, i.e., fills every leaf with a probe. For our testing configurations this means a value of 64 and 512. In the case for the surfel hit fractions, the maximum number of surfels is based on the splitting condition range of $\frac{1}{4}$ and $\frac{1}{2}$, which is why the y-axis for those graphs are between those values. The results for the other scene and configurations can be found in appendix A.

4.4.1 Cornell Box with Skull

Since the number of probes were constant before and after animation, we only show the zoomed in parts of the animation. During animation, we can observe that the number of probes changes, as shown in figures 4.21 to 4.23. This is something we would expect, since the tree is iteratively built based on changes in the scene.

Surfel hits share this pattern, although, whenever the probe count increased, the surfel hit fractions decreased. This behavior is shown in figures 4.24 to 4.26. Finally, we have again put the averages in table form to more easily compare them, which can be seen in figures 4.27 and 4.28.

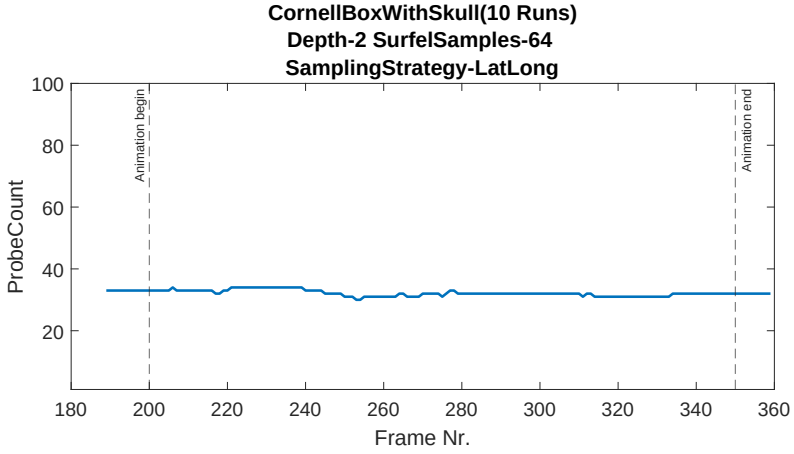


Figure 4.21: Probe count for octree depth 2 using a latitude-longitude lattice.

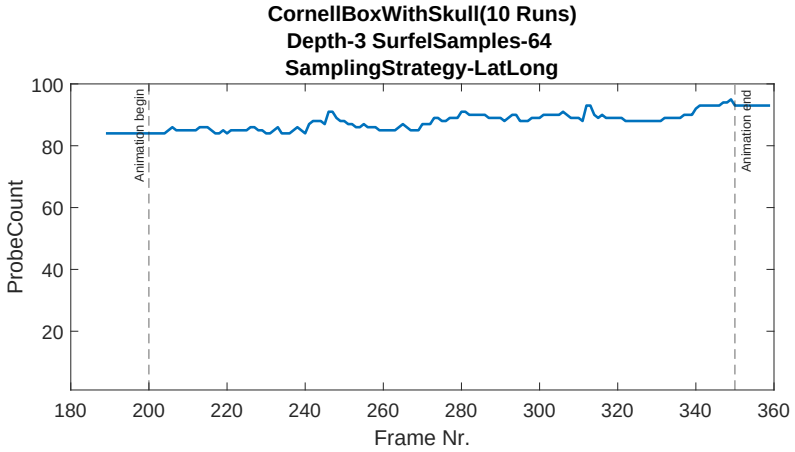


Figure 4.22: Probe count for octree depth 3 using a latitude-longitude lattice.

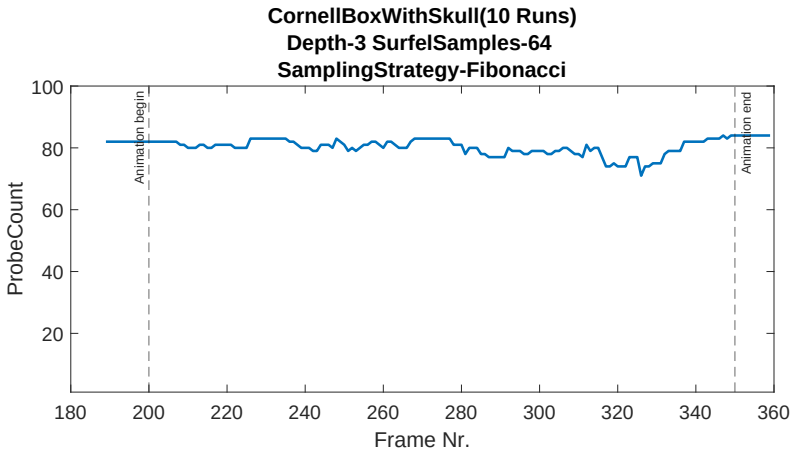


Figure 4.23: Probe count for octree depth 3 using a Fibonacci lattice.

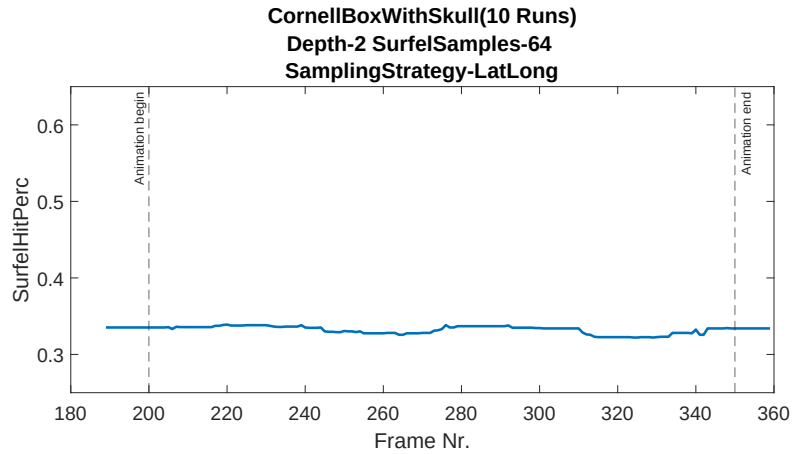


Figure 4.24: Surfel hit fraction for octree depth 2 using a latitude-longitude lattice.

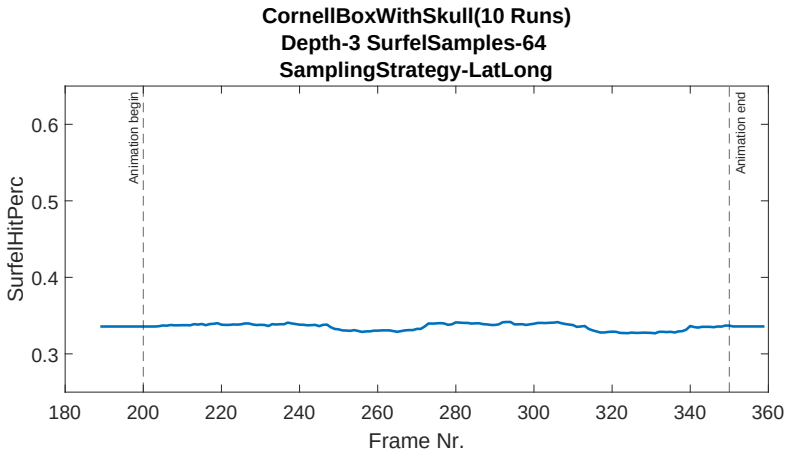


Figure 4.25: Surfel hit fraction for octree depth 3 using a latitude-longitude lattice.

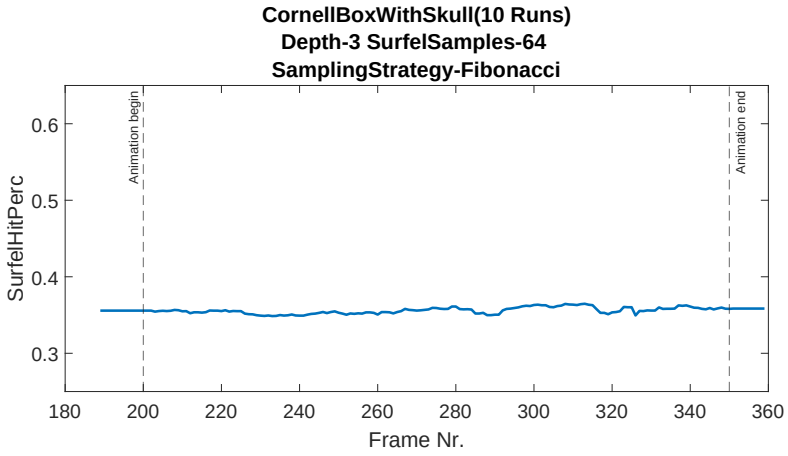


Figure 4.26: Surfel hit fraction for octree depth 3 using a Fibonacci lattice.

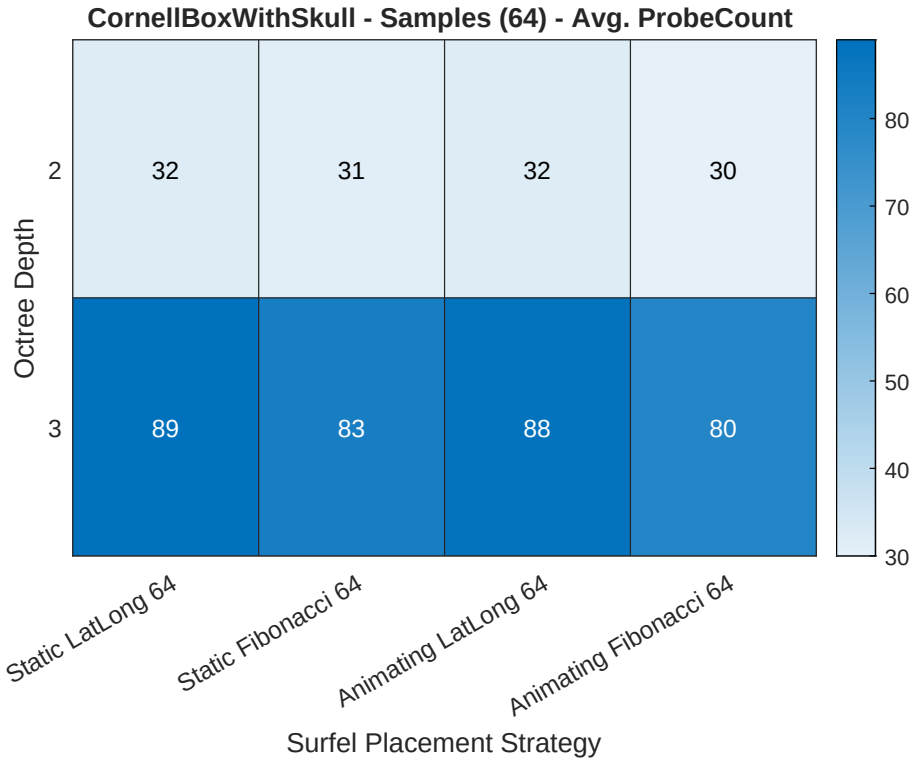


Figure 4.27: Probe count averages at depths 2 and 3.

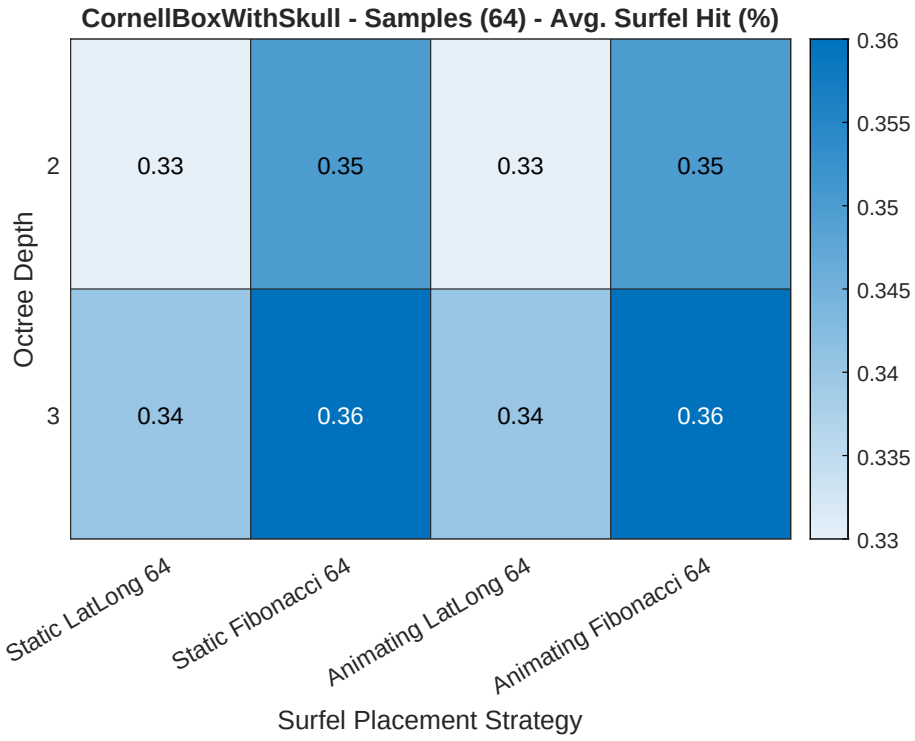


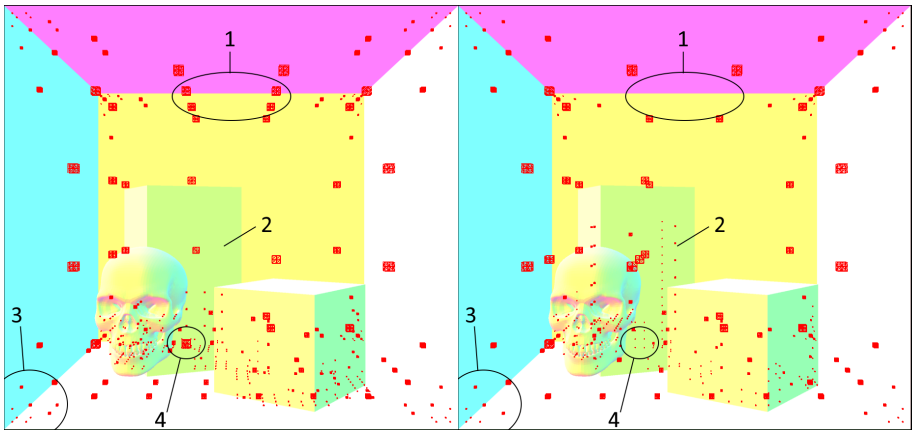
Figure 4.28: Surfel hit fraction averages at depths 2 and 3.

4.4.2 Visualization of Probes in the Scene

In addition to the numerical measurements of probe counts and surfel hit fractions, we could also visualize the probes in the scene through our debugging system, as shown in figure 4.29. The probes are rendered as small red boxes, scaled with their depth in the octree, and the rest of the scene is rendered using the hit normal to more easily see the probes.

In figure 4.29a we can see that the probes, when the latitude-longitude lattice is used, become more concentrated at the top and bottom of the scene, as indicated by the region marked as 1 and toward the bottom of the two blocks. We can also see in figure 4.29b that, when the Fibonacci lattice is used, probes are more easily placed along the vertical wall of the block indicated by 2, whereas the latitude-longitude has more difficulty in capturing this.

Both placement methods tend to creep into the corners of the scene, which can be seen in the region marked by 3. The region marked by 4 shows a region in between the three objects that splits when the Fibonacci lattice is used, but not when the latitude-longitude lattice is used.

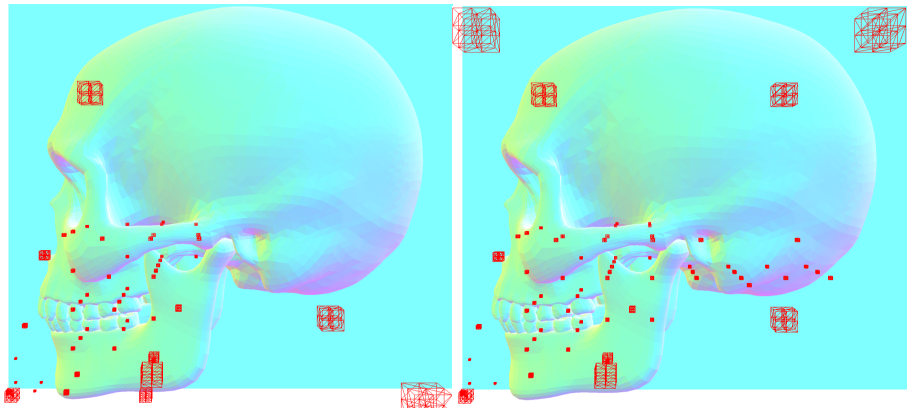


(a) Lat-Long **(b)** Fibonacci

Figure 4.29: Comparison of probe placements in the CornellBoxWithSkull scene with a maximum octree depth of 5 and 128 surfels per probe.

In figure 4.30 we see a more zoomed in part of the same scenes shown in figure 4.29. Here we can see that the Fibonacci lattice in figure 4.30b is able to place probes under the skull and around the top, while in figure 4.30a we can see that the latitude-longitude method is unable to do the same. This is most likely due to the chosen splitting range of $\frac{1}{4}$ and $\frac{1}{2}$ of the total number of surfels, and the fact that the latitude-longitude lattice was unable to split the node in the region marked as 4 in figure 4.29a.

With a more lenient splitting range we would expect the latitude-longitude lattice to be able to place probes under the skull due to its concentration in the poles, but at a cost of more probes being added to the scene. This further points to the need for more experiments with different splitting ranges.



(a) Lat-Long **(b)** Fibonacci

Figure 4.30: Comparison of probe placements in the CornellBoxWithSkull scene with a maximum octree depth of 5 and 128 surfels per probe, zoomed in on the skull.

4.5 Discussion

In this section we discuss the different measurements from the previous sections and the validity of our results.

4.5.1 Visual Quality

From figure 4.6 we can see the final image of the scene with the skull which looks reasonable. We can see that the DDGI system accurately demonstrates bounced light illuminating, for example, the right side of the skull and the taller box with green light, and this effect is more easily seen in figure 4.5. We can also see this effect in the top of the room where light bounces from the respective side walls.

The image is not perfect, however, as there are artifacts present, mainly the small circles that can be noticed along the walls and the dark regions that look like shadows. We suspect that these artifacts are caused by the way we sampled the probes as described in sections 3.2.4 and 3.3. To reduce the circle artifact we clamped the falloff used to scale a probe's contribution when the hit point is inside the volume covered by the probe. See figure 4.31 for an example of what happens when we do not clamp the falloff. Additionally, we further reduced the contribution by the volume spanned by the probe in order for the system to scale as our octree subdivides. The reason for this is that, after subdivision, the same volume is potentially covered by more probes. If no scale factor would be used then the luminance of the volume would increase as we continued to subdivide the octree.

Another artifact is the dark regions that look like shadows, which can be seen on the left side of the room in figure 4.5. We believe these issues arise due to the discretization of probe placement by the octree and our sampling technique. If a probe is placed in a region blocked by geometry relative to the primary ray hit, it will fail the visibility check. This results in a sharp falloff between hit points that can see probes and those that cannot.

From the visual comparisons of the scene with the skull in figure 4.6 we can see that the skull model is more accurately lit in the octree implementation, capturing the dark regions

of the models eyes and teeth giving it a more realistic feel. We can also see that the visual artifacts of the small white boxes which are most noticeable on the box behind the skull are gone.

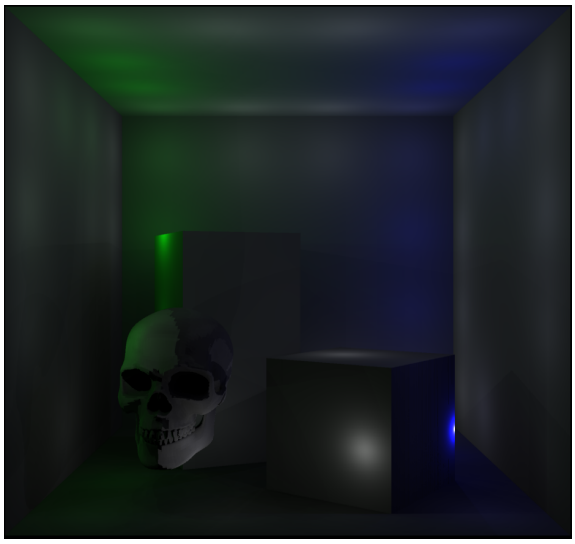


Figure 4.31: Figure that shows what happens if we do not clamp the falloff of the probe contributions.

4.5.2 Execution Times

As figures 4.9 to 4.20 show, the execution time of the primary ray hit is quite high which increases with the depth of the tree. This is most likely due to how we sample the probes mentioned in sections 3.2.4 and 3.3, since we iterate through all active probes for each hit and perform visibility checks for each. This is a costly operation and does not scale well with the number of active probes in the scene.

When animating we can see from figures 4.10 to 4.12 and figures 4.16 to 4.18 that this is exacerbated. This is most likely because the octree updates its regions where the splitting condition criterion has changed during animation, and new probes might be placed or old ones removed. We suspect that the spikes have to do with our linear search of the buffers from we allocate, where we currently iterate linearly through the buffers until a free element is found. This is not optimal, and an improvement would be to try only a certain number of times before rejecting.

If we focus our attention on the other shader passes, we can see from figures 4.13 and 4.14 and figures 4.19 and 4.20 that the total time for the probe grid is higher in all cases then the combined time for the memory and light update shader stages for the octree. We believe this is in part due to the fact that we do not handle missed surfels, leading to a reduced amount of time spent in the light update shader. This time could be reduced further by enabling the use of multiple threads for the different compute shader passes, allowing multiple surfel placements and lighting calculations to be done in parallel and then synchronized, instead of only utilizing one thread per thread group. We did however not enable this part of our implementation since we wanted to keep the measurements fair when comparing to the previous system.

4.5.3 Probe Counts and Surfel Hits

From the results shown in figures 4.27 and 4.28, we can see that the number of probes placed is significantly lower than for the probe grid case. This is expected since the octree is able to subdivide and only place probes where they are needed. It should be noted that for the different depth values of the octree the maximum number of probes placed is a power of 8, in which case it then devolves into a probe grid of a certain dimension.

From the results we see that in the case of depth 2 we have already reduced the probe count by more than half, and even more at depth 3. From preliminary tests we saw that the reduction was larger the deeper we allowed the tree to be and eventually stopped subdividing. However, this is most likely affected by the splitting range as well.

4.5.4 Latitude-Longitude and Fibonacci Lattices

When comparing the results in figure 4.27 we can see that the number of probes placed is slightly lower for the Fibonacci lattice. We can also see that the average surfel hit fractions are higher in figure 4.28. This happens regardless of the way we have constructed the latitude-longitude lattice.

Since our splitting condition is based on the amount of placed surfels we can visualize where each lattice is able to place more surfels. In figure 4.29 we see that the use of a latitude-longitude lattice results in the top and bottom of the scene getting more probes. This is something we would expect since the points on this type of lattice get concentrated toward the poles. However, this should be taken with a grain of salt, since this could also be partly affected the splitting condition range. It also shows that the Fibonacci lattice is able to perform slightly better than the latitude-longitude lattice in a more constrained environment.

4.5.5 Validity

As with most evaluations, there are some limitations to our results. The most notable is the number of configurations tested as well as the number of scenes used. We only ran the full tests on octrees with maximum depths of 2 and 3, and the number of surfel placements were limited to 64 and 128. We recognize that additional scenes and configurations, especially for the splitting condition, could have given us a better understanding of how the system behaves.

Furthermore, we can also see from the measurements that in some cases the system execution time looks like it is increasing. This was a concern of ours that the system might be unstable, but we believed it was due to the test doing a 45-degree rotation instead of a full revolution, meaning that the scene did not look the same. This lead us to introduce an additional test to test the stability of the system which can be seen in appendix B. Additionally, we only tested the system on a single machine, which means that the results might not be generalizable to other machines.

Chapter 5
Conclusions

This chapter will conclude our thesis by answering the research questions posed in the introduction. We will also discuss possible future work that could be done to improve our implementation.

5.1 Answers to Research Questions

- **How well does the new system for probe and surfel placement reduce the number of placed probes while retaining a similar, or producing a better, visual quality compared to the previous system in a more complex environment?**
  From the results we can see that the octree is successfully able to reduce the amount of probes from 9261 to around 30 at depth 2 and 80 at depth 3, keeping similar values during animation. This is a significant reduction in the number of probes needed while still maintaining a reasonable visual quality. However, this came at a great expense in terms of performance in the hit shader.

- **How significantly does the choice of spherical lattice affect the surfel placement?**
  The results also showed that the use of a Fibonacci lattice did, on average, slightly improve the surfel hit fractions in contrast to the use of a latitude-longitude lattice, resulting in better probe placements. We do note, however, that this improvement is only marginal, and the visual differences were also miniscule.

5.2 Future Work

There exist several areas of improvement for the current implementation of our thesis. This section lists some possible future works.

5.2.1 Probe Lighting and Sampling

As described in section 3.3, the current system does not have an optimal solution when it comes to processing and sampling the probes and this is the main reason for the high execution times. It would be better if only a subset of all probes in the scene were considered for a hit point. We believe that using the octree or an auxiliary structure to encode which probes are visible from a given hit point would be a good solution to this problem. We see this as the most important area to improve in order to make the system more efficient, since the vast majority of the frame time is taken up by the closest hit shader and not the octree update shaders.

5.2.2 Splitting Condition

As it currently stands, we are using a simple splitting condition to determine when a node in the octree should subdivide or be left as a leaf. This condition is based on the surfel hit fraction for a probe as a range of a lower and upper bound. This condition could be improved since it does not account for situations where we could lose information about the geometry in the scene. One such example would be when a node in the octree is very close to a wall and decides to subdivide. After subdividing the new children might be inside geometry or not see the wall anymore. Taking other aspects than just the number of surfels placed into account when deciding to split, such as surface properties, could also be a way of improving the system.

5.2.3 Ground Truth

In order to evaluate the visual quality of our implementation, we simply empirically compared the images produced by our system to the previous. Having a better way to measure visual quality by, for example, having access to a ground truth image, would allow for a more accurate evaluation of the image quality and how well the new system performs in that regard.

References

[1] Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michał Iwanicki, and Sébastien Hillaire. *Real-Time Rendering 4th Edition*. A K Peters/CRC Press, Boca Raton, FL, USA, 2018.

[2] Elmer Dellson. Dynamic diffuse global illumination using probes and surfels, 2023. Student Paper.

[3] Sergey Egelsky. Human skull. https://sketchfab.com/3d-models/human-skull-b0251e48e906418ebae34b7f811ca065, 2019.

[4] Álvaro González. Measurement of areas on a sphere using fibonacci and latitude–longitude lattices. *Mathematical Geosciences*, 42(1):49–64, Jan 2010.

[5] Jie Guo, Zijing Zong, Yadong Song, Xihao Fu, Chengzhi Tao, Yanwen Guo, and Ling-Qi Yan. Efficient light probes for real-time global illumination. *ACM Trans. Graph.*, 41(6), nov 2022.

[6] Henrik Halén, Andreas Brinck, Kyle Hayward, and Bei Xiangshun. Global illumination based on surfels, 2021.

[7] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, aug 1986.

[8] Zander Majercik, Jean-Philippe Guertin, Derek Nowrouzezahrai, and Morgan McGuire. Dynamic diffuse global illumination with ray-traced irradiance fields. *Journal of Computer Graphics Techniques (JCGT)*, 8(2):1–30, June 2019.

[9] Donald Meagher. Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer, 10 1980.

[10] Microsoft. Compute pipeline. `https://learn.microsoft.com/en-us/windows/uwp/graphics-concepts/compute-pipeline`. Accessed: 2024-06-02.

[11] Microsoft. Directx raytracing (dxr) functional spec. `https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html`. Accessed: 2024-06-02.

[12] Microsoft. Queries. https://learn.microsoft.com/en-us/windows/win32/direct3d12/queries. Accessed: 2024-06-07.

[13] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: surface elements as rendering primitives. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, page 335–342, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[14] Martin Roberts. How to evenly distribute points on a sphere more effectively than the canonical fibonacci lattice. https://extremelearning.com.au/how-to-evenly-distribute-points-on-a-sphere-more-effectively-than-the-canonical-fibonacci-lattice/. Accessed: 2024-06-03.

[15] Nikolay Stefanov. Global illumination in tom clancy's the division. https://www.youtube.com/watch?v=04YUZ3bWAyg. Accessed: 2024-06-07.

[16] Hongqi Zhang. Design and implementation of a global illumination rendering system based on surfels. In *2023 8th International Conference on Intelligent Informatics and Biomedical Sciences (ICIIBMS)*, volume 8, pages 548–553, 2023.

Appendices

Appendix A
Average Probe Counts and Surfel Hits

Here follows a set of heatmaps showing the average probe counts and average surfel hit fractions as measured from the different scenes and configurations.

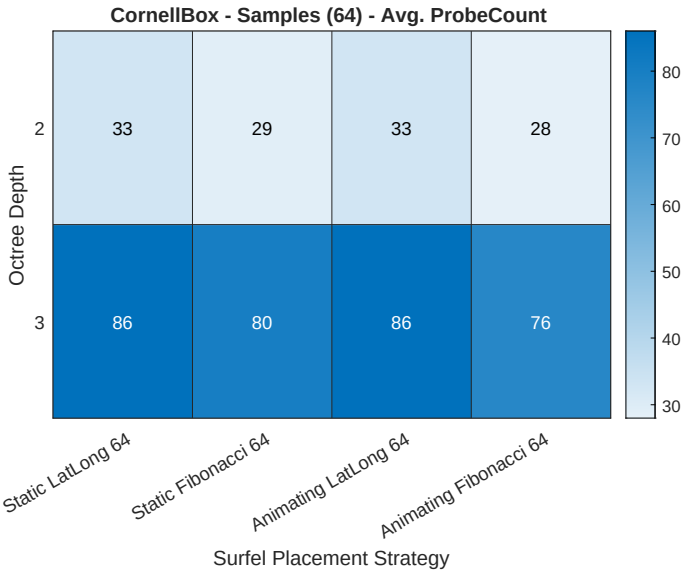


Figure A.1: Average probe counts for CornellBox using 64 surfels per probe.

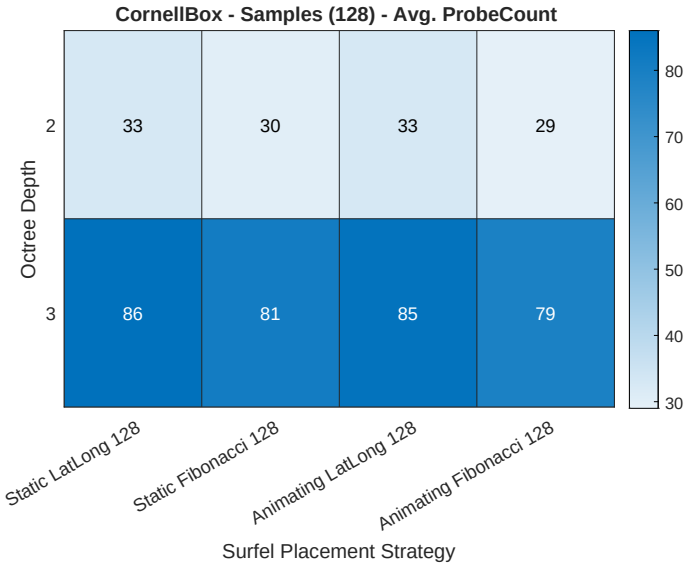


Figure A.2: Average probe counts for CornellBox using 128 surfels per probe.

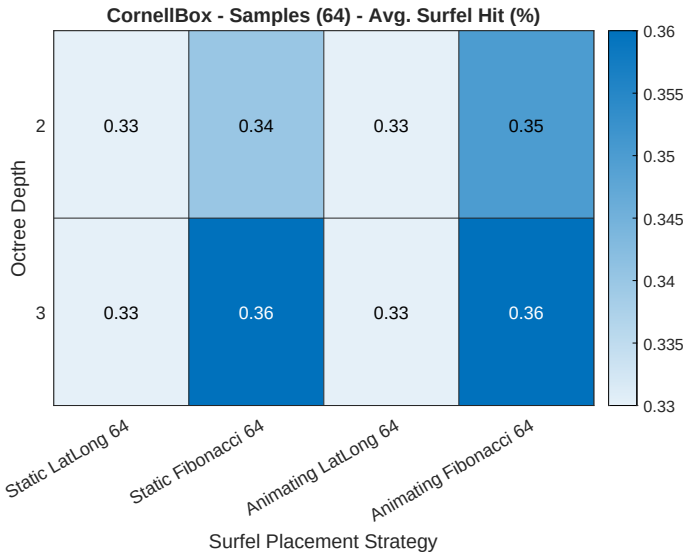


Figure A.3: Average surfel hit fractions for CornellBox using 64 surfels per probe.

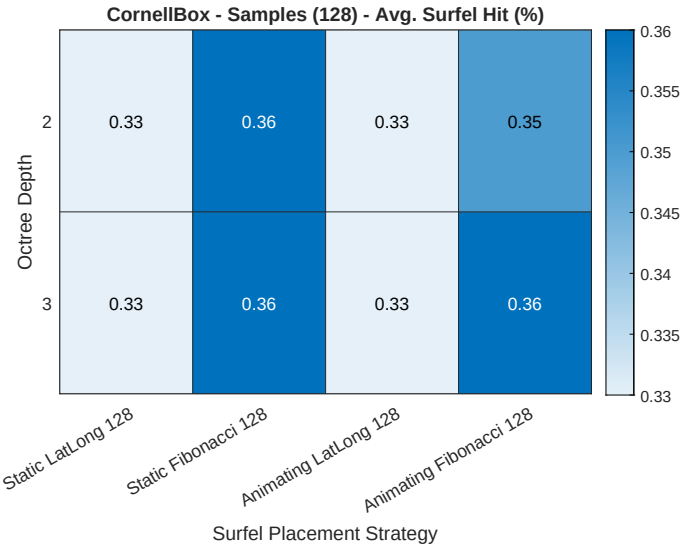


Figure A.4: Average surfel hit fractions for CornellBox using 128 surfels per probe.

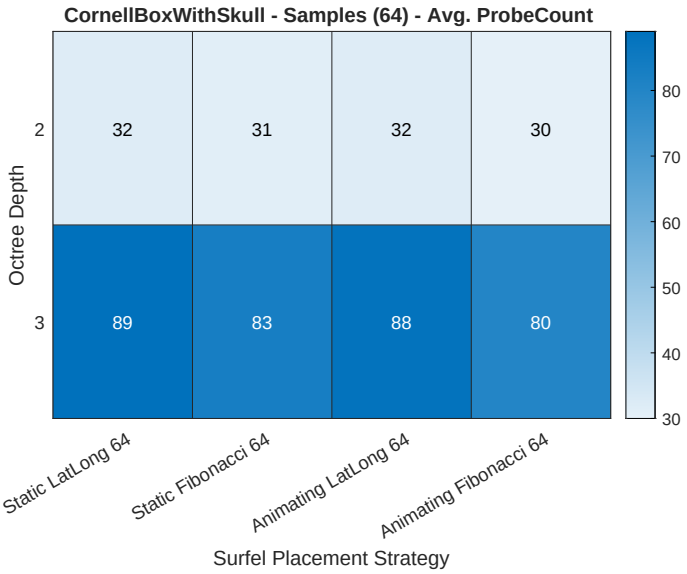


Figure A.5: Average probe counts for CornellBoxWithSkull using 64 surfels per probe.

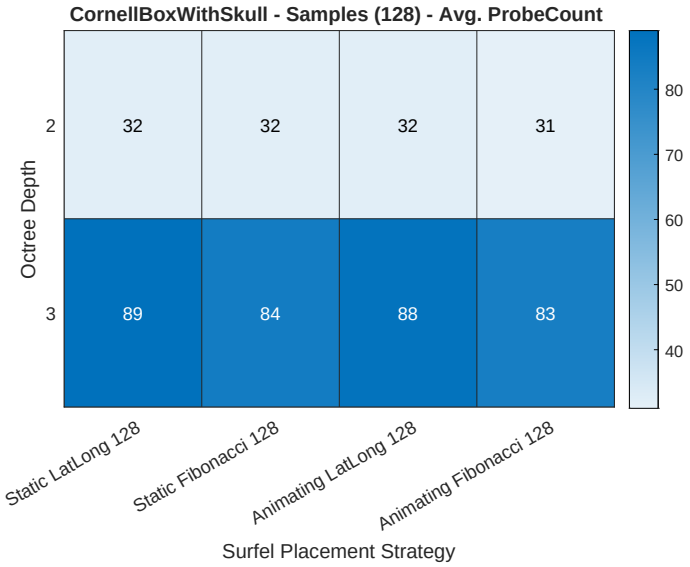


Figure A.6: Average probe counts for CornellBoxWithSkull using 128 surfels per probe.

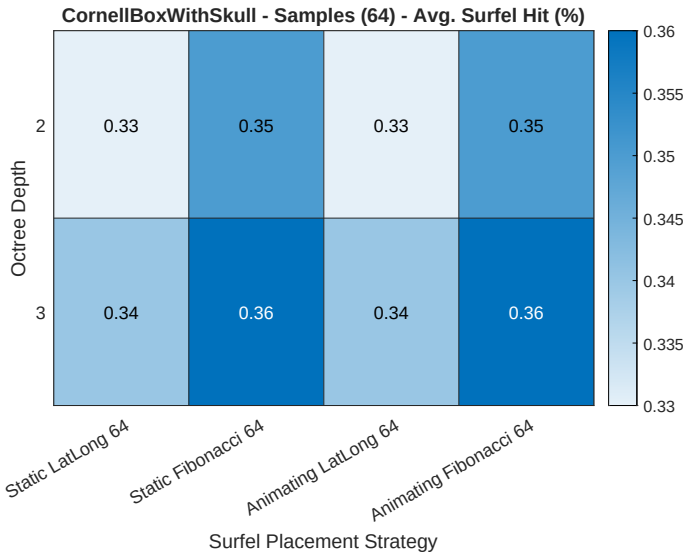


Figure A.7: Average surfel hit fractions for CornellBoxWithSkull using 64 surfels per probe.

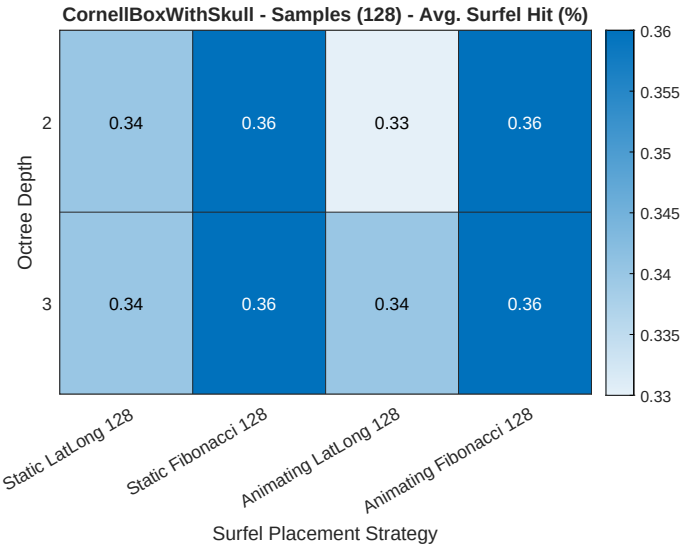


Figure A.8: Average surfel hit fractions for CornellBoxWithSkull using 128 surfels per probe.

Appendix B
Additional Measurements

Here follows an additional measurement in order to test the stability of the octree. In an effort to minimize the amount of data we only show this from one configuration. The test rotated the objects in the scene for one full revolution around their vertical axis over 150 frames and then paused for 50 frames. This was repeated three times for a total of 600 frames. In figure B.1, the marked data points are the current probe count in the scene after each rotation.

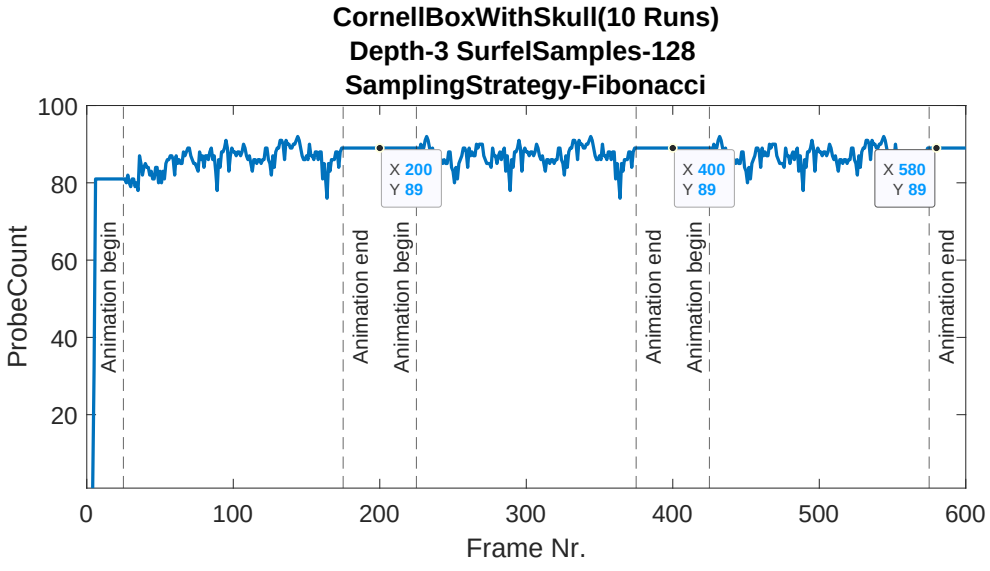


Figure B.1: Probe count over time for the CornellBoxWithSkull scene using 128 surfels per probe and a Fibonacci surfel placement method.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2024-06-13

EXAMENSARBETE Improving Probe and Surfel Placement for Dynamic Diffuse Global Illumination
STUDENTER Patrik Fjellstedt, Martin Antoniev
HANDLEDARE Michael Doggett (LTH), Calle Lejdfors (AMD)
EXAMINATOR Per Andersson (LTH)

Smartare placering av ljussonder för ljussättning av 3D-grafik

POPULÄRVETENSKAPLIG SAMMANFATTNING **Patrik Fjellstedt, Martin Antoniev**

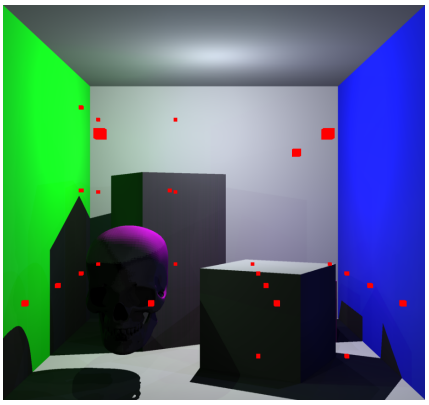
Vi har i det här examensarbetet utvecklat en metod som på ett smartare sätt placerar ljussonder i en tredimensionell virtuell värld för att beräkna ljus som reflekterats. Den nya metoden lyckades markant minska antalet ljussonder i en enkel testvärld till färre än en hundradel av det ursprungliga antalet och ändå ge liknande ljussättning.

En av de större svårigheterna och tidskrävande problemen att lösa när det handlar om tredimensionell datorgrafik, är att ljussätta den virtuella världen på ett sätt som är övertygande för det mänskliga ögat. Det har på grund av detta tillkommit en rad olika lösningar genom tiderna, där en av dem är användningen av ljussonder.

En ljussond kan liknas vid en kamera som ser i alla riktningar omkring sig och placeras i den virtuella världen för att samla in och beskriva det reflekterade ljuset som den kan se, vilket i sin tur sedan kan användas för att ljussätta delar av världen med ljus som reflekterats.

Hur kan man då gå tillväga för att placera ljussonderna i världen? Ett väldigt enkelt sätt är att lägga dem i ett tredimensionellt rutnät med litet avstånd mellan dem. Problemet med detta är att det snabbt blir många ljussonder som inte tillför mycket alls till världens ljussättning. I många fall hade man egentligen kunnat använda flertalet färre ljussonder och uppnå liknande resultat, och på så sätt även spara på resurser.

I det här arbetet har vi utgått från ett system där ljussonderna placerades i just ett sådant rutnät och utvecklat en metod för att istället placera dem på ett smartare sätt. Detta kunde vi åstadkomma genom att även låta ljussonderna bestämma ifall deras närliggande omgivning är intressant eller ej, och på så sätt antingen behålla ljussonder, placera fler, eller ta bort dem.



Bilden ovan visar ett exempel på hur vår metod på ett klokt sätt kan placera ljussonder, markerade i rött, där den övre delen av världen får färre då där finns ett större tomrum. Bilden visar också att ljussonderna lyckats se, bland annat, ljuset från taket som reflekterats på de färgade väggarna, vilket sedan använts till att färglägga andra delar av världen.