Marc Aurel Kastner

Supervisor Pablo Bauszat

Referee Prof. Dr. Ing. Marcus Magnor

Co-Referee Prof. Dr. Ing. Friedrich M. Wahl

The Splitted Grid – An acceleration structure for ray tracing

Bachelor Thesis

April 2, 2013

Computer Graphics Lab, TU Braunschweig

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Braunschweig, 2. April 2013

Marc Aurel Kastner

Zusammenfassung

Uniform Grid ist eine häufig genutzte Datenstruktur zum Beschleunigen von Ray Tracing Verfahren. Dennoch hat das klassische Grid einige Nachteile, wie den hohen Speicherverbrauch, der proportional mit der Auflösung zunimmt und die daraus resultierenden Probleme mit dichter Geometrie in großen Szenen umzugehen (vgl. mit dem so genannten „Teepot im Stadion“-Problem). Dennoch sind ihre Konstruktions- und Traversierungszeiten sehr schnell im Vergleich zu Bounding Volume Hierarchien (BVH) und anderen hierarchischen Ansätzen.

Diese Abschlussarbeit beschreibt eine neue Beschleinigungsdatenstruktur für Ray Tracing, die Vorteile aus vorherigen Methoden kombiniert. Sie basiert auf Ansätzen von hierarchischen Raum-unterteilenden Datenstrukturen sowie verschachtelten Grids. Der Algorithmus wird in einer naiven Variante, die 1-dimensionale Grids in einer Hierarchie verschachtelt, eingeführt. Durch die Struktur ist es weiterhin möglich mit Techniken ähnlich dem 3D-DDA Algorithmus zu traversieren. Heuristische Methoden zur adaptiven Berechnung von Parametern wie der Auflösung und Tiefe der Datenstruktur werden evaluiert. Darüber hinaus wird eine speziellere Variante der Datenstruktur präsentiert, in der eine Loose-Speicherung zum Umgehen doppelter Referenzen eingesetzt wird. Beide Varianten werden auf der CPU implementiert, im Detail evaluiert und in bekannten Szenen mit anderen Datenstrukturen verglichen, die dem Stand der Forschung entsprechen.

Abstract

Uniform Grids are commonly employed acceleration structures for speeding up ray tracing. Unfortunately, the classic Grid suffers from several problems such as a high memory footprint, the inability to handle clustered geometry (tea-pot-in-a-stadium problem) and a scaling linear to its resolution. However, their construction and traversal times are slightly faster compared to Bounding Volume Hierarchies (BVH) or other hierarchical approaches. Extensions to the uniform Grid are Hierarchical or Nested grids, which try to overcome some of the known drawbacks.

This thesis provides a new acceleration data structure for ray tracing that takes the advantage of the best of previous methods. It combines common approaches of hierarchical space-partitioning data structures with nested grids. The algorithm is introduced in a naive way which nests 1-dimensional grids in a hierarchy. Due to its structure, it can use techniques similar to 3D-DDA from uniform grids for traversal. Heuristic approaches to calculate parameters like resolution and hierarchy depth adaptively are evaluated. Furthermore, an additional technique to reduce the number of double references by using a loose variant is presented. Both approaches are implemented on CPU, evaluated in detail with common scenes and compared to other state-of-the-art acceleration data structures.

Contents

List of Figures

Chapter 1

Introduction

After over 20 years of research and development in areas of computer graphics and ray tracing, the topic still gains interest and popularity. Usually, ray tracing approaches can be divided in two fields of usage.

First, makers of computer-animated films or computer-animated imagery (CGI) already use ray tracing for the last decade in a variety of different movies. As it is possible to approximate graphical phenomena like global illumination and perfect mirroring in a physically based rendering system, especially path tracing is getting a popular method to create *realistically looking images*. In CGI applications, raytracing is often used in *offline rendering*, meaning there is no important need for speed optimization as render farms or clouds are used. Optimizations still have a big impact on rendering speed for CGI applications, though. For big scenes and complex structures, single pictures may still take days of rendering, so speed optimizations can often mean big differences in rendering times of full movies. After all, the goal in these areas are physical correctness and high-quality graphics.

Second, the development in interactive ray tracing, which means real time applications, is ongoing. With graphics cards getting more powerful in fields of parallel computation, the move to a GPU ray tracer using CUDA or OpenCL made real time ray tracing possible. As a result, future video game engines in development start using hybrid techniques or ray tracing instead of solely rasterization (like DirectX or OpenGL). In rasterization techniques, complexer graphical phenomena are usually approximated or simply *faked* by manual effort from artists and game designers, which results in higher costs developing these games. For real time uses, speed optimization is the most important issue.

Today, a lot of research done in field of raytracing is based on increasing the performance, while maintaining most of its physically correctness, which targets the interactive ray tracing sector. The key idea is to decrease the amount of mathematical calculations used, or approximating values which would be expensive to calculate in other ways. For example, using statistical approaches like *Monte-Carlo-Integration*, complex integrals can be solved numerically by using a high sampled random distribution.

An usual approach to receive performance are so-called *acceleration structures*, which optimize a given scene in a pre-procession step into a data structure, which better fits the way, a ray tracer accesses its data. Therefore, reducing the amount of ray-geometry intersection tests. While there are a variety of common approaches, each one has downsides and problems.

In this thesis, a new approach is presented, which tries to combine advantages of classic methods to algorithm with fewer downsides. Using a combination of usual space cutting techniques like kD-Trees and Octrees with the concept of Nested Grids, the new approach is more complex, but usually fits the scene in a faster way, resulting in good traversal performance. Both the construction and traversal algorithm will be presented, as well as additional techniques to speed up this approach. A second variant will feature a way to get faster construction time and lower memory usage by using a loose approach.

1.1 Outline of this thesis

This thesis is structured in the way, one would approach a new topic. After a small introduction of the basics about ray tracing and acceleration structures in Chapter 2, the theoretical concept of the main idea is explained in Chapter 3.

This main chapter will give a deep look into the mechanics of construction and traversal algorithms as well as additional approaches to raise their performance. Section 3.4 will introduce the idea of the loose variant on top of the naive concept. Afterwards, Chapter 4 will explain the implementation of all algorithms explained in previous chapter in a pseudo-code manner. In Chapter 5, reference implementations in C++ are used in a variety of scenes, giving a comprehensive analysis on how these approaches compete with other state-of-the-art acceleration structures.

Finally, the thesis concludes the analysis with a summary on how well current implementations perform against other approaches, as well as some brief hints for not-yet-tested ideas, which can be used in potential future work regarding the Splitted Grid or similar acceleration structures in 6.

Chapter 2

Related Work

This chapter will explain background information and theory of ray tracing, giving an introduction to the topic.

First, Section 2.1 will show how ray tracing works in theory, giving examples of different techniques to approximate physically correctness, and how performance will be achieved by using several techniques of the specialized algorithms, like *Whitted Style Integration*. Afterwards, the usage and advantages of acceleration structures are shown in Section 2.2. Therefore, the most important state-of-the-art acceleration structures will be explained as examples for this technique.

2.1 Ray Tracing Algorithm

A typical computer monitor has a raster of pixels, each having a different color. Obviously, a picture rendered by raytracers is also of a given size of NxM pixels, each having a specific color value which results in a full picture from distance for the human eye.

In a real life ray tracing application, the goal typically is to create this image by shooting $N * M$ rays into a scene consisting of geometric primitives and getting color values as a result of calculation, forming the wanted picture.

Each ray $R(t) = o + t * d$, described by its origin o and its direction d, is intersected with the geometry in the scene. The ray origin is the point of view of the camera, which views the scene, whereas the direction depends on the pixel position in the resulting image. Imagine a pinhole camera concept, where each ray is a $\frac{1}{N*M}$th of the full picture, each in a little bit different direction to get a full picture.

After intersection with geometry, the parameter t is set to distance to intersection point p with geometry. In usual application, one would want the *closest* intersection to the origin. The color at point p is saved for the pixel which represents the current ray. Determining the color at a given surface is done by so-called *shading models*. When there is no intersection point with geometry, the color value is typically set to a default value, which is often set to black.

This way, an image without shadows, refraction or reflection is created, which is called *Ray casting*. In this very simple way of raytracing, complexer shading techniques using textures are usually not used.

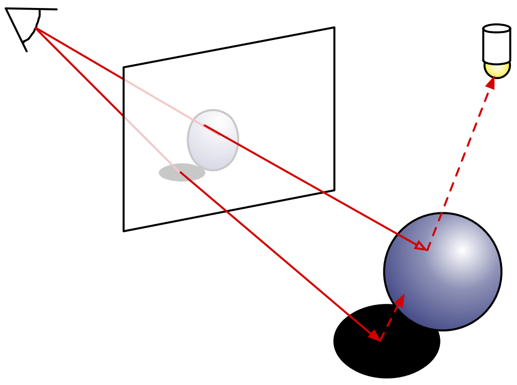


Figure 2.1: **Ray tracing:** This example shows, how rays from the eye are casted in direction of a given scene consisting of a purple sphere. At the point of intersection with the surface, shadow rays (dotted) are casted in direction of the light source. At the bottom ray, the casted shadow ray hits the sphere before light source, resulting in shadow in a rendered image.

To receive more complex visual phenomena like shadows, refraction or reflection, there are different ways, which are a trade-off between performance and physically correctness. A simple approach to this problem is called *Whitted Style Integration* [Whi80]. To create shadows, it casts new rays $R_l(t) = o_p + t * d_l, \forall l \in L$, where L is the set of all light sources in the scene. This time, resulting rays are intersected with the light source l. If it hits another geometry between point p and l, the current point p is occluded by another object, therefore, the color is darkened. If not, the color value is used as before. These additional rays are called *shadow rays*. The basic concept of ray casting and shadow rays is visualized in figure 2.1. Similarly, the same technique is used to get reflection and refraction on surfaces, by adding additional rays in direction of refraction (using Snell's law and similar formula).

Figure 2.2 shows a simplified pseudo code for a minimal ray tracer. Note that the function *closest_intersection_point()* is an expensive task to do. In

```
def traceImage(scene, image):
  for (i,j) in image:
    ray = pixelToWorld(i,j)
    point = closest_intersection_point(ray, scene)
    intensity = 0.0

    for light in lightsources:
      shadowray = calculateRay(light)
      if intersect(shadowray, scene) == 0:
        increaseIntensity(intensity)

    image[i,j] = intensity * shade(point)
```

Figure 2.2: Pseudo Code for ray tracing

the simplest way to achieve this, it does an intersection test for every ray with every primitive in geometry, which results in a complexity of $O(n)$ per ray. The function *shade()* refers to a shading function determining the color at this point of surface. As an example, the Phong shading model could be used, as introduced by Phong [Pho75].

As intersection tests are actually the biggest part of calculations in runtime of a ray tracer, much ideas to reduce the number of these test has been developed.

2.1.1 Monte-Carlo Integration

A task, which costs additional time, is the physically correct rendering of shadows and lightning, by casting additional rays on intersection points. Physically, the task to receive light radiance on a point of surface is to solve the rendering equation, as presented by Kajiya [Kaj86]:

$$L_o(x, \vec{\omega}) = L_e(x, \vec{\omega}) + \int_\Omega f_r(x, \vec{\omega}', \vec{\omega})\, L_i(x, \vec{\omega}')\, (\vec{\omega}' \cdot \vec{n})\, d\vec{\omega}', \tag{2.1}$$

where L_o is the light radiance on the point x to direction $\vec{\omega}$, L_e is the light emission at the point x to direction $\vec{\omega}$, Ω is the unit hemisphere, f_r a function determining the proportion of light reflected from $\vec{\omega}'$ to $\vec{\omega}$ called *bidirectional reflectance distribution function*, L_i the incoming light on point x from direction $\vec{\omega}'$ and $\vec{n}$ describing the normal at point x. This equation is not solvable efficiently, as it contains an integral. Therefore, couple of tweaks are needed to achieve performance and a faster approximation of the rendering equation.

As previously noted, *Whitted Style Integration* is a way to achieve this approximation by casting single additional rays for shadows, refraction and lightning. As a single ray is casted per surface hit point, the approximation is nowhere near an integral of incoming light intensity, therefor lacking global illumination. In simple shading models, like *Phong shading*, an ambient light term is used to achieve a similar, but more unrealistic[1], effect.

In a lot of cases of ray tracing, one wants to have global illumination. In a pathtracer, each intersection of rays with surfaces will create additional random distributed rays, a technique introduced by Kajiya [Kaj86]. This way, a *Monte-Carlo Integration* is used to find a numerical solution for the equation 2.1. The number of additionally created random rays is chosen arbitrarily, but there is noise on images created with a low amount of samples. There are ways to improve the speed of path tracing like *bidirectional path tracing*, where some rays are casted from the eye, whereas others are casted from light sources, introduced by Lafortune [Laf96].

2.1.2 Acceleration

As noted previously, a high impact on performance is the amount of intersection test done. Acceleration structures are a way to put primitives in a three dimensional geometry to a more organized structure. This way, the performance of ray tracing can be severely increased by decreasing the amount of intersection tests with geometry. A more detailed view into acceleration structures is presented in section 2.2.

An obviously big impact on performance is hardware. Traditionally, ray tracers have been implemented on CPUs, therefore getting faster with every new generation of processors. Since the availability of multi-core processors, parallel computing is getting popular. A ray tracer is naturally a good usage of parallel computing, as each ray can be calculated separately, giving the possibility to use up to one thread per ray.

With *OpenCL* and *CUDA* getting more popular, GPU-accelerated ray tracers also gained popularity. Modern graphic cards are already optimized for floating point operations, specifically doing linear algebra, which are a big part of ray tracing, giving the high amount of intersection tests. Additionally, GPU computation are highly parallelized, which also benefits ray tracing. Therefore, porting a ray tracer to the GPU does make it possible to render simple ray tracing applications in real time.

For a deeper view into ray tracing techniques and possibilities for acceleration, additional information can be found at [Gla89] [PH10].

2.2 Acceleration structures

Acceleration structures have a very high impact on performance. When testing each ray with all primitives, one would get a runtime of $O(n)$. Therefore,

[1]These model do not adhere the law of conservation of energy.

a lot of work has been put into researching different kinds of acceleration structures. Kay et al. [KK86] presented an approach to reduce the number of intersection tests, by creating a bounding volume around the scene and testing if the ray hits the bounding volume. The main idea is to reduce the number of intersection tests for rays which go in a direction, where no primitives are.

Usually, in a more advanced approach, this will be achieved by pre-processing the given scene by dividing the space into smaller parts. Depending on the technique, either the space itself can be split, or the object inside the space are split in a way, a structure can be constructed to organize the primitives in a better way to access the data.

Afterwards rays can traverse the scene more efficiently, because the pre-processing assures, that the rays will not have to intersection test with each primitive anymore. For example, a ray which directs into the lower left corner of the scene will only have to test intersection with primitives, which are in the lower left corner. This information is achieved by traversing the acceleration structure. Using a binary acceleration structure, it achieves a logarithmic runtime ($O(\log n)$).

As the pre-processed data is saved in memory, acceleration structures have an impact on memory usage of the raytracer though. There are a few implicit techniques, which try to avoid memory usage while still using pre-processed structures. This is possible and used by Eisemann et al. [EBM12], but usually much slower than using a structure with memory usage. The reason is, in an implicit approach, trees must be calculated while traversing the tree itself, which are a lot of additional calculations while traversing. Also, the quality of implicit approaches does not compete with usual techniques, as there is no proper focus on researching these. Another approach by Áfra [Áfr12] proposed an implicit way of accelerating raytracing by using divide and conquer techniques.

2.2.1 BVH-Tree

The idea, by Rubin et al. [RW80] and Kay et al. [KK86], is to create a hierarchy of bounding volumes, the root being the scene bounding box, with children nodes being subparts of the scene bounding box and leaf nodes are bounding boxes of single primitives, or groups of primitives. This approach is called *Bounding Volume Hierarchy* (BVH) and still a very popular to solve this problem. The way to construct a BVH has major influences in the speed increase, and there is a trade-off between construction time and traversal time.

A BVH tree is usually built as a binary tree without duplicated references, so when traversing a child, one half of primitives are cut. Due to the design of a BVH, the memory usage is linear and predictable. Each node in a BVH stores a full bounding box.

An overview of different build methods for BVHs is presented by Wald et al. [WBS07], including the *surface area heuristic*, short SAH, which minimizes a cost function to determine the best split on each level while constructing the acceleration structure. A cost estimate is usually similar to:

$$Cost(c) = 2 * T + I * \frac{1}{A} * (A_1 * p_1 + A_2 * p_2), \tag{2.2}$$

with A being the area of the overall bounding box, while A_i being the area for a children node, and p being the number of primitives which overlap a children node. T and I are constants for traversal and cost estimates.

There are very fast construction methods for BVH-SAH as used by Wald et al. [Wal07] [WIP08], so it is a good approach to use for real time applications, where animations and therefore moving primitives take place. There are other methods like *spatial medium split* or *object medium split*, but these are usually slower than a SAH approach in traversal speed and therefore losing relevance.

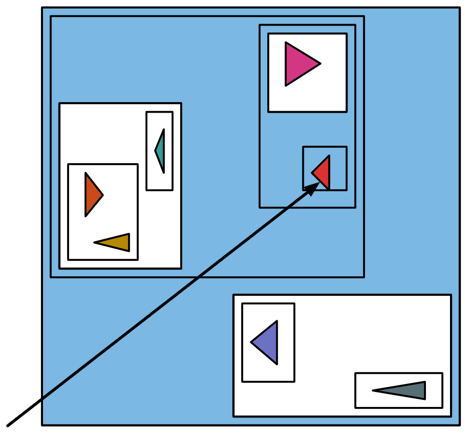


Figure 2.3: **BVH Example:** A two-dimensional example of a bounding volume hierarchy. The blue area are nodes accessed, when traversing this specific ray.

Usually, a BVH tree is used in applications like real time with animations, because of very fast constructing methods and low memory usage. For GPU raytracing, efficient stack-less traversal methods are existing, like presented by Hapala et al. [HDW$^+$11].

2.2.2 kD-Tree

A kD-Tree is a space-partitioning acceleration structure introduced by Bentley [Ben75]. The name stands for k-dimensional tree. A kD-Tree uses one dimensional splitting planes to cut space in halves in a round robin manner through dimensions. The result is an unbalanced binary tree.

Today, a kD-Tree commonly uses SAH construction methods, using heuristic cost estimates to calculate the position of splitting plane, similar to those explained in previous subsection for a BVH in equation 3.1. Same as a BVH, there are other construction methods like SMS or OMS available, which usually result in slower traversal speeds.

Usually, a kD tree has the fastest traversal speed in common approaches for raytracing, but has slight disadvantages in construction time and memory usage. For GPU raytracing, there are high performance traversal algorithms like by Popov et al. [PGSS07].

2.2.3 Grid

A grid is a space partitioning acceleration structure for raytracing, introduced by Fujimoto et al. [FTI86]. In its most common approach, the Uniform Grid, it divides the three dimensional space into NxMxO cells of same size, so called *Voxels*. Each primitive in the scene will get assigned to every voxel it overlaps.

The construction times for grids are usually extremely fast, because each primitive only uses constant time to find overlapping voxels, resulting in $O(1)$. As the construction only depends on projections, not involving a hierarchy, efficient parallel algorithms like Kalojanov et al. [KS09], can be used.

A problem for grids are very big primitives and the question how to choose the optimal resolution, because bigger primitives overlap a large number of voxels, which result in a very high memory usage and traversal time through duplicated references and therefore duplicated intersection tests.

It is possible to use a *three-dimensional digital difference analyzer* algorithm, short 3DDDA, to traverse the data structure, presented by Fujimoto et al. [FTI86] and optimized by Amanatides et al. [AW87]. This algorithm steps through the grid in a linear manner. A visualization of this traversal process is presented in figure 2.4

While used often in the past, recently, Grids are used fewer, because of their high memory usage and problems with very big primitives. Another problem is the choice of resolution in scenes with the so-called *teapot-in-the-stadium* problem. This problem is named by a small teapot in a big stadium and refers to a scene, where small high resoluted models are clustered inside of big areas with usually wide areas of empty space, but often surrounded by lower resoluted geometry. This is typically the case in architectural scenes.

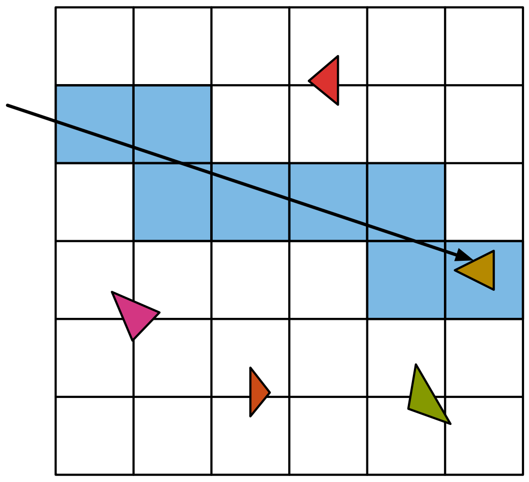


Figure 2.4: **Grid Example:** A two-dimensional example of a uniform grid. The blue area are nodes accessed, when traversing this specific ray.

Resulting, in a scene having this scenario, the average size of primitives is very small compared to the space, and often there are a few very big primitives in contrary.

Using a Grid for these scenes lead to issues: A high resolution would cause a lot of unused cells to traverse, while too low resolutions cause slow traversal speeds near to objects. Also, bigger primitives would lie in a high amount of cells.

Because of its very fast construction time of $O(n)$, there are newer concepts and research projects building upon the Grid.

To avoid typical problems of Grids, there are some approaches worth mentioning. There are *Hierarchical* or *Nested* Grids, variations of a Grid, which try to avoid the resolution problems by nesting a Grid into cells of a Grid and therefore resulting in a hierarchical acceleration structure.

In a simple approach, Jevans et al. [JW89] used a *Recursive Grid* to nest Uniform Grids into each other. This approach usually nests Grids of the same dimension and resolution. Klimaszewski [KS97] proposed a variant called *Adaptive Grid*, where he preprocessed the geometry to create clusters of objects and then creating BVH-like trees around them. Note that the resulting trees differ, because the first approach can not have overlapping cells, whereas the second one can. An usage of nested grids for GPU ray tracing is presented in Kalojanov et al. [KBS11].

2.2.4 Others

Aside from the popular ones, there are other examples of good acceleration structures which can be used in ray tracing.

The *Octree* is a space divisioning acceleration structure like a kD-Tree, but not as a binary tree, but having 8 childs per node, which has advantages in the way it fits to more complex scenes. An Octree has a special variation called *Loose Octree*, where duplicated references are avoided by projecting the mid-point of each primitive into a single node. To still traverse all wanted primitives, each primitive needs to be smaller than double the size of the node. Then, while traversing, the bounding box of each node is getting extended, so each ray will still intersection test all geometry needed.

Whang et al. [WSC$^+$95] proposes another variation of the Octree called *Octree-R*, which does not subdivide the space at spatial median point. It minimizes a cost function to calculate a best split point, similar to the approach of a surface area heuristic explained previously.

As previously mentioned, in a special form of BVH called *No-Memory-Hierarchy* (NMH), Eisemann et al. [EBM12] managed to create a BVH-like acceleration structure using no additional memory at all. This implicit variant sorts the primitives array in a specific way, in which the BVH structure can be calculated on traversal runtime. This of course is expensive, therefore being slower than other approaches. Using no memory at all, it can be used on embedded systems or for very big scenes on a GPU, where memory usually is an issue.

Another notable variant of a BVH is called *bounding interval hierarchy* (BIH), developed by Nam et al. [NS04] and Zachmann [Zac02] independently. The BIH is a combination of a BVH construction with a kD-Tree-like traversal algorithm. A BIH stores two planes per node, so it can both allow overlapping children like a BVH as well as ordering children along axes like a kD-Tree.

Chapter 3

Splitted Grid

As mentioned in previous chapter, acceleration structures have a huge impact on the performance of a ray tracer. While classic hierarchical approaches as well as Grid approaches both have their downsides, the following will introduce a new approach building upon these, trying to combine some advantages of either type of acceleration structures.

Section 3.1 will first introduce the idea of the Splitted Grid and gives theoretical background about how a Splitted Grid works. The following section 3.2 will then explain, how the acceleration structure will be built. Therefore, a naive algorithm for construction will be presented, and then enhanced by adding arbitrary choice of parameters per inner node and a heuristic way to determine these parameters, using a surface area heuristic. In 3.3, the traversal algorithm will be presented. The last section 3.4 will step further by introducing the Loose Splitted Grid. As its main concept is similar to the Splitted Grid, this section will not re-explain the basics, but concentrate on explaining how the loose variant differs.

3.1 Concept of Splitted Grids

The concept introduced in this thesis is called *Splitted Grid.* It tries to combine an Uniform Grid with the concept of a hierarchical data structure. The key concept is an one-dimensional grid which gets nested hierarchically. Usual approaches of *Hierarchical* or *Nested* Grids put three dimensional grids into each other, while this technique focuses on one dimension for each recursion step.

The root node is an inner node using the scene bounding box for determining its childs bounding boxes. On each recursion step, the space is split on one axis into n bounding volumes of the same size, each being a node. A node can either be an *inner* or *leaf* node. Each Splitted Grid inner node consists of n child nodes, which refer to either further inner nodes or leaf nodes. A leaf node holds the primitives.

Consider it this way: An inner node symbolizes a bounding box, and its child nodes split this bounding box at a given axis into n pieces with same volume. If a child node has only a small amount of primitives, it is a leaf node. If there are a lot of primitives, a new inner node gets created which repeats this process on the most efficient axis.

This way, the acceleration structure is able to adapt to the scene by choosing these parameters on a per-node basis. The amount of child nodes per inner node, as well as its splitting plane, can be chosen differently for each inner node. Figure 3.1 shows this idea in a two-dimensional example.

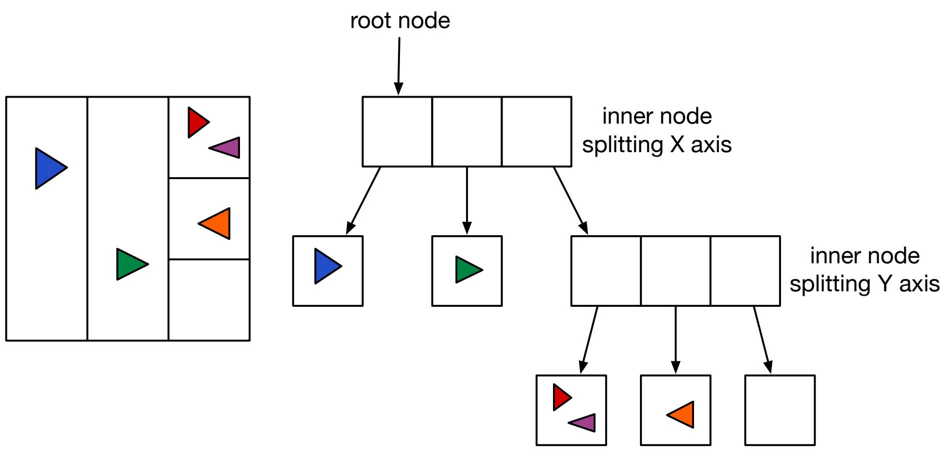
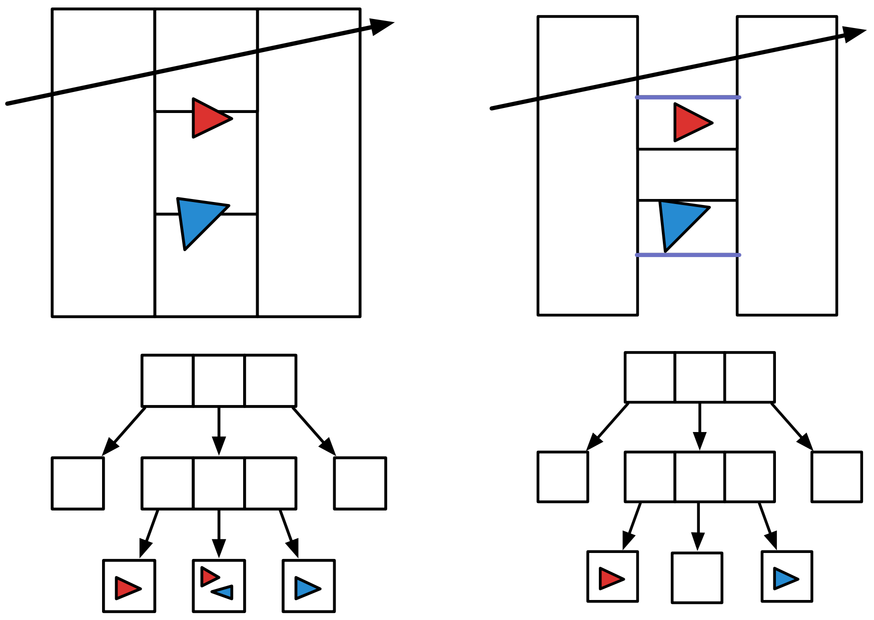


Figure 3.1: **Concept of a Splitted Grid:** A two-dimensional scene with 5 primitives is built into a Splitted Grid. The first level has an inner node with a resolution of 3, splitting the X axis. As there are more than two primitives in the most-right child node, we add another level of splitting. The node gets an inner node with again a resolution of 3, this time splitting the Y axis of the bounding box of its parent node. As a result, each leaf node has a maximum of two primitives. Note that there could be another recursion using a third inner node with resolution 2 splitting the X axis, to have a tree with no more than 1 primitive per leaf node, but in this case one leaf node holds two primitives.

For an additional performance increase, the Splitted Grid uses a concept called *bounding planes*. This technique introduces cutting off empty space. As shown in figure 3.2, the bounding box of a node is not equal to the overall bounding box of its contained primitives. In most cases, the calculated splitted bounding box is too large, not cutting away all space it could. At this point, it is possible to cut away the empty space of one dimension by using two planes - the *bounding planes*. These planes are limiting the area containing primitives. Afterwards, child nodes split the volume between the bounding planes instead of the whole volume of their parent node. Using this method, nodes can cut off a lot of the empty space on their axis, resulting in a tree better adapting to the scene. Additionally, traversal algorithms can skip whole parts of the tree when moving through empty space.

After all, each inner node needs to know its splitting axis, its resolution, an offset to its childs and to its bounding planes. A leaf node needs its count and offset to primitive indices array. As the leaf node does not need bounding planes for traversing, the bounding planes itself should be saved outside of the node, so only inner nodes will have them. In both cases, the needed values can be bit-shifted in two *integers*, which results in an usage 8 bytes per node. For each inner node, the bounding planes use two *floats*, which use additional 8 bytes. In memory, all nodes from 0 (root) to n (last node) are saved in an array. Using the offset in the inner nodes, m child nodes can be referenced on the n-th node as $n+1$ to $n+1+m$ in the array.



(a) Without bounding planes (b) With bounding planes

Figure 3.2: **Concept of Bounding Planes:** By introducing bounding planes (purple color), the Splitted Grid can cut off additional empty space in each inner node. This way, the traversal algorithm can skip party of the tree. Additionally, the limited space usually guides to a tree better adapting to the scene.

3.2 Construction of a Splitted Grid

There are different ways, how a Splitted Grid can get constructed. To get a very optimized fitting to the geometry presented in scene, the parameters for each scene must be chosen adaptively, which is complex and costs time. For different purposes, one can distinguish between two ways of construction, a naive approach, which uses default parameters for each inner node or slight variations, and a second one using complex algorithms to determine a set of parameters for each recursion step. Basically, there is a trade off between traversal time and construction time.

In a most naive way, the construction algorithm uses a default resolution for each inner node. For the order of splitting axis, it uses a round robin approach. Figure 3.3 shows a resulting tree for this naive way of construction. The efficiency of traversing this tree is depending on scene geometry. In a cubic scene with a wide distribution of even sized primitives, this simple way of construction often adapts to the scene in a fast manner. If there are bigger differences in size of primitives or length of axes, there are a few ways to optimize this naive approach.

In a Splitted Grid, poorly constructed trees often do not result in bad traversal times, but very high memory usages, because of either duplicated references, and/or empty cells.

Instead of choosing inner node splitting axes in a round robin manner, one can choose the longest axis of the parent bounding box. In another approach, the resolution of inner nodes could be changed depending on the tree depths, using lower resolution sizes deeper in the tree.

3.2.1 Heuristic construction using surface area heuristics

To get a tight fit to the geometry, the goal is to get a tree like in Figure 3.4, where both parameters are varying independent to depth or node order. Therefore, a heuristic method has been developed, which determines best choices for resolution and splitting axis on each recursion step.

To achieve this behavior, the idea is to use a similar approach to a surface area heuristic in a BVH or kD-Tree, as described in [Wal04]. In a Splitted Grid, there are two values, which must be calculated for each inner node, the resolution and the splitting axis.

In a surface area heuristic approach, the goal is to minimize a cost estimate for traversing a sub-tree. On each inner node, the algorithm guesses, based on the current situation and distribution of primitives to the children, which set of parameters might be the best for this node. For a Splitted Grid, there are two parameters, splitting axis and resolution, which are available in $n * m$ combinations.

For each combination c, the cost is calculated based on the distribution of primitives over the children:

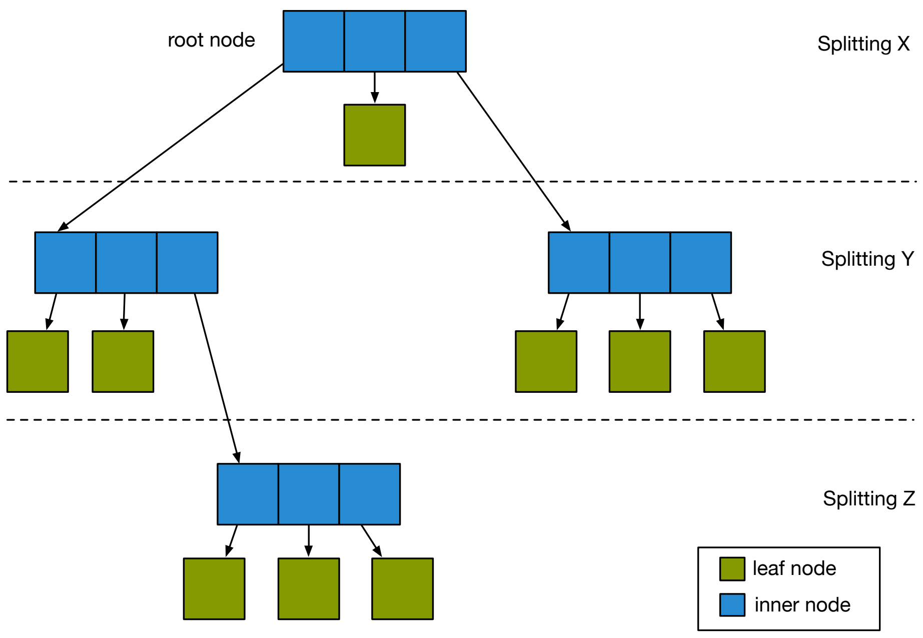


Figure 3.3: **Splitted Grid Uniform:** In an Uniform construction, the resolution of inner nodes does not vary. The splitting axis starts with X and then goes round robin.

$$Cost(c) = r * T + I * \frac{1}{A} * (\sum_{i=1}^{r} A_i * p_i), \tag{3.1}$$

with r being the resolution, A being the area of the overall bounding box while A_i being the area for a children node, and p_i being the number of primitives which overlap a children node. Afterwards, we are looking for the axis and resolution:

$$min(Cost(c)), \forall c \in C \tag{3.2}$$

with C being a set of all combinations of splitting axis and resolution. Note, there are two yet unnamed parameters T and I in equation 3.1, which are constants for traversing and intersection cost estimates.

In binary trees, the traversing cost is usually lower than intersection costs, because it only traverses two childs per step. In a Splitted Grid, this is not the case and therefore these constants are behaving differently, as there are up to r childs per recursion step. After all, a higher traversing cost is expected.

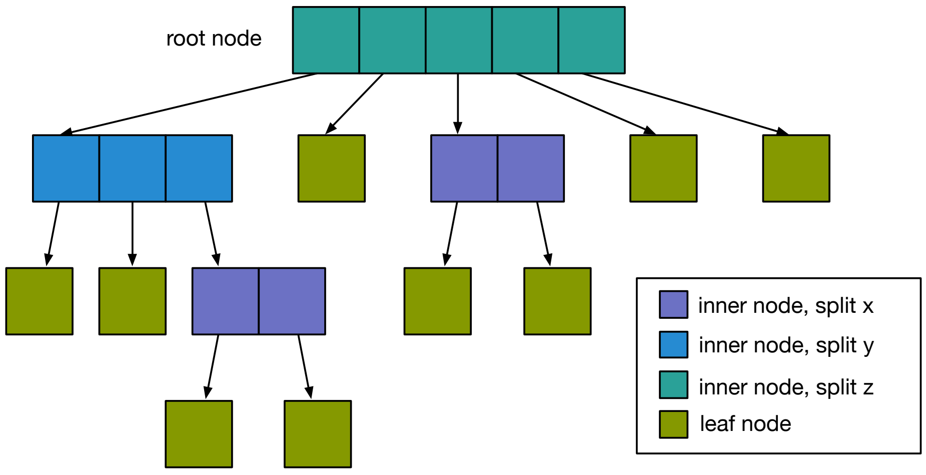


Figure 3.4: **Splitted Grid SAH:** In a heuristically approach, all inner node parameters are calculated on the fly. This way, a construction algorithm can fit the tree more easily to a complex scene.

3.3 Traversal

For a traversing the Splitted Grid, a usual hierarchical traversal method is used, but a few special changes ensure the decrease of tested nodes. For easier understanding, the following will reference to a recursive variant. The algorithm can be implemented iteratively for performance purposes.

On a given node, t_{Near} and t_{Far} for the ray intersecting the bounding box of the node gets calculated. Note that this calculation is cheap, because on every inner node, there are only two one dimensional bounding planes as explained in Figure 3.2, which need to be clipped against. This behavior is ensured, because every recursion level only splits one axis. Only the root node gets a full bounding box intersection test to exclude rays never hitting the scene.

If $t_{Far} > t_{Near}$, the node can be skipped. Otherwise, it is possible to calculate an entry and exit node based on the projection of the ray into a scale of 0 to n with n being the resolution of the current node

$$node_{entry/exit} = \frac{(o_{Ray} + d_{Ray} * t_{Near/Far}) - b}{size_{child}}, \tag{3.3}$$

with b being the lower value bounding plane and $size_{child}$ being the extent of a child node in axis direction, which is a constant value for all nodes in one inner node. Note that o_{Ray} and d_{Ray} are no vectors, but the value on the axis equal to splitting axis.

After calculating entry and exit nodes, the algorithm recursively calls itself on each node between the *entry*-th and the *exit*-th. Depending on ray direction, the calculated entry and exit cells need to be swapped to ensure the right order traversing children nodes. After each recursion call, it is possible that the ray already hit a primitive deeper in the tree. In this case, no further nodes need to be traversed. This can be checked by again comparing ray hit value t against the children bounds, which are cheap to calculate as both the lower bounding value as well as the extent of child nodes are known.

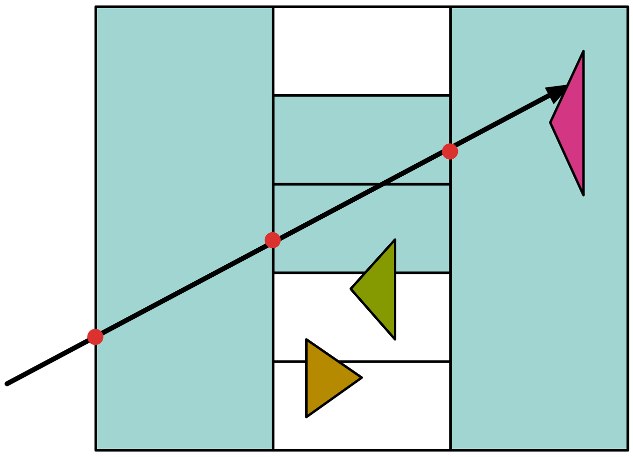


Figure 3.5: **Traversing a Splitted Grid:** In this example, a ray traverses a Splitted Grid from the bottom left to the upper right. The area marked in turquoise are the leaf nodes, which are traversed and therefore intersect tests are done for primitives overlapping these. The green triangle is tested, because it overlaps a turquoise one, whereas the yellow one isn't tested. Pairs of red dots are values for t_{Near} and t_{Far}.

Figure 3.5 shows an example of how traversal looks like. To make this example easier, no additional bounding planes are used. Instead, the bounding planes of the parent node is used. Pairs of red dots are values for t_{Near} and t_{Far}, which are used for determining the entry and exit nodes. Starting the traversal, the left bottom red dot as well as its correspondent red dot in the upper right corner (not shown as it does not get hit), will be used to calculate the entry and exit node. In this case, all three nodes must be traversed. After traversing the first one, which is an empty leaf node, at the second node it will meet another inner node, where it again calculates t_{Near} and t_{Far}. Here, the entry and exit node would be 3 and 2, when counting the 5 nodes from top to bottom. The algorithm now traverses the third node (leaf node, intersection test with the green primitive which does not get hit) and then the second (empty leaf node) node. As this is the exit node and tFar gets hit, the recursion goes one step up, to traverse the last, third, node of the root node. Inside, the pink triangle gets hit, therefore stopping traversal.

As the example already shows, it is possible to stop the traversal algorithm when $t_{Hit} < t_{Far}$. This is called *early out*.

3.4 The Loose Splitted Grid

The Loose Splitted Grid is a slightly different concept, which is built upon the Splitted Grid. The Splitted Grid construction method saves a primitive in each leaf node it is overlapping. Duplicated reference cause an additional *integer* to be saved each. As duplicated references are not foreseeable, the memory usage of a tree is not behaving predictable. These additional references can cause some severe memory issues for higher resoluted inner nodes and a high maximum depth, causing bigger primitives to be in every leaf node of an inner node.

A solution is a loose variant of the algorithm. In a loose variant, like previously explained for an Octree in Section , the mid-point of a primitive gets projected into a single leaf node. *Loose bounding planes* will save the distance between the leaf bounding box and the actual end of all primitives inside it. Afterwards, the traversal method needs to get adjusted, so all necessary nodes get traversed correctly, not leaving out primitives which are now in only in one node instead in all it overlaps.

This way of constructing a tree, no primitive is assigned to multiple nodes, because their mid-point only lie in an unique one. Now, all duplicated references are avoided, leading to a much smaller and more predictable use of memory.

As a result of this design change, the construction method immediately gets faster. Determining the node, a primitive falls in, is a thing of a projection, making the complexity of a recursion step for resolution r and n primitives $O(n)$ instead of up to $O(n*m)$, because there is no need to check the overlap for each primitive/child combination.

3.4.1 Loose Bounding Planes

To assure the traversal algorithm working correctly, further concepts need to be introduced. To know, which additional nodes must be traversed, each node gets two additional planes, the *loose bounding planes*. These planes determine how big the largest overlapping primitive of each node is.

Loose bounding planes need to be calculated while construction and saved per inner node. Theoretically, it can take place of the previous concept of *bounding planes*, by adding these to values to one. For the sake of simplicity, the concept will keep both planes.

When traversing a scene, loose bounding planes can be used to determine which nodes must be traversed in addition, by dividing it by the size of a child. This way, one will get the offset of additional nodes, which the biggest primitive is overlapping beside the current node. To do intersection tests with all necessary primitives, these offsets need to be added to the numerical value of $node_{entry}$ and $node_{exit}$, enlarging the range of traversed primitives.

Figure 3.6 visualizes this concept. The dotted lines are saved as loose bounding planes. Afterwards, the blue leaf nodes can be traversed correctly, leading to an intersection test of the green primitive. Without loose bounding planes, the primitive would not be hit, because its mid point lies in the bottom node, which would not get traversed without this extension.

Note that the amount of additional traversed nodes is increasing significantly, the bigger the loose bounding planes are. To get better traversal times, the loose bounding planes need to be at a lower value.

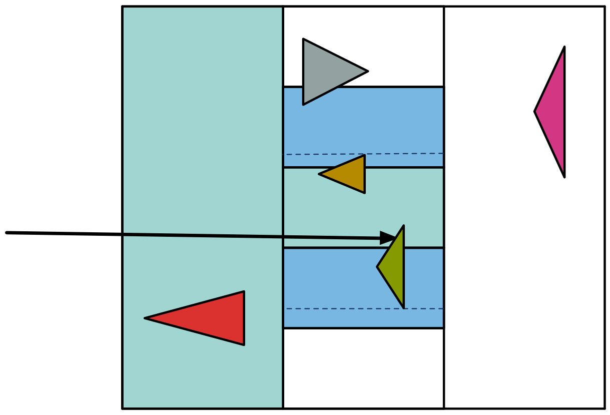


Figure 3.6: **Loose Bounding planes:** The yellow primitive is put into the turquoise leaf node in the middle, whereas the green primitive is put into the bottom blue node, both because of their mid points. The dotted lines are the extent of the biggest primitives overlapping the middle node, which are then saved as *loose bounding planes*. When traversing, the turquoise area is traversed like usual. When bringing in the loose bounding planes, the blue nodes will be traversed additionally. This ensures the green primitive being intersection-tested, even though it is not saved into the middle node.

3.4.2 Avoiding big primitives

There are certain tricks to keep the size of bounding planes at a low level. The first one was already introduced, by saving the loose bounding planes on each inner node instead of a more rare use of loose bounding planes. To save memory, loose bounding planes could be saved per depth, but this can have very significant impact on performance.

Big primitives, which get into deeper parts of the tree, nullify the purpose of the concept of traversal, because the loose bounds get big enough, the traversal algorithm would start to traverse every node at a certain depth of the tree.

To avoid this behavior, big primitives need get saved beforehand, keeping them nearer to the root node. This way, very big primitives will be intersection tested by almost each ray, which is unfortunate. But on the other hand, it will decrease the size of loose bounds drastically.

A way to save primitives early, is introduced in the concept of *inner node primitives*. This concept allows to save primitives next to other nodes in any inner node. To achieve this, each inner node gets a flag, if there are *inner node primitives*. If this flag is set, the construction method creates a *r+1*th node, for resolution *r*, which links to a leaf node with the same bounding box as the inner node. This way, the big primitives, which are over a size of choice bigger than the area of an inner node can be saved directly in the inner node instead of deeper into the tree.

In the traversal algorithm, the flag gets checked at the beginning of each node, and underlying primitives get tested if the flag is set.

Figure 3.7 shows a Loose Splitted Grid. The gray primitive is saved into the inner node, which results in much smaller loose bounding planes for the most left and most right nodes it overlaps.

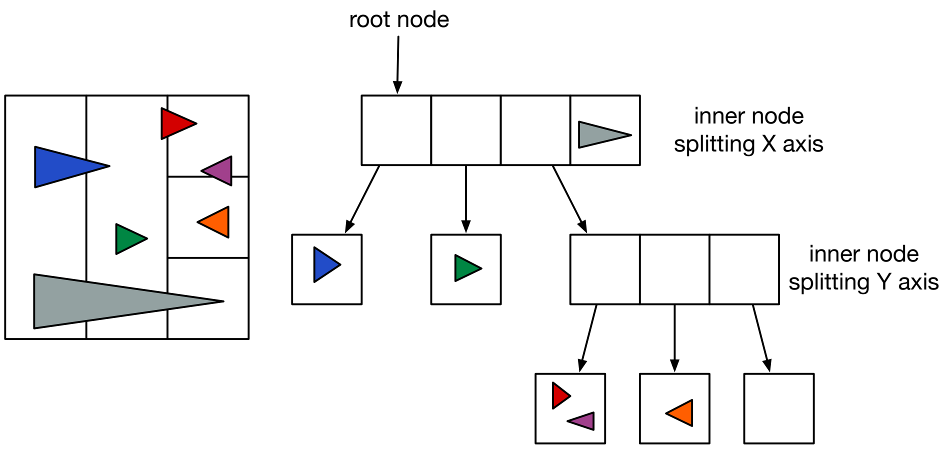


Figure 3.7: **Loose Splitted Grid:** In contrary to the Splitted Grid, in a loose variant, each primitive is put into the node, where its mid point lies. A concept called *loose bounding planes* will ensure that all primitives, the ray could hit, are intersection-tested correctly. To raise performance, big primitives, which would vastly increase the size of loose bounding planes can be saved directly in an inner node.

Chapter 4

Implementation

This chapter will annotate the implementation of both the Splitted Grid and Loose Splitted Grid. The Splitted Grid implementation will be explained in deep in section 4.1, whereas the Loose Splitted Grid is built upon the Splitted Grid in section 4.2. Therefore, the Loose variant uses the same base code, just with modifications and additions explained in its section.

While the original algorithm was implemented in C++, the following descriptions use a pseudo code similar to Python syntax, but with types at some points. This way, the code is short and clean, but does not lack information needed to understand the concept.

4.1 Splitted Grid

The Splitted Grid surrounding data structure saves its data in three arrays. The first one is a list of nodes. Each node in the resulting tree is saved in this array. The root node is the first element with its child directly after. Afterwards, childs-of-childs can be accessed by using offsets.

Each Splitted Grid node consists of two *unsigned int*. Depending on whether it is an inner node or leaf node, it saves information in different ways.

There are four conditions, a node can be in. It is a leaf node or an inner node splitting either of three axes in 3 dimensional space. This *axis/leaf* can be expressed in two bits, which are always stored in the upper two bits of the first integer. For an inner node, the next 30 bits store the offset to child nodes, whereas a leaf node stores the count of primitives it contains.

The second integer saves an indices offset for leaf nodes. This offset is used to access the primitives in traversal. In inner nodes, this second integer again serves purpose for two things. First, it saves the resolution of the current node in its first bits. In this implementation, the maximum resolution used is 16, so four bits suffice. The rest stores an offset to the boundary planes for this node. The amount of bits for saving resolution is arbitrary.

For higher resolutions, the amount of bits can be adjusted. This way of implementation could use up to 1,073,741,824 nodes and 4,294,967,296 primitives, which is more than sufficient for the tested scenes in Chapter 5.

To decrease the amount of bits needed for saving resolution, it is possible to save resolution as exponent of the power of two. This is a performance trade off though, as it loses accuracy determining resolution using surface area heuristics.

The second array is a list of primitive indices. As the Splitted Grid does not avoid duplicate references, the size of this array is usually quite bigger than the amount of primitives. The offset saved in leaf nodes is used to access this list.

The third and last array saves the boundary planes, used for clipping and stepping through the traversal. This concept was previously described in figure 3.2. There are two floats for each inner node, describing the planes on the axis which is currently used for splitting space. Note that, bounding planes are saved in an additional array to cut down two floats per leaf node, as the leaf node does not need bounding planes.

4.1.1 Construction

Figure 4.1 contains a full example of resulting pseudo-code for a normal build method. The code sample is pretty straight forward, as the construction is similar to other hierarchical construction methods used in raytracing. The *nodeAABB*, which is given as a parameter, is the parents bounding box. Starting with this, the bounding planes get calculated by calculating an primitive bounding box, which exactly fits the primitives in this node. Afterwards, a clipped bounding box is determined by shortening the *nodeAABB* on the splitting axis to primitive bounding box values. The clipped bounding box has the bounds, which are used for calculating further childs and therefore gets saved in bounding box array. The child bounding boxes can be calculated by dividing the clipped bounding box by resolution r, giving the exact size of a bounding box. Afterwards, the list of primitives is sorted to pass it to the next recursion step. If the maximum depth is exceeded or the number of primitives are lower than a certain threshold, a leaf node gets created.

The surface area heuristic variant is very similar. It does not save constants for resolution and does not round robin through the axes. Instead, in the beginning of each *subdivide()* call, it calculates the best resolution and axis for this particular node. A pseudo code for the surface area heuristic is given in figure 4.2. The two cost estimates for traversal cost and intersect cost are chosen beforehand. For acceleration structures like BVH-SAH or kD-SAH, intersect costs are often higher than traversal costs, resulting in a lower value for traversal cost. As we can use up to 16 childs per node in this implementation of the Splitted Grid comparing to two in BVH/kdTree, this

```
def subdivide(int[] indices, SGNode node, AABB nodeAABB, int& iNode,
     int& iIndex, int& iPlane, int depth, int axis):
  if c <= m_PrimitivesPerLeaf || d >= m_MaxDepth:
    createLeaf(indices, c, node, iIndex);
  else:
    int subIndex = iNode
    iNode += m_Resolution
    node = SplittedGridNode(iPlane, subIndex, a, m_resolution)

    // Exact bounding box
    AABB exact = ComputeExactBox(indices)
    AABB clipped = ClipBoundingBox(exact, nodeAABB)
    SaveBoundingPlanes(clipped, axis)

    float gridSubSize = clipped.Size(axis) / m_resolution

    // Calculating bounds of sub node
    foreach i in m_Resolution:
      AABB subCellBounds = CalculateChildBox(clipped, i, gridSubSize)
      int[] subIndices = CalculateOverlappingPrims(subCellBounds,
          indices)

      SGNode& sNode = m_Nodes[subIndex + i]
      subdivide(subIndices, sNode, subCellBounds, iNode, iIndex,
          iPlane, depth+1, (axis+1)%3)
```

Figure 4.1: Pseudo Code - Full implementation of construction of Splitted Grid. This form of implementation does not use the SAH.

is not the case anymore. Therefore, traversalCost and intersectCost are set to a similar value.

The SAH method evaluates each combination of axis and resolution with the cost estimate function presented in equation 3.1, getting the overall minimum cost estimate. The resulting *best* axis and resolution are then used in *subdivide()* code from figure 4.1.

To speed up construction times a little bit, the surface area heuristic got changed to only use resolutions of a power of two. The results were pretty similar, but with shorter construction times, due to fewer combinations to check for the heuristic.

4.1.2 Traversal

Figure 4.3 contains a full example of resulting pseudo-code for traversing a Splitted Grid. Note that the pseudo code is based on a recursive scheme, which is easier to understand, but usually slower than an iterative implementation. A secondary iterative algorithm has been implemented, which is

```
def findSplit(int[] indices, int count, AABB aabb, int& bestAxis,
    int& bestResolution):
  float bestCost = m_IntersectCost * count;
  float invSA = 1.0f / aabb.Area();

  for axis = 0 .. 3:
    for resolution = 0 .. m_MaxResolution:
      float gridCellSize = aabb.Size(axis) / resolution;
      float allCosts = 0.0f;

      for i = 0 .. resolution:
        bb = CaclulateChildBox(clipped, i, gridCellSize)
        num = CalculateOverlappingPrims(bb, i);
        allCosts += bb.Area() * num;

      float thisCost = resolution * m_TraversalCost + m_IntersectCost
          * allCosts * invSA;

      if thisCost < bestCost:
        bestCost   = thisCost;
        bestAxis   = axis;
        bestResolution = resolution;
```

Figure 4.2: Pseudo Code - Main part of Surface Area Heuristic

fairly straight forward. There are three parameters, which it carries along on the stack: A pointer to the current node as well as two floats for *tNear* and *tFar*.

The code first checks, if the current node is a leaf node and if so, does intersection tests. If not, the bounding planes are used to clip the ray and check, if its even hitting the node. If this is the case, the entry and exit nodes are calculated as presented in equation . Afterwards, the range of nodes in-between are recursively called. To save further calculations, the new t_{Near} and t_{Far} values get calculated and passed to the recursion step. For further performance optimization, the calculated values can be used to determine if the ray already hit a primitive. This is the case if $t_{Far} > t_{Hit}$ and leads to an early out.

4.2 Loose Splitted Grid

This section will modify the Splitted Grid to make it loose. Therefore, the changes from Splitted Grid to Loose Splitted Grid are explained, whereas everything else will reference to the previous section.

In a loose variant, additional loose boundaries for inner node needs to be saved. In this implementation, loose boundaries are saved in an additional list not overwriting existing list for shortened planes, because it is used in different ways in traversal. An additional optimization step would be to merge these two lists. To keep this implementation simpler, this is avoided.

Of course, both boundary plane lists have the same indices per node. Therefore, no additional offset needs to be saved in inner nodes.

There is a need for a new bit flag for inner nodes, whether they save primitives on their *r+1*-th child or not. This can be bit-shifted into either one of the available integers. In this implementation, the *resolution/planes* integer was used for this purpose.

The rest of acceleration data structure stays the way the Splitted Grid implementation does it.

4.2.1 Construction

In the construction algorithm, instead of looping through the primitives for each child node, determining if it overlays the child node, it can now project the mid-point of a primitive directly to a node, reducing the complexity of this part of construction.

To calculate the loose bounds, a new pre-procession step needs to be added though. In a simple implementation, as shown in figure 4.4, the primitives get looped through, and for the node its center is in, the size of overlap is calculated and saved. The loose bounding planes are the overall maximum of these values.

Achieving the *inner node primitives* is cheaper. In the same pre-procession step, one can check if a primitive is overly large. Is this the case, the primitive will be added to a *r+1*-th leaf node, and the flag will be set.

4.2.2 Traversal

Code changes for traversal are relatively small. First, the algorithm needs to check, if the *inner node primitives* flag is set, and do intersection tests accordingly. This part is implementable in a straight-forward manner, so there is no need for a code example.

The use of bounding planes differs though, because of newly introduced loose bounding planes.

The new loose bounding planes must be added at each calculation involving ray clipping, including the projection of the ray to $node_{entry}$ and $node_{exit}$. Therefore, the methods *GetDistanceTo(plane0, plane1)*, *CalculateEntryExitCells(ray, childsize, plane0)* and *DistanceToChildPlanes()* are changed accordingly.

```
def traverse(unsigned int node, Ray& ray, float tN, float tF):
  const SplittedGridNode* cur = &nodes[node]
  bool hitFound = false

  if current->IsLeaf():
    foreach(i in cur->GetCount())
      hitFound |= Intersect(ray, indices[cur->GetOffset() + i])
  else:
    int axis = cur->GetAxis()
    int sign = ray.d[axis] > 1 ? 1 : -1

    // Clip ray against the node's planes
    plane0, plane1 = GetBoundingPlanes(cur)
    pd0, pd1 = GetDistanceTo(plane0, plane1)

    float dN = max(tN, min(pd0, pd1))
    float dF = min(tF, max(pd0, pd1))

    if dN > dF:
      return false

    // Compute entry and exit cell
    float childSize = (plane1 - plane0) / cur->GetResolution()
    entry, exit = CalculateEntryExitCells(ray, childSize, plane0)

    // Recursively call childs
    foreach i in entry..exit:
      cd0, cd1 = DistanceToChildPlanes()

      // If missed, the following can be skipped too.
      float minN = min(cd0, cd1)
      if minN > dF:
        break

      float cN = max(dN, minN)
      float cF = min(dF, max(cd0, cd1))

      // Traverse the child node recursively and update dF
      hitFound |= traverse(cur->GetChild(i), ray, cN, cF, step)
      dF = min(dF, ray.t)
  return hitFound;
```

Figure 4.3: Pseudo Code - Full implementation of traversal of Splitted Grid

```
foreach i = 0 .. count:
  AABB primitiveAABB = getPrimitiveAABB(i)
  float center = primitiveAABB.center()

  foreach cell = 0 .. resolution:
    if bb[cell].Inside(axis, center):
      primitiveMin, primitiveMax = ExtentOfOverlapping(bb[cell],
          primitiveAABB, axis)

      overallMin[cell] = max(overallMin[cell], primitiveMin);
      overallMin[cell] = max(overallMax[cell], primitiveMax);
```

Figure 4.4: **Pseudo Code - Changes for constructing a Loose Splitted Grid**: This pre-procession step will calculate loose bounds for all child nodes in current inner node.

Chapter 5

Results

The performance, both traversal and construction time, as well as memory usage of the *Splitted Grid* and the *Loose Splitted Grid* were measured for the CPU. Both acceleration structures need to get evaluated and compared to other state-of-the-art approaches, to check its necessity in real world applications. For purpose of these statistics, a variety of test scenes were chosen, to give a clear look, how the approaches perform in differently structured scenes.

Section 5.1 will feature a description on how tested scenes are designed. Section 5.2 first evaluates the parameters used for the Splitted Grid and Loose Splitted Grid and then compares these methods to other approaches. Afterwards, previous results and additional tests are used to give an estimated first guess on GPU performance in section 5.2.3.

5.1 Test environment

There are different kind of scenes, which need to be tested, as there are different kind of real life scenarios. Objects in scenes can be small and very detailed. In a video game scenario, these would be typically used for humans or important objects, which are in the direct view of the player. Other objects in the background or far away from playable area might be lower resoluted, which often results in very big primitives for floors or walls. In an open world or architecture, there might be big houses which small objects inside, where a combination of problems come to mind: A very big scene, where the size of primitives varies largely, as well as big parts of empty space (*tea-pot-in-the-stadium*).

To get a reliable guess, on how well an acceleration structure performs in random, real world scenarios, a variety of scenes covering different aspects of problems need to get tested and compared to other approaches. The scenes used for analysis have been chosen with this in mind. In the following, tested scenes are further explained:

- *Scene 1 - Head*: Head is a quite simple scene with around 17684 smaller sized primitives which are even distributed. There are two point light sources in *Head* scene.
- *Scene 2 - Sponza*: The popular Sponza scene features a lot of unevenly distributed primitives, as well as big differences of primitives sizes. There are primitives at the size of the whole scene, as well as very small ones. The tested version has 279163 primitives. Sponza is an architecture scene. There is one point light source in *Sponza*.
- *Scene 3 - Sibenik*: Cathedral Sibenik is an architecture scene having 76521 primitives, which similar to Scene 2. It features very big primitives, which are usually a problem for hierarchies based on splitting room. There is one point light source in *Sibenik*.
- *Scene 4 - Fairy Forest*: Fairy Forest is a scene of 172669 primitives, which features a teapot-in-the-stadium problem, representing high resoluted objects in an open area. This scene uses two light sources, one being a normal point light source, while the other being an infinite light using an environment map.
- *Scene 5 - Dragon*: Stanford Dragon is a very popular scene. Featuring 7219045 primitives, it serves the purpose of showing a very high resoluted object from low distance. The chosen scene uses three point light sources.

5.2 Test results

All statistics were measured on a system with an Intel Core i7-3720QM with 2.60 GHz per core and 16 GB DDR3-RAM running Arch Linux (Feb 2013, 64-bit). The processor features 4 physical cores and Hyper-Threading, resulting in an usage of 8 threads for multi-threaded parts. In the current implementation, the construction method is not capable of multi-threading, so only traversal benefits from it. The code is implemented in C++ and an in-house implemented ray tracing framework called *JUNO*.

A resolution of 1024 x 768 pixels was used for all scenes. For ray casting, one sample per pixel was used. For path tracing, a maximum count of four bounces as well as a maximum of four samples per pixel were used.

5.2.1 Evaluating parameters

In both new approaches, there are a few parameters, which must be chosen for each scene. Keeping in mind that a Splitted Grid is no binary tree, but having an arbitrary resolution, usual parameters used in kD-Tree and BVHs do not work efficiently.

In uniform construction methods, the resolution and maximum depth of the tree are set beforehand. In all internal tests while development, the order of splitting axes in a round robin manner was the most efficient approach, setting this the standard method for an uniform construction of both a Splitted Grid and a Loose Splitted Grid.

For a Splitted Grid, the uniform construction method tend to get a very high memory usage as trade off for a good traversal time, which is the result of default values for resolution in scenes having a tea-pot-in-the-stadium problem. Especially with keeping in mind that Splitted Grid does not avoid duplicated references, high resolutions in deeper depths or higher resoluted parts of the scene produce a huge memory footprint. A high depth causes the same problem, which limits the maximum depth of this acceleration structure. This problem is avoided in most cases by using the SAH method.

In almost all tested scenes, a resolution between 6 to 10 and a depth of 6-12 has been proven the best. After determining efficient parameters in a SAH construction, examining the resulting average resolution built by SAH can also hint parameters for approximately good results at uniform construction. Therefore, additional effort were put into determining parameters for SAH construction.

Using a SAH construction algorithm, both axis order and resolution are calculated heuristically for each node. Therefore, in a SAH construction, only maximum depth is set by the user. Theoretically, the limits of resolution, a SAH algorithm can choose, are also free parameters for the user to decide. As it obviously makes no sense to create one children on an inner node, because the inner node could then be a leaf node itself, the lower limit of resolution is always set to 2.

The higher limit for a Splitted Grid was set to 16, as it was the last step where noticeable differences in tree construction took place. Higher values resulted in the same tree, which means the SAH never used these values. Lower values caused a decrease in performance.

For a Loose Splitted Grid SAH, the higher value was set to 2, resulting in a default resolution of 2 for the whole tree. The SAH construction still performs better, because of the adaptive choice for splitting axes. The reason, because the resolution was set to this low value, is the loose design of a Loose Splitted Grid. When having loose borders, the traversal algorithm needs to check additional nodes on each recursion step. When having a higher number of resolution, the size of sub-trees which get traversed additionally increases exponentially, resulting in an explosion of traversal time.

After the Loose Splitted Grid now being a binary tree, which has no duplicated references per design choice, the depth can be set to a much higher value, not causing high memory footprints. In tests, the Loose Splitted Grid could get a depth of over 20 without issues in memory or performance.

Again, in case of a uniform Loose Splitted Grid construction method, the values of a SAH build were a good hint, on how to select parameters.

In evaluating parameter choices for the SAH, figure 5.1 shows results for Splitted Grids, whereas figure 5.2 shows it for Loose Splitted Grids. The traversal time is based on a ray casting analysis, using one sample per pixel.

As previously expected, the depth increases memory usage of a Splitted Grid in a linear way. While the traversal time usually gets better with a higher depths, the construction time also takes longer. This behavior is expected, as the higher depth causes more recursion steps for construction and therefore more calls for the SAH method.

The Loose Splitted Grid scales in a same behavior for performance times, but having fewer scaling issues with memory usage, as there are no duplicated references.

Notice that, there is a trade off between those three values, depending on how one chooses the parameters. For the next subsection, values with priority on very low traversal time while keeping memory usage and construction time on a relatively low level are used.

5.2.2 Comparison with different acceleration structures

For comparison with bounding volume hierarchies, an implementation of a BVH using surface area heuristic and a kD-Tree using spatial median split were used. Note that a SAH kD-Tree would be a minor performance advantage against a SMS constructed tree, but the given framework had no working implementation of this construction variant, so a SMS tree was used. The BVH implementation uses 32 bytes per node, which is explained in detail in section 5.2.3. The kD-Tree implementation saves 8 bytes per node.

SG and LSG reference the Uniform construction method, whereas SG-SAH and LSG-SAH reference the construction using surface area heuristics.

Figure 5.3 until 5.7 show analyses for tested scenes. In the detailed overview *(c)* of each figure, it shows absolute values of the results. The row *trav1* reflects the results for ray casting the scene. This test, using no shadow rays and textures, provides a good example on how good the acceleration structures perform for primary rays. The second row listing *trav2* shows results for path tracing the scene. The construction time is listed as *constr* and *mem* refers to memory consumption. Furthermore, the row *param* lists used parameters, where r stands for *resolution* and d stands for *maximum depth.* In cases of SAH construction for Splitted and Loose Splitted Grid, *resolution* means a maximum resolution the SAH algorithm may use. The parameter for *minimum primitives per leaf node* was always set to 4.

Evaluating the figures, there are some things noticeable:

- A SAH Splitted Grid behaves similarly in most of the scenes. In ray casting runthroughs, it usually outperforms BVH, and in some cases even a kD-Tree. Note, that the kD-Tree implementation uses a SMS approach, so a SAH-kD-Tree might still be a little bit faster.

- An Uniform Splitted Grid has a high memory footprint, highly depending on the choice of parameters. When testing around with different parameters, it turns out that an Uniform Splitted Grid can outperform a kD-Tree in any case, but often has a memory usage at an extremely high level. The high memory usage is due to the default resolution in uneven distributed scenes, where high resolution deeper in the tree cause an overload of duplicated references.

- The SAH variant of the Splitted Grid is not necessarily faster than the Uniform variant, but usually serves a much better fit scene and therefore a lower memory footprint. A higher maximum depth would often cause better traversal performances for the SAH Splitted Grid, but then increasing the amount of memory used, as explained in Section 5.2.1.

- While the Loose Splitted Grid is usually much slower, the relative performance compared to competitors, varies. In the Sibenik scene, the LSG has a traversal time up to 7 times higher than its competitors. In the Dragon scene, it is even quicker than an Uniform SG. When comparing the scenes, it seems that the current implementation of the Loose Splitted Grid has severe problems traversing scenes with a teapot-in-the-stadium (Scene 2 and 3) problem. On even distributed scenes, it performs better (Scene 1 and 5).

- In both architectural scenes, the SAH Loose Splitted Grid seems to be inferior to the Uniform construction method in a level, that does not seem right. This indicates some issues in the design of the LSG methods, which have not been addressed yet.

- The Loose Splitted Grid has great memory usage in comparison to other approaches.

- In Dragon scene, which has the biggest amount of primitives by far, the kD-Tree is slower than the BVH-Tree. This is due to the fact, the kD-Tree does not use a SAH construction algorithm.

As a result of the fairly poor results of the Loose Splitted Grid in some tested scenes, the surface area of resulting trees has been analyzed, as well as some statistics on how the trees are traversed have been created. As presented in figure 5.8, the results differ vastly between the scenes. Mostly, the Loose Splitted Grid is slower, because a lot additional sub-trees are traversed due to the loose boundary planes. In case of *Sponza* and *Sibenik* scene, this behavior creates a vast amount of primitive intersection tests in addition, which result in the very slow performances shown in figure 5.4 and 5.5.

5.2.3 Estimate of GPU performances

After previous analyses, which are all based upon a CPU implementation of the algorithm, the performance and therefore value of this technique on graphic cards may differ largely. A GPU implementation is not part of this thesis, leaving room for speculation. Usually, the memory throughput and with this stack accesses are a first clue, on how an algorithm performs on a GPU. This is caused by the way a GPU is designed.

Figure 5.9 shows a brief comparison of these indicators with a BVH. The BVH is currently the most used acceleration structure in GPU and real time applications, because of its fast construction time and predictable memory usage while having a fast traversal method.

In the Splitted Grid, the amount of memory per stack entry is 12 bytes, 4 bytes for the node index as well as two floats for t_{Near} and t_{Far}. The node itself has 8 bytes as mentioned previously. Additionally, on each inner node traversal, 8 bytes for two floats from the planes array are accessed. In the Loose Splitted Grid, the amounts are the same. But on each inner node, it accesses 16 bytes, because of the loose bounding planes in addition. In a BVH, the stack only saves a pointer to the node, which can also be the index of the node, so 4 bytes. The node is 32 bytes, consisting of a bounding box which has 6 planes, so 6 floats, as well as two integers. The stack memory throughput is simply the amount of stack access, multiplied by the size of a stack entry. The node memory throughput is the memory, is the node itself plus linked memory, which is accessed while traversing inner nodes, like bounding planes, multiplied by the number of inner node traversals from figure 5.8.

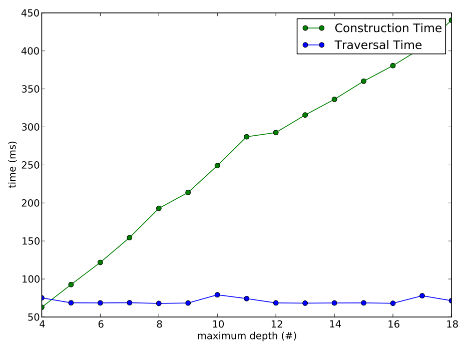
The results are similar to those in previous subsection. In scene 2 and 3, the Loose Splitted Grid has a very high amount of memory throughput, which is predictable when seeing the amount of inner nodes traversed in figure 5.8. The Splitted Grid has a relatively small memory throughput. This is due to the early out, which often decreases the amount of stack pops. This makes it an acceleration structure worthy testing out in a GPU implementation.

When comparing stack and node memory usages, the Splitted Grid has a very good memory throughput compared to BVH, in terms of node memory accesses. The memory throughput on the stack is higher though.

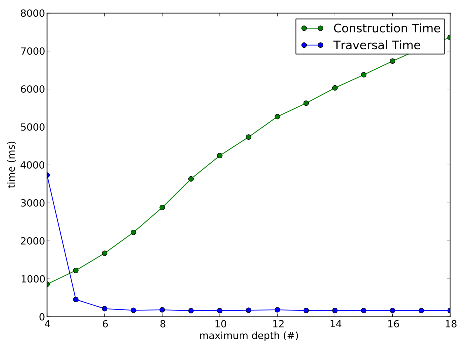
Keep in mind when comparing this, that the BVH uses standard traversing implementation as popular in CPU raytracers, and no GPU-oriented stack-less method as presented [HDW$^+$11].

Note, that there is the possibility of decreasing the amount of memory throughput and stack push/pops in the Splitted Grid by changing the traversal algorithm. The values t_{Near} and t_{Far} are calculated on each level and put on stack. It might be faster on the GPU, to dump them from stack and therefore calculating them on-the-fly more often, with some small modifications.

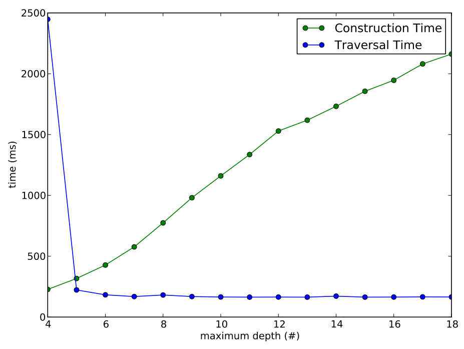
Another type of traversal algorithm could save intervals of entry/exit nodes by saving offset and counts to the stack, instead of pushing each node individually to the stack. This variant could vastly decrease the amount of stack pushes/pops.



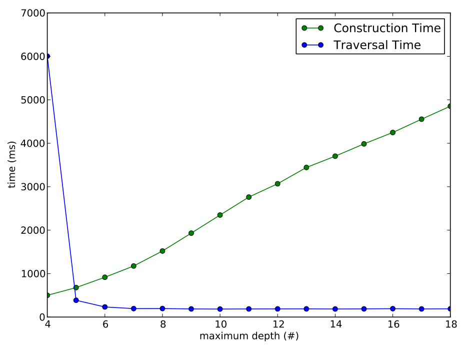
(a) Scene 1: Head



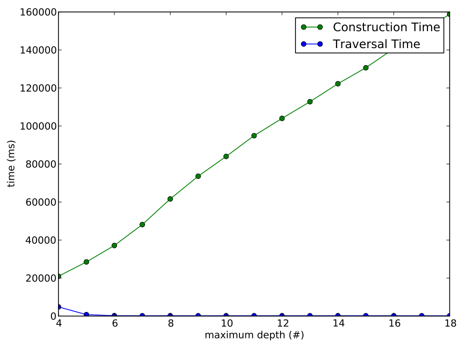
(b) Scene 2: Sponza



(c) Scene 3: Sibenik



(d) Scene 4: Fairy Forest

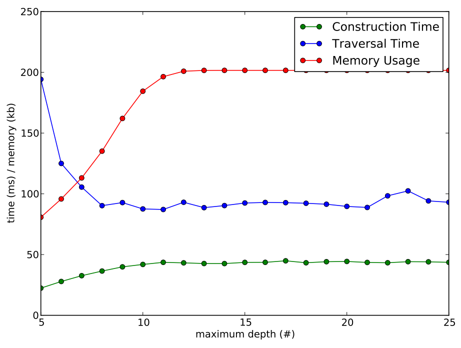


(e) Scene 5: Dragon

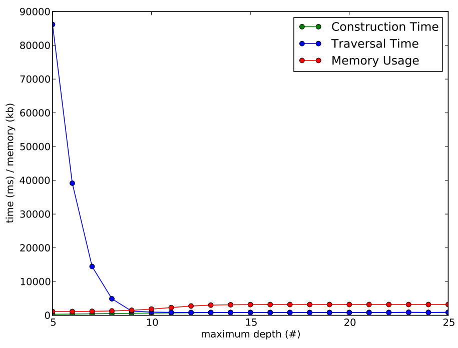
Scene	mem *depth=4*	mem *depth=11*	mem *depth=18*
Head	273kb	2276kb	4109kb
Sponza	2300kb	37709kb	65835kb
Sibenik	627kb	11237kb	20104kb
Fairy	1240kb	21234kb	44189kb
Dragon	37336kb	705242kb	>1500000kb

(f) Memory Usage

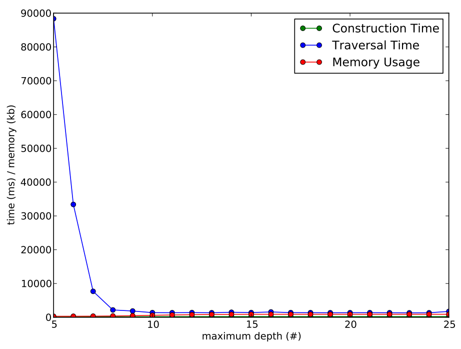
Figure 5.1: **Splitted Grid SAH Parameter Analysis:** As the memory usage scales almost linearly, only three example values are given.



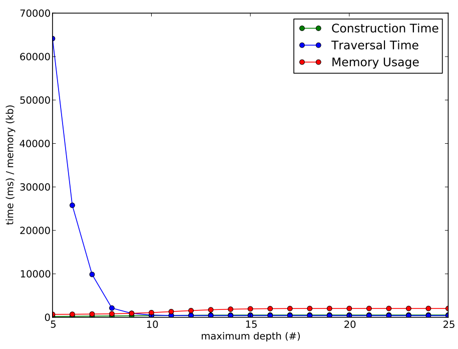
(a) Scene 1: Head



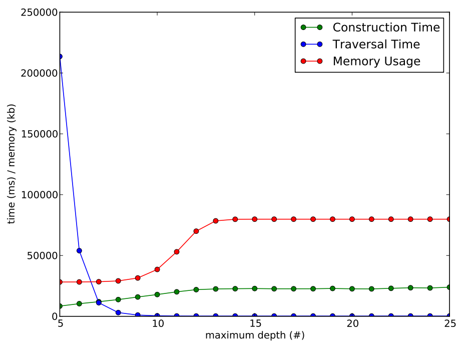
(b) Scene 2: Sponza



(c) Scene 3: Sibenik

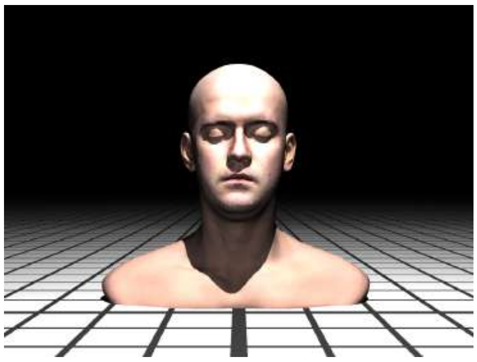


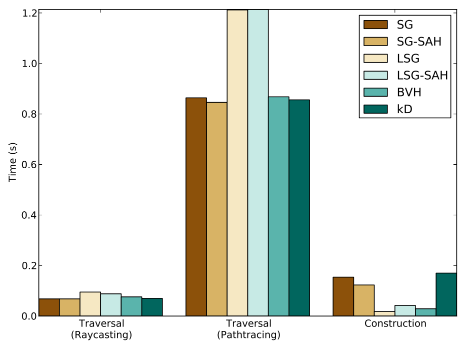
(d) Scene 4: Fairy Forest



(e) Scene 5: Dragon

Figure 5.2: **Loose Splitted Grid SAH Parameter Analysis:** In most cases, all results above a depth over 13-18 are the same, because in each case the same tree is built. Variations in the time are due to an error of measurement and natural variations in computing time. As far as traversal time goes, the higher maximum depth, the lower is resulting traversal time.


(a) Tested scene, *n=17684*

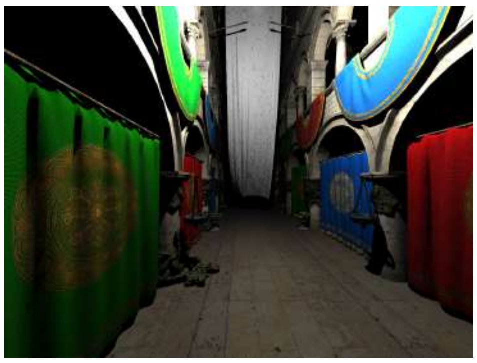


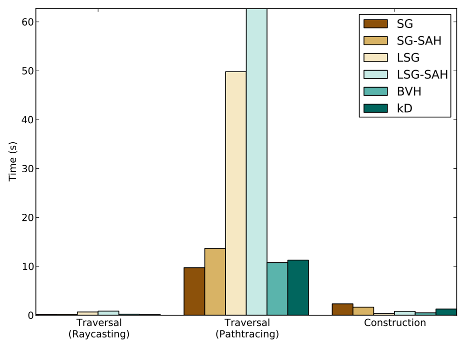
(b) Speed comparison

Head	SG	SG-SAH	LSG	LSG-SAH	BVH	KD
trav1	0.068s	0.068s	0.095s	0.088s	0.076s	0.070s
trav2	0.864s	0.846s	1.212s	1.214s	0.868s	0.856s
constr	0.154s	0.123s	0.018s	0.042s	0.029s	0.170s
mem	5086kb	779kb	221kb	201kb	421kb	1592kb
param	r = 8 d = 10	r = 16 d = 6	r = 4 d = 25	r = 4 d = 25	d = 25	d = 25

(c) Detailed overview of results

Figure 5.3: Analysis for scene 1 - Head Scene


(a) Tested scene, *n=279163*



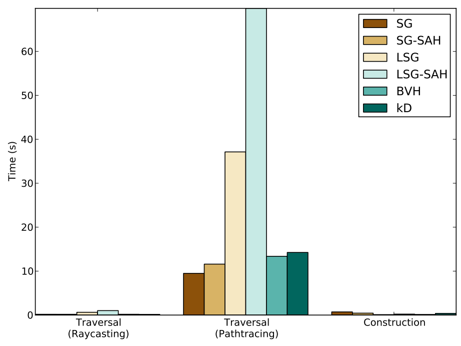
(b) Speed comparison

Sponza	SG	SG-SAH	LSG	LSG-SAH	BVH	KD
trav1	0.186s	0.187s	0.675s	0.848s	0.214s	0.166s
trav2	9.734s	13.681s	49.812s	62.715s	10.788s	11.277s
constr	2.339s	1.653s	0.364s	0.812s	0.510s	1.276s
mem	64955kb	6916kb	4183kb	3198kb	6341kb	12568kb
param	r = 8 d = 10	r = 16 d = 6	r = 2 d = 25	r = 4 d = 25	d = 25	d = 25

(c) Detailed overview of results

Figure 5.4: Analysis for scene 2 - Sponza Scene

(a) Tested scene, *n=76521*

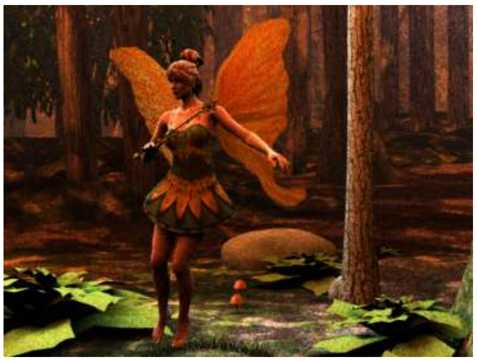


(b) Speed comparison

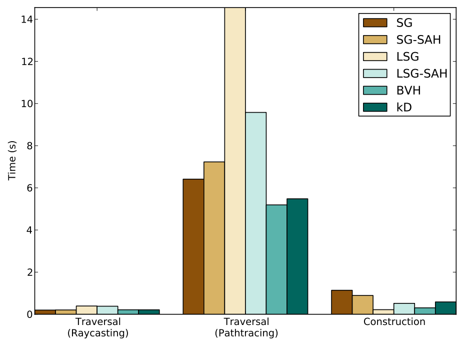
Sibenik	SG	SG-SAH	LSG	LSG-SAH	BVH	KD
trav1	0.177s	0.173s	0.631s	1.015s	0.179s	0.149s
trav2	9.488s	11.594s	37.129s	69.796s	13.372s	14.252s
constr	0.719s	0.437s	0.096s	0.215s	0.135s	0.375s
mem	21229kb	2041kb	1133kb	897kb	1827kb	3018kb
param	r = 8 d = 10	r = 16 d = 6	r = 2 d = 25	r = 4 d = 25	d = 25	d = 25

(c) Detailed overview of results

Figure 5.5: Analysis for scene 3 - Sibenik Scene



(a) Tested scene, *n=172669*

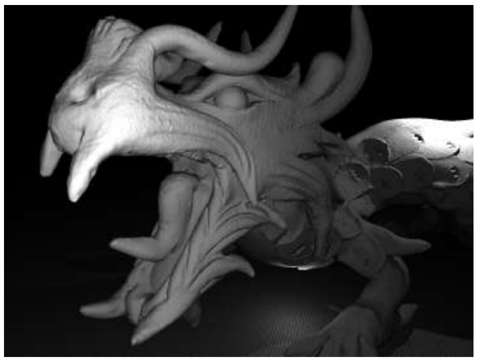


(b) Speed comparison

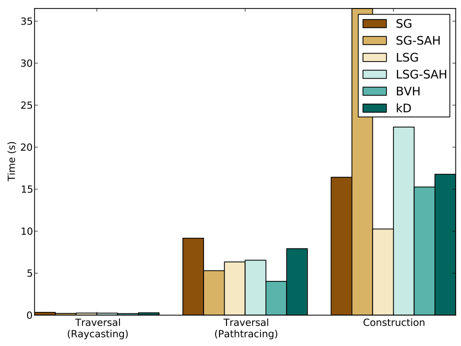
Fairy	SG	SG-SAH	LSG	LSG-SAH	BVH	KD
trav1	0.200s	0.206s	0.393s	0.383s	0.214s	0.213s
trav2	6.413s	7.230s	14.552s	9.578s	5.190s	5.480s
constr	1.140s	0.893s	0.219s	0.519s	0.307s	0.587s
mem	27107kb	3525kb	2183kb	2031kb	4020kb	3752kb
param	r = 8 d = 10	r = 16 d = 6	r = 4 d = 25	r = 4 d = 25	d = 25	d = 25

(c) Detailed overview of results

Figure 5.6: Analysis for scene 4 - Fairy Forest Scene



(a) Tested scene, *n=7219045*



(b) Speed comparison

Dragon	SG	SG-SAH	LSG	LSG-SAH	BVH	KD
trav1	0.339s	0.213s	0.252s	0.244s	0.189s	0.288s
trav2	9.163s	5.300s	6.335s	6.546s	4.027s	7.920s
constr	16.419s	36.507s	10.257s	22.394s	15.256s	16.774s
mem	88925kb	118660kb	89093kb	79855kb	172102kb	53233kb
param	r = 8 d = 7	r = 16 d = 6	r = 4 d = 25	r = 4 d = 25	d = 25	d = 25

(c) Detailed overview of results

Figure 5.7: Analysis for scene 5 - Dragon Scene

	SG	SG-SAH	LSG	LSG-SAH
		Head		
sa	2,682	2,582	2,309	2,115
pit	1,256,137	1,554,753	3,674,168	4,083,506
tin	2,450,626	2,396,075	4,850,796	4,903,404
tln	85,934	52,751	701,910	374,400
		Sponza		
sa	2,502,832,128	1,884,039,552	3,608,101,120	2,301,579,008
pit	7,148,465	16,816,158	55,767,103	137,306,763
tin	30,844,931	18,765,122	121,796,021	80,391,290
tln	4,511,333	6,165,578	13,622,717	28,324,701
		Sibenik		
sa	77,896	58,381	108,004	93,804
pit	7,408,410	11,329,798	56,683,329	151,301,282
tin	21,771,987	25,467,874	106,762,892	116,277,187
tln	4,009,849	3,931,449	17,754,913	49,498,327
		Fairy Forest		
sa	5,731	3,436	3,960	4,253
pit	11,334,281	17,943,501	384,159,544	38,377,664
tin	28,164,253	30,136,692	55,789,704	52,270,831
tln	10,566,321	4,612,806	15,070,382	15,234,843
		Dragon		
sa	3,910,959	4,867,261	4,805,505	4,596,750
pit	59,099,747	19,836,105	10,710,582	10,512,202
tin	18,462,513	20,300,633	30,761,915	30,392,431
tln	2,550,212	3,198,146	5,524,866	4,071,065

Figure 5.8: **Traversal comparison**: This figure shows the size of surface area for each given scene-acceleration structure combination as *sa*. The row *pit* shows *primitive intersection tests* whereas *tin* and *tln* stands for *traversed inner nodes* and *traversed leaf nodes* These results can be an indicator on how well the tree is built and reflect performance issues. The parameter used for tests where the same as in previous analyses. For this analysis, the ray casting approach was used.

	SG	SG-SAH	LSG	LSG-SAH	BVH
Head					
push	3,716,016	3,397,572	7,886,493	7,561,195	4,249,241
pop	2,014,812	1,799,482	5,228,711	4,835,030	4,249,241
stack mem	65.58mb	59.48mb	150.09mb	141.86mb	32.42mb
node mem	56.75mb	55.24mb	190.40mb	189.91mb	143.01mb
Sponza					
push	40,116,784	30,600,991	141,342,441	112,668,664	37,851,327
pop	20,391,289	18,697,816	70,213,106	73,521,058	37,851,327
stack mem	692.46mb	564.18mb	2421.06mb	2130.77mb	288.78mb
node mem	740.40mb	476.54mb	4750.08mb	3282.79mb	1225.62mb
Sibenik					
push	30,364,763	34,311,111	131,615,155	168,300,563	27,459,098
pop	16,295,460	22,591,815	64,894,307	115,881,741	27,459,098
stack mem	533.98mb	651.20mb	2248.87mb	3252.21mb	209.50mb
node mem	528.91mb	612.91mb	4208.14mb	4813.26mb	901.68mb
Fairy Forest					
push	41,276,918	37,782,311	81,520,006	79,526,630	35,022,499
pop	22,164,962	23,613,104	56,139,486	52,620,765	35,022,499
stack mem	726.03mb	702.61mb	1575.39mb	1512.31mb	267.20mb
node mem	725.24mb	724.97mb	2243.19mb	2110.21mb	1148.96mb
Dragon					
push	24,004,047	27,985,104	47,836,258	45,899,883	22,821,350
pop	14,492,154	18,751,708	31,671,835	29,835,165	22,821,350
stack mem	440.55mb	534.87mb	909.90mb	866.72mb	174.11mb
node mem	442.03mb	489.04mb	1215.63mb	1190.44mb	739.84mb

Figure 5.9: **GPU comparison**: This figure shows a comparison of stack accesses (amount of pops and pushes), written as *pop / push*, as well as the memory throughput for stack and nodes, which is called *stack mem / node mem*.

Chapter 6

Conclusion and Discussion

In summary, the analysis in Chapter 5 showed capabilities of the Splitted Grid. For ray casting, the traversal speeds are on same or better level than common techniques like a BVH or kD-Tree. For pathtracing, the performance differs, the Splitted Grid competes overall well against it's competitors, with some minor flaws. Splitted Grid can not compete against a BVH in construction time. However, its SAH method is in many cases on a similar level then our kD-Tree implementation, while having a slight advantage in memory usage.

An important issue to work on is the traversal speed of the Loose Splitted Grid. Unfortunately, the Loose variant, while having a very good average memory consumption and construction time compared to other approaches tested, could not compete with other methods in traversal speed. While the current construction methods are very fast, the traversal speed performs poorly. In future research, new concepts to achieve better traversal performance for the Loose Splitted Grids need to be developed for a performance, which is expected to achieve.

For the Splitted Grid, the current traversal is heavily optimized. The iterative implementation saves *tNear*, *tFar* and the node index on the stack. While this is expected, the number of stack accesses might be higher than it needs to be. Especially for an use of the Splitted Grid in a GPU raytracer, it is possible to reduce the amount of stack accesses by instead saving intervals of $(entryNode, count)$. Additionally, *tNear* and *tFar* could get calculated on use, which reduces the amount of variables on the stack, therefore reducing memory throughput further. In this scenario, a trade-off between stack usage and performance needs to be evaluated.

The current construction method is fairly simple, giving space for more advanced construction concepts. In current implementation, the algorithm is not capable of multi-threading, which could be a first feature to add - especially with GPU computing in mind.

As the usage of real time raytracers gets popular, the analysis in Sec.

5.2.3 gave a first hint in how well the acceleration structures may perform on a GPU, which has naturally a different kind of data procession and therefore differs in the way, the algorithm gets computed. In the first priority, a GPU implementation of the algorithm needs to be implemented and evaluated.

Furthermore, as it is possible to implement a loose variant of a Splitted Grid, there is an option to create an implicit variant of the algorithm. A similar approach, which was done in [EBM12], created a similar designed variant of a BVH using no additional memory for an acceleration structure by sorting primitive indices in a manner, so that a temporary BVH tree can be calculated on traversal time at any point. When using the uniform construction algorithm for the Splitted Grid, the resolution as well as splitting axis for each point is predictable. Having no duplicated references, the design of Loose Splitted Grid satisfies all criteria needed for an implicit approach built upon.

Bibliography

[Áfr12] Attila T. Áfra. Incoherent ray tracing without acceleration structures. In *Eurographics (Short Papers)*, pages 97–100, 2012.

[AW87] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *In Eurographics '87*, pages 3–10, 1987.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[EBM12] Martin Eisemann, Pablo Bauszat, and Marcus Magnor. Implicit object space partitioning: The no-memory BVH. Technical Report 16, Computer Graphics Lab, TU Braunschweig, January 2012.

[FTI86] A. Fujimoto, T. Tanaka, and K. Iwata. Arts: Accelerated raytracing system. *Computer Graphics and Applications, IEEE*, 6(4):16–26, 1986.

[Gla89] A.S. Glassner. *An Introduction to Ray Tracing.* Morgan Kaufmann Series in Computer Graphics and Geometric Modeling. Acad. Press, 1989.

[HDW$^+$11] Michal Hapala, Tomas Davidovic, Ingo Wald, Vlastimil Havran, and Philipp Slusallek. Efficient stack-less bvh traversal for ray tracing. In *27th Spring Conference on Computer Graphics (SCCG 2011)*, 2011.

[JW89] D. Jevans and B. Wyvill. Adaptive voxel subdivision for ray tracing. In *Graphics Interface*, pages 164–172, 1989.

[Kaj86] James T. Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM.

[KBS11] Javor Kalojanov, Markus Billeter, and Philipp Slusallek. Two-Level Grids for Ray Tracing on GPUs. In Oliver Deussen Min Chen, editor, *EG 2011 - Full Papers*, pages 307–314, Llandudno, UK, 2011. Eurographics Association.

[KK86] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. *SIGGRAPH Comput. Graph.*, 20(4):269–278, August 1986.

[KS97] Krzysztof S. Klimaszewski and Thomas W. Sederberg. Faster ray tracing using adaptive grids. *IEEE Comput. Graph. Appl.*, 17(1):42–51, January 1997.

[KS09] Javor Kalojanov and Philipp Slusallek. A parallel algorithm for construction of uniform grids. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 23–28, New York, NY, USA, 2009. ACM.

[Laf96] Eric Lafortune. Mathematical models and monte carlo algorithms for physically based rendering. Technical report, 1996.

[NS04] Beomseok Nam and A. Sussman. A comparative study of spatial indexing techniques for multidimensional scientific datasets. In *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, pages 171–180, 2004.

[PGSS07] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3):415–424, September 2007. (Proceedings of Eurographics).

[PH10] M. Pharr and G. Humphreys. *Physically Based Rendering: From Theory To Implementation.* Morgan Kaufmann. Morgan Kaufmann/Elsevier, 2010.

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975.

[RW80] Steven M. Rubin and Turner Whitted. A 3-dimensional representation for fast rendering of complex scenes. *SIGGRAPH Comput. Graph.*, 14(3):110–116, July 1980.

[Wal04] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination.* PhD thesis, Computer Graphics Group, Saarland University, 2004.

[Wal07] Ingo Wald. On fast Construction of SAH based Bounding Volume Hierarchies. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing*, 2007.

[WBS07] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1), 2007.

[Whi80] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, June 1980.

[WIP08] Ingo Wald, Thiago Ize, and Steven G Parker. Fast, Parallel, and Asynchronous Construction of BVHs for Ray Tracing Animated Scenes. *Computers and Graphics*, 32(1):3–13, 2008.

[WSC$^+$95] Kyu-Young Whang, Ju-Won Song, Ji-Woong Chang, Ji-Yun Kim, Wan-Sup Cho, Chong-Mok Park, and Il-Yeol Song. Octree-r: An adaptive octree for efficient ray tracing. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):343–349, December 1995.

[Zac02] Gabriel Zachmann. Minimal hierarchical collision detection. In *Proc. ACM Symposium on Virtual Reality Software and Technology (VRST)*, pages 121–128, Hong Kong, China, November11–13 2002.