Utrecht University

Improvements for Extended Morton Codes for Bounding Volume Hierarchy Construction on GPUs in Problematicly Large Scenes

Author:
Sietze N. Riemersma

Supervisors:
Prof. dr. Alexandru C. Telea
Dr. Frank Staals

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Department of Information and Computing Sciences

November 2024

Utrecht University

Abstract

Department of Information and Computing Sciences

Master of Science

Improvements for Extended Morton Codes for Bounding Volume Hierarchy Construction on GPUs in Problematicly Large Scenes

by Sietze N. Riemersma

We present three new methods to enhance the positional accuracy of Morton Codes in large scenes that have distant or large primitives, which is an essential aspect of many Bounding Volume Hierarchy (BVH) construction algorithms on the GPU. Two methods will use an additional occupation grid to remove unnecessary bits and move splits if considered more beneficial to combat the issue of distant primitives. The other method updates the computation of the scene bounds for scenes with large primitives at the edge of the scene. We show by building linear BVHs on the GPU for different scenes with the improved Morton Codes that there is room for improvement for both distant and large primitives in a scene. The new scene bounds computation shows a performance improvement of 0.41% - 0.93% in two games, a small but significant improvement according to the stakeholder (AMD). However, the other two methods show a potential increase in tracing performance in cases with distant primitives, but it takes too much time to create the occupation grid.

Contents

1 Introduction

Simulating light realistically in artificial 3D scenes is a crucial problem, as realistic lighting enhances the immersion of the viewer watching the artificial content. *Ray tracing* has been a method at the forefront of this problem, as this method traces light rays through a scene to simulate shadows, reflections, and refractions, which similar to how light works in the real world. Until recently, ray tracing seemed impossible in *real-time applications* and was only feasible for *offline applications* with more time to render a frame. However, with improvements to ray tracing due to research and more specialized hardware from *GPU* vendors, ray tracing is now possible in real-time applications such as games.

Ray tracing is a technique that involves tracing rays from a *camera* through a *scene*, with the rays querying the scene to identify the nearest intersections with the scene's *primitives*, such as triangles, quads, or full objects, to determine a color for each pixel in an image. Extending the camera rays with additional rays, such as shadow rays, can further enhance the scene's realism. Figure 1 shows a visualization of this process, and Figure 2 shows a visual comparison of Cyberpunk [1] with ray tracing enabled/disabled.

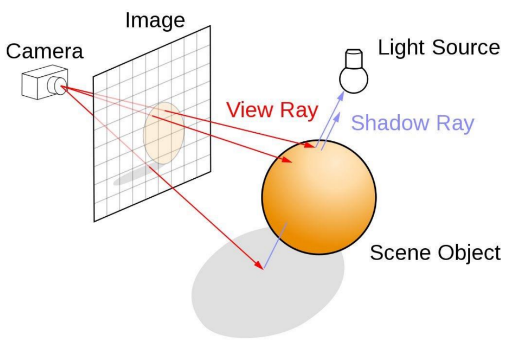

Figure 1: Visualization of a camera tracing rays through a simple scene, creating rays that intersect the sphere primitive and creating shadows by casting rays to the light source.

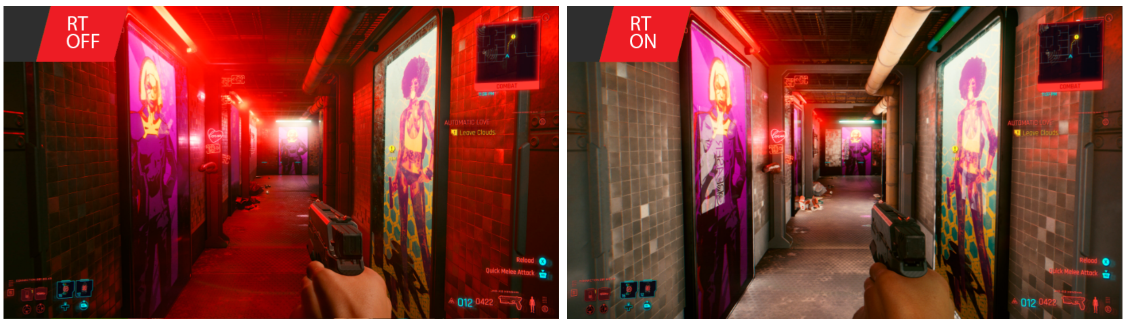

Figure 2: Comparison of ray tracing vs the standard lighting implementation in Cyberpunk [1]. In the right image, we can see much better shadows and reflections on the floor and walls.

However, ray tracing requires many rays per pixel to obtain an accurate color; otherwise, the final image might be noisy or miss essential details in the lighting. Due to the inherent parallelism on GPUs, they make it trivial to trace rays in parallel as every ray does not rely on any other rays in the frame unless it is an extension of a previous ray. Even with the GPU and specialized hardware, it is still crucial to minimize the time it takes to find an intersection, or the *query time of the scene*. This is because a large number of rays are required to produce an accurate image. Therefore, a single frame of ray tracing consists of two phases: the *build phase* and the *tracing phase*.

In the build phase, the ray tracer builds an acceleration structure to reduce the query time of the scene significantly. The industry standard acceleration structure is the *Bounding Volume Hierarchy* (BVH), which organizes primitives in hierarchical *bounding volumes*, which are usually *axis-aligned bounding boxes* or *oriented bounding boxes* [2] [3]. A ray then traverses the hierarchy of the BVH when intersecting an object, removing the need to test for intersection with part of the hierarchy if the ray misses a bounding box. Spending more time during the build phase to create a higher quality BVH can significantly benefit the tracing phase, as this could greatly reduce the query time. However, spending too much time on the build phase could reduce the total *framerate* if many builds are required.

Because *static scenes* do not deform or change, we require only a single build phase for these scenes at the start of a program. However, games and movies use *dynamic scenes* with many moving and animating objects. Due to the change in the position of the primitives, many BVHs either need to be (partially) rebuilt or refit every frame to accommodate the underlying deformation of geometry [4][5]. A fast operation for this is called *Refitting* [5], which changes the existing hierarchy to accommodate the change in the existing hierarchy. However, Refitting can significantly worsen the overall performance of a BVH because the refitted hierarchy could have unconsidered overlap between branches of the hierarchy, which results in poor query times. Therefore, many applications build full BVHs after a certain amount of deformation to achieve a faster tracing phase [5]. However, because players want their games to run at 30+ fps (and higher is better), there is only a limited amount of time available to build BVHs. Therefore, we need to have low build times for BVHs, potentially at the cost of quality. So, many studies investigate how to build a 'good' quality BVH every frame in parallel on the GPU. We prefer construction on the GPU over the CPU as the GPU can handle more primitives in parallel, and construction on the CPU requires uploading the BVH to the GPU, adding further overhead to the total rendering pass. Many of these fast GPU construction algorithms require the use of *Morton Codes* [6] [7] [8] [9] [10], as these make the parallelization of the construction easier.

Morton Codes [11] describe a *curve* (called the *Morton Curve*) created by spatial median splits, where an b-bit integer value maps to a point along the curve. Two b-bit integer values close to each other in value will also be close in 3D space, so an array of primitives sorted along this curve is helpful for many algorithms. Sorting is a well-known problem with many algorithms available for the GPU [12] [13], so this is a fast operation. Examples of construction methods that use sorted Morton Codes are Parallel Locally-Ordered Clustering (PLOC) [8], Approximate Agglomerative Clustering (AAC) [14], and LBVH [6]. PLOC and AAC use this sorted array to quickly search for the nearest neighbor. LBVH exploits the fact that sorting the primitives along the Morton Curve and looking at their *b*-bit values implicitly describes a BVH that uses spatial median splits. The LBVH is today's fastest construction algorithm and one of the most prominent BVHs in real-time applications that use many moving and animating objects.

Although there has been much research in improving the quality of these LBVHs [7] [10] or improving the construction time [15], there is not much research in improving the *accuracy and quality of the Morton Codes*, such as positional accuracy and splitting along better planes than the spatial median plane. One paper describes *Extended Morton Codes* (EMC) [16], which improves the accuracy for irregularly shaped scenes by assigning more accuracy to longer axes and encoding primitive size. However, both Morton Codes and Extended Morton Codes have one fatal flaw: they are directly related to the *scene's bounds*. One possible situation that a programmer could find himself in is that he wishes to 'hide' some primitives from the player's view by moving the primitives far away, such as projectiles, or an artist uses large primitives far away from the player to create a background for the scene, both creating a larger scene size. The larger scene size deteriorates the accuracy of the Morton Code and the Extended Morton Code, which leads to poorly constructed BVHS and increased rendering times.

Therefore, we ask the following research question:

Can we improve the Extended Morton Codes further to construct BVHs of equal or higher quality in real-time applications? In particular in situations where the scene's bounds grow out of proportion due to distant or large primitives at the extent of the scene?

We will achieve this improvement for distant primitives by creating an *occupancy grid* containing information about specific sections of the scene, potentially moving splits or changing parts of the code to enhance the accuracy of the Extended Morton Code. Furthermore, we propose minor changes to the Extended Morton Code's computation to handle large primitives at the extent of the scene to enhance the positional accuracy that can be implemented immediately in any existing computation of the codes.

This thesis starts with a literature review in Section 2 of the current research of construction methods with BVHs, including Morton Codes, Extended Morton Codes, and some extensions for BVHs. Section 3 describes our newly proposed methods of this thesis in more detail. Section 4 shows how we set up the experiment, gives some information on the hardware used, and shows the results for the given methods, followed by Section 5, which will conclude and discuss the results of this thesis.

2 Background and Related Work

This Section will discuss the field of BVH construction. In Section 2.1, we will explain a Bounding Volume Hierarchy (BVH) and its benefit over other acceleration structures. Then, we describe some quality metrics for the BVH in Section 2.2. After that, we show some existing construction methods and discuss their way of solving our problem in Section 2.3. Then, we introduce Morton Codes in Section 2.4 along with some fast construction methods that use Morton Codes in Sections 2.4.1 and 2.4.2, followed by an extension to the Morton Code in Section 2.4.3. We then list some extensions for BVHs used in this research in Section 2.5. Lastly, in Section 2.6, we will conclude the currently available methods to deal with distant or large primitives at the edges of the scene.

Furthermore, Table 1 shows the definitions used throughout the thesis.

Definition	Meaning	**Abbreviation**
Scene	A set of 2D or 3D primitives	
Real-time Applications	Applications that require high framerates such as games	
Offline Applications	Applications with more render time available per frame, such as movies	
Static scenes	Scenes where all objects are static and do not animate	
Dynamic scenes	Scenes where objects can move and have animation	
GPU	Graphics Processing Unit	**GPU**
Morton Codes	A Bitvector describing a d-dimensional position	**MC**
Morton Curve	Z-Curve created by Morton Codes	
Extended Morton Codes	An extension to Morton Codes with more information in the bitvector	**EMC**
Scene's bounds / Scene size	The bounding box of all primitives in a scene	
Bounding Volume Hierarchy	A hierarchy consisting of bounding volumes that encapsulate primitives	**BVH**
Bounding Box	An Axis Aligned or Oriented Bounding Box that can be used as a Bounding Volume in a BVH	**AABB** or **OBB**
BVH cost	The expected cost of tracing a given BVH	
Surface Area Heuristic	A heuristic specifying the geometric cost of a BVH	**SAH**
End-Point-Overlap	Additional cost for SAH to penalize surfaces that are in multiple Bounding Volumes	**EPO**
Linear BVH	A BVH that only uses Morton Codes for construction	**LBVH**
Hierarchical LBVH	An addition to LBVH by using a SAH top-down builder at the top-most levels of the tree	**HLBVH**
TLAS/BLAS	Top- and Bottom-Level Acceleration structures	**TLAS/BLAS**
Teapot in a stadium	Problem where a teapot is in a stadium, requiring high accuracy at certain sections in a large scene	
Dword	A 32-bit unsigned integer	

Table 1: Definitions used throughout this thesis.

2.1 The Bounding Volume Hierarchy (BVH)

The BVH, first described by Clark et al. [17], is a hierarchy that does not split space but rather splits a set of primitives into subsets with a bounding volume around each subset. The bounding volumes can be any volume, but these are usually Axis-Aligned Bounding Boxes (AABB) or Oriented Bounding Boxes (OBB) [2] [3], as boxes are fast to compute, fit inside of each other well, and are memory efficient (AABBs only require 6 floats per box, 3 for the minimum bounds and 3 for the maximum bounds). In the BVH, subtrees are allowed to overlap, and primitives are stored once in a single leaf. This object-splitting hierarchy differs from spatial-splitting hierarchies like KD-Trees, which can have multiple objects in the leaves, but nodes cannot overlap [18]. BVHs are the industry standard acceleration structure nowadays due to the following reasons:

- **Fast query times:** For a set of random rays that want to intersect a set of non-overlapping primitives, the BVH has an average $O(\log n)$ query time, where n is the number of primitives, as we can efficiently prune branches that do not intersect a given ray. Furthermore, BVH traversal algorithms typically have a small memory footprint and a compact traversal state, which makes them well-suited for GPU traversal. Compared to a kD-Tree, another acceleration structure with an average $O(\log n)$ query time, where n is the number of primitives, the BVH is at least comparable to or even better than the kD-Tree in traversing the acceleration structure and pruning branches [18].
- **Predictable memory footprint:** The memory footprint of a BVH is bounded by the number of primitives since each primitive is referenced only once in a leaf. The BVH contains $v \leq 2n-1$ nodes, where n is the number of primitives and v is the number of nodes, which can be the case for a binary BVH (a BVH with two children per node) [19]. The kD-tree does not have a predictable memory footprint, making it hard to use on GPUs.
- **Generality:** The BVH can be constructed for any scene due to its hierarchical nature. For instance, the BVH can handle the *teapot in a stadium* problem, where a high-resolution version of the Utah teapot is at the center of a large, low-resolution stadium. The teapot in a stadium was a complex problem for acceleration structures that use uniform grids, as this scene would have a considerable bounding box requiring large grid cells. However, for the teapot, smaller grid cells would be beneficial. Although the kD-Tree can also handle the teapot in a stadium problem, the BVH can solve this by separating the teapot from the stadium at the first split, which would require six planes in a kD-Tree.

2.2 Quality Metrics for BVHs

Quality metrics describe how well a BVH for a scene will perform for a given set of rays. A higher quality BVH usually also indicates better rendering performance than a BVH with lower quality, as higher quality BVHs tend to have less overlap and smaller boxes for many primitives, which allows them to prune more primitives efficiently.

2.2.1 Cost of a BVH

The cost of a particular BVH is an estimation of the expected number of operations needed to find the nearest intersection [20]. The cost of a BVH from subtree x is the following recurrence equation:

$$COST(x) = \begin{cases} c_I + \sum_{y \in Y_x} P(y|x)COST(y) & \text{if x is interior node} \\ c_E|x| & \text{otherwise} \end{cases} \tag{1}$$

where $COST(x)$ is the cost of a subtree with root node x, y is a child subtree of the set of children Y_x of node x, $P(y|x)$ is the conditional probability of traversing a node y when intersecting node x, and $|x|$ is the number of primitives in a subtree with root x. The recurrence also contains two constants, c_I and c_E, which express the average cost of a traversing an internal node and intersecting a leaf node, respectively. Finding the optimal lowest possible cost $c(\mathbf{x})$ of a root node $\mathbf{x}$ of a BVH is believed to be an NP-Hard problem [21].

2.2.2 Surface Area Heuristic (SAH)

SAH expresses the conditional probabilities of the recurrence in Equation 1 as a geometric probability [22]. This geometric probability is the surface area of the bounding box (AABB or OBB) of a node x, as intuitively, if we have a smaller surface area, there is a smaller chance that a random ray will intersect the bounding box. Therefore, we can write the probability relation of the formula 1 as follows:

$$P(y|x)^{\text{SAH}} = \frac{SA(y)}{SA(x)} \tag{2}$$

Where $SA(x)$ is the surface area of the bounding box of node x. We substitute this probability into formula 1:

$$SAH(x) = \begin{cases} c_I + \sum_{y \in Y_x} \frac{SA(y)}{SA(x)} SAH(y) & \text{if x is interior node,} \\ c_E|x| & \text{otherwise} \end{cases} \tag{3}$$

Unrolling removes the recurrence:

$$SAH(x) = \frac{1}{SA(x)} \left(c_I \sum_{y_i \in I_x} SA(y_i) + c_E \sum_{y_e \in E_x} SA(y_e)|y_e| \right) = \sum_{y \in Y_x} c_y \frac{SA(y)}{SA(x)} \tag{4}$$

Where I_x and E_x are the set of interior and leaf subtrees with root x, respectively, with $I_x \cup E_x = Y_x$ is the set of children of node x. Next, c_I and c_E are constants of the average cost of a traversing an internal node and intersecting a leaf node, respectively, and c_y is the cost of traversing node y. Formulas 3 and 4 show that we prefer small boxes for as many primitives as possible.

2.2.3 End-Point-Overlap (EPO)

SAH assumes a uniform distribution of ray origins and directions in an n-dimensional space and that rays originate outside the scene bounds. However, many types of rays, such as shadow and reflection rays, originate from the surfaces of primitives. Therefore, EPO aims to penalize overlapping surfaces between two bounding boxes that are not in the same subtree, as a ray originating from a point within the overlap has to test for intersection in both subtrees [23]. EPO assumes the uniform distribution of ray origins and hit points on surfaces. The probability of having a hit point inside a node is proportional to the surface area of the primitives inside that node's volume. The expected cost of searching for a ray's origin or end point from the tree is:

$$EPO_{cost}(x) = \sum_{y \in Y_x} c_y \frac{A(F \cap y)}{A(F)} \tag{5}$$

Where F is the set of all surfaces in the scene, y is a child subtree of the set of children Y_x of node x, $A(F \cap y)$ is the total area of surfaces inside the bounding volume of child subtree y, normalized to the probability that the query point resides inside y, and c_y is the cost of traversing node y. We then define EPO as:

$$EPO(x) = \sum_{y \in Y_x} c_y \frac{A((F \setminus Q(y)) \cap y}{A(F)} \tag{6}$$

Where y is a child subtree of the set of children Y_x of node x, $Q(y)$ is the set of surfaces that belong to the child subtree y, $(A \setminus Q(y)) \cap y$ is the geometry that does not belong to the subtree of y but lies within the volume of y, and c_y is the cost of traversing the child subtree y. The authors designed EPO to be zero if there is no overlap between subtrees, so it is usable as an addition to SAH [23]. EPO shows a good correlation between actual cost and render times. However, we cannot use EPO during construction because the entire BVH tree is necessary for EPO evaluation. Popov. et al. did come up with another heuristic that penalizes the overlap of child-bounding boxes, which is less descriptive and has a weaker correlation with rendering times than EPO but can be evaluated at construction time [24]. Nevertheless, the heuristic shows a significant reduction in the cost of a ray when restricting the amount of overlap between child-bounding boxes.

2.3 BVH Construction

There are many different algorithms to construct BVHs. However, we will only cover some of the construction algorithms relevant to this research and the domain here. To learn about the other methods and a more complete survey of the whole ray tracing research, we refer to the survey of Meister et al. [20].

2.3.1 Top-Down Construction

Top-down construction starts with a set of all primitives in the root node. It iteratively splits the set of primitives until a node meets some termination criteria to become a leaf or when reaching a depth bound. There are exponentially many ways to split the set of primitives. Popov et al. showed that $O(n^6)$ partitionings exist for axis-aligned bounding boxes [24], where n is the number of primitives. In practice, none of the top-down approaches check all of the $O(n^6)$ different partitionings, as the runtime of such an algorithm would be infeasible. In practice, we split the set of primitives by axis-aligned planes, similar to how kD-tree construction works. The position of a primitive relative to the splitting plane is determined based on its centroid point. A fast Top-Down approach, running in expected $O(n \log n)$ time where n is the number of primitives, splits the set of primitives based on the spatial or object median, as the primitives only have to be compared to a single splitting plane. However, median splits can only generate mediocre BVHs at best, so some Top-Down approaches opted to approximate the SAH score at each level for k splitting planes. We cannot use the cost model directly as the cost of the children is unknown; therefore, this is a greedy approach. The greedy approximation of the SAH score at the current level is as follows [25]:

$$SAH_{greedy}(x) = \min_{L \subset N_x, R = N_x \setminus L} \left(\frac{SA(L)}{SA(x)} * |L| + \frac{SA(R)}{SA(x)} * |R| \right) \tag{7}$$

Where N_x is the set of primitives of node x, L and R are subsets of N_x and the splitting plane's left and right sides, respectively. $SA(X)$ is the surface area of the set X and $|X|$ is the number of primitives in the set X. From this score, we can also derive a termination criterium: when the cost of the node being a leaf outweighs the cost of splitting this node, the node is considered a leaf and thus will not split further. Aila et al. showed that Top-Down builders are better in terms of EPO than Bottom-Up builders, as they implicitly reduce EPO [23]. The runtime of this approach is $O(kn \log n)$ expected time, where k represents the number of splitting planes, and n is the number of primitives. Along a single axis, there are at most $n-1$ possible splitting planes, as there are n primitives in a scene; therefore, $k = n-1$ if we want to compare all splitting planes along an axis, which gives a runtime of $O(n^2 \log n)$. However, Wald et al. showed that even with a limited number k of equally spaced bins, it is possible to construct high-quality BVHs [25]. They further worked out this idea by proposing a parallelized approach that utilized SIMD instructions and multithreading on a CPU. Top-down SAH builders would handle the case of distant or large primitives well, as they would immediately recognize a significant drop in SAH cost if they first split off the problematic primitives; however, for animating scenes, the construction time of these builders is too high.

2.3.2 Bottom-Up Construction

Bottom-up construction starts from the leaves and eventually reaches the root node through multiple merging steps. Walter et al. [9] proposed using agglomerative clustering, which merges pairs of sets based on a distance function at each iteration step. The distance function for two sets of primitives is the surface area of a bounding box that encloses both sets, so smaller parent boxes indicate a low distance. This process repeats until there is only one cluster left. There are two issues with this approach:

1. This iteration step works very well for lower levels, but top levels can be poorly optimized.

2. The construction time is slow, as in the best case with a standard distance function, there are $O(\alpha^2)$ nearest neighbor searches at each iteration step, where α is the number of clusters.

Walter et al. [9] proposed accelerating the nearest neighbor search with a heap data structure and a kD-Tree, which would be difficult to parallelize. Therefore, Gu et al. [14] proposed approximate agglomerative clustering (AAC), which uses the Morton Curve (see Section 2.4) to make initial partitions of the primitives until a subtree contains less than a predefined number of clusters. To reduce the number of clusters in a subtree, AAC merges the clusters until a small number of clusters remains, which are, in turn, merged using agglomerative clustering. To accelerate the nearest neighbor search, the authors proposed using a distance matrix with a quadratic number of entries based on the number of clusters. This distance matrix requires a large stack state, which is not GPU-friendly.

Meister et al. [8] came up with a more parallelizable approach for AAC, called Parallel Locally-Ordered Clustering (PLOC). They used an observation that the distance function follows the non-decreasing property, which says that if two nearest neighbors mutually correspond, then there will be no better neighbor. PLOC then merges all mutually corresponding nearest neighbors in parallel. PLOC also uses the Morton Curve, but instead of reducing the number of clusters, it uses the curve to accelerate the nearest neighbor search. Each cluster is sorted along the Morton Curve and finds its nearest neighbor by looking in both directions along the curve for γ steps.

PLOC can run entirely on the GPU in three steps. Each cluster finds its nearest neighbor in the first step by looking along the Morton Curve. Then, we merge mutually corresponding pairs and assign their new point along the Morton Curve to be the point of the first cluster. Finally, a parallel prefix scan removes the holes created by merging. In the worst case, the algorithm may only merge a single cluster at each iteration step, making the worst-case runtime on the GPU of this algorithm $O(\gamma n)$, where n is the number of merges, and γ is the number of nearest neighbor searches at each level. In the best case, every cluster has a mutually exclusive neighbor, which makes the best-case runtime $\Omega(\gamma \log n)$. In practice, the algorithm comes closer to the best-case runtime than the worst-case runtime.

Bottom-up builders would handle the case of distant or large primitives well, as they would not merge distant or large primitives early. However, most Bottom-up builders use Morton Codes, so these methods also suffer from large scene bounds, and the methods that do not use Morton Codes have high construction times.

2.4 Morton Codes

As we already read in Section 2.3.2, some algorithms use the Morton Curve. The Morton Curve is a space-filling curve described by Morton Codes (MCs) [11], which provides a coherent ordering of quantized vectors, meaning that vectors with subsequent MCs are spatially close to each other [16]. This curve is also called the Z-curve because of the shape of the curve in the 2D case (see Figure 3).

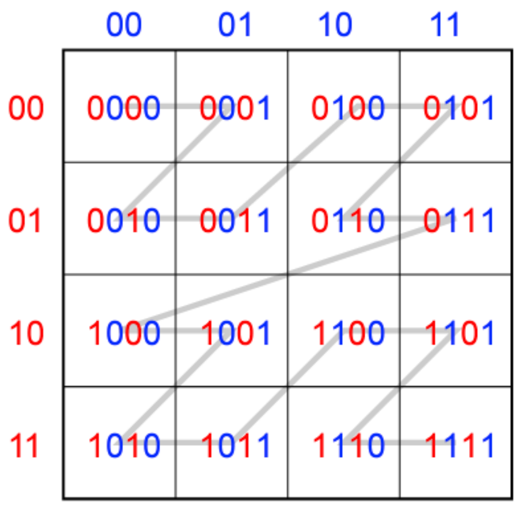

Figure 3: Visualization of the MC-based space-filling curve in 2D. Image is from the paper by Bittner et al. [16]

For a given point p in a scene, we find a point along the curve p' closest to p, or in other words, for a given point p, find out in which cell the point p lies. More formally, finding this point along the curve is a function $f(p) : \mathbb{R}^d \to B^b$, where d is the number of dimensions of a point, b the length of the vector, and $B = \{0,1\}$, such that $f(p)$ is a bit vector that encodes the point along the curve. A larger b gives us a more accurate bit vector, as every increase in b will increase the number of points along the curve by 2^b. We generally define this bit vector as an b-bit Morton Code, which maps to the closest point along the curve for the point p.

We can obtain the b-bit Morton Code of a normalized two-dimensional vector $\langle\vec{v}\rangle = (\vec{v}_x, \vec{v}_y) \in [0,1)$ by calculating which bin the vector lies in along each axis and interleaving the bits of the index of the different bins. For example, if there is a vector $\vec{v} = (12,4)$ in a scene with bounds $(0,0)$ and $(18,18)$ then the normalized vector $\langle\vec{v}\rangle = (2/3, 2/9)$. Using an 8-bit MC, the value of the axes can be calculated by multiplying by a factor of $2^{(b/2)} = 2^{(8/2)} = 2^4$ to get the bin along each axis, in this case, $(x,y) = (2/3 * 2^4, 2/9 * 2^4) = (10,3)$ (rounded down for integers). In 4-bit binary representation, this is $(x,y) = (1010, 0011)$, and interleaving these bits in the following fashion: $x_0y_0x_1y_1x_2y_2x_3y_3$ gives us the final 8-bit MC for this vector 10001101. This example also shows that the accuracy increases if we increase the number of bits used in the MC as the number $2^{(b/2)}$ increases, creating more bins along each axis. In 3D, we include an additional axis in the interleavement and change the factor to $2^{(b/3)}$.

2.4.1 Linear BVH (LBVH)

Another BVH construction method, created by Lauterbach et al. [6], uses MCs to construct a BVH. The algorithm assigns an MC to each primitive in the scene. The algorithm continues construction in a Top-Down fashion, looking at the most significant bit of the MC for each primitive and placing those with 0 and 1 bits in child buckets 1 and 2. We apply this procedure recursively to each child until the algorithm has consumed all of the bits of the Morton Code. The authors note that this is equivalent to the steps of a most significant-bit radix-2 sort using the Morton Codes as keys, allowing us to use a higher branching factor i using a radix-i algorithm instead.

The main advantage of LBVH is that it is inherently parallel, as we can quickly make a correct ordering of the primitives using a parallel radix-sort algorithm. Next, the MCs already encode all the necessary information about where in the tree a particular primitive will go, which means that MCs already encode a BVH based on spatial median splits. Therefore, Lautherbach et al. proposed to create split lists, where a thread for each sorted primitive determines where in the tree it splits with its right neighbor. This split list is then sorted based on depth, so they obtain a list of all splits that should occur on each level. This method does come with one unwanted side effect: it can create chains of singleton nodes in the tree, which are nodes that only have a single child, which requires an additional post-process step to roll up singleton chains. Finally, the bounding boxes must be processed, a procedure similar to a refit pass (see Section 2.5.1). The time complexity of such a BVH is $O(n)$, where n is the number of primitives, which makes them fast to construct.

Karras later improved the LBVH algorithm [15] and constructed the whole LBVH tree in a single GPU kernel. Karras used a different node layout, where primitives and internal nodes exist in a separate array, and constructed a binary radix tree. The construction works by determining the direction of an internal node's range and finding the start/end of that range and the split position using a binary search to construct an ordered binary radix tree, which runs in $O(n \log h)$ time, where n is the number of primitives and h is the height of the radix tree. One issue with this approach is that it requires another pass over the nodes to compute the bounding boxes for each internal node. Apetrei solved this issue by proposing constructing the binary radix tree and the bounding boxes in one pass [26]. These improvements, combined with the low runtime, make LBVH the fastest construction algorithm to date.

However, in cases where there are distant primitives, LBVH starts to perform significantly worse, as the Morton Codes heavily rely on the scene bounds. In those cases, LBVH will group many primitives into the same boxes, which reduces the overall quality of the BVH.

2.4.2 Hierarchical Linear BVH (HLBVH)

The main disadvantage of LBVH is that it does not consider quality metrics, such as SAH or EPO (See Section 2.2); instead, it just splits primitives based on spatial median splits. Therefore, the quality of the LBVH is medium at best, and its query time is worse than most other construction methods. Lauterbach et al. suggested using a top-down SAH builder for small subtrees close to the leaves in the original paper. However, Granzha et al. [7] tried to improve upon LBVH by using a SAH builder at the topmost part of the tree, stating that this is superior in the context of ray tracing, as the essential part of the hierarchy where spatial overlap needs to be minimal is the top of the tree. They called this new method HLBVH and used LBVH as a compression step to run a Top-Down SAH builder at the top of the tree, resulting in a 3-19% reduction in the overall cost of the BVH. HLBVH was later improved with work queues to improve the construction speed [10].

Although HLBVH improves upon LBVH, the computation of the Morton Codes happens before using the Top-Down SAH builder. Therefore, the part of the BVH that LBVH builds could be significantly worse if there are distant primitives in the scene.

2.4.3 Extended Morton Codes

Extended Morton Codes (EMC) extends MC by additionally encoding object size, using a variable bit count for longer axes and an adaptive axis order [16]. EMC adds object size to the code by checking if the diagonal of the bounding box of the primitive is larger than the diagonal of the current bit in the EMC. The bounding box corresponding to a particular MC prefix follows an implicitly known regular subdivision of the scene volume. Figure 4 shows an example of splitting with EMC.

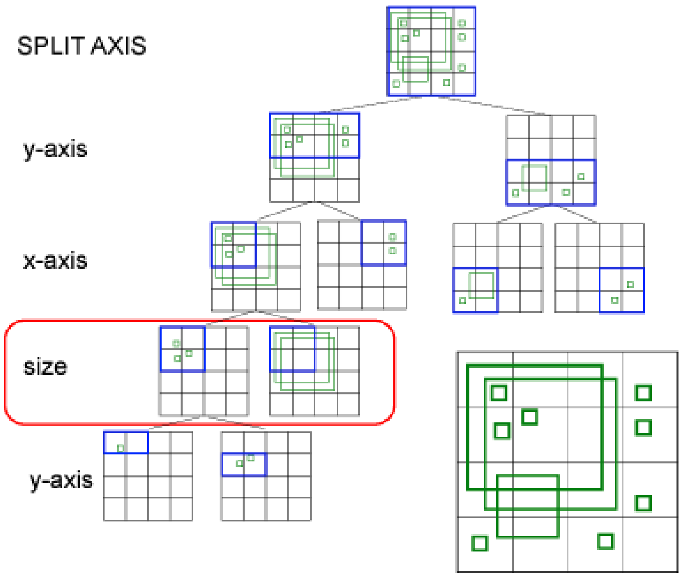

Figure 4: Example of how EMC splits primitives with Size bits on a simple scene. Illustration is from the paper by Bittner et al. [16]

To encode a variable bit count for each axis and an adaptive axis order, the EMC computes the order and the multiplication factor for each axis based on the size of the scene bounds. From these bounds, we can determine the longest axis for which we must do the first split at the first bit in the code. Then, we cut the longest axis in half and repeat this process for the subsequent splits in the code. After that, we count the number of bits for each axis and calculate their multiplication factor to obtain the final EMC. Note that this code is no longer the traditional MC that uses regular bit interleaving, but we can still compute the code quickly using loops and arrays.

These new codes for LBVH showed a 0 - 52% SAH cost reduction, with an average of 20% over well-known default scenes such as Sponza [27] and San Miguel [28]. EMC also shows a SAH cost reduction for other methods that rely on MCs, such as HLBVH [7] (16%), Approximate

Agglomerative Clustering [14] (11%), and Agglomerative Treelet Restructuring BVH (ATRBVH) [29] (7%).

EMC helps with distant primitives in a scene, as this computation will give more bits to larger axes. However, EMC still heavily relies on the scene bound, and if the scene bound grows too much out of proportion, many bits will be wasted on these large axes, deteriorating the final BVH.

2.5 Further Extensions for BVHs

Next to the different construction algorithms, BVHs also have some extensions to handle animated scenes, improve the quality further, or improve the render performance. Therefore, this Section will list some of the extensions used in this research.

2.5.1 Refitting

BVHs can handle animated objects in two ways:

1. Either rebuild the BVH every frame using a fast BVH construction algorithm or
2. Refit the existing BVH to reflect the deformation in geometry

Refitting is an operation whose time complexity is $O(v)$ on the CPU, where v represents the number of BVH nodes. On the GPU, the nodes on the same level can run in parallel, achieving a time complexity of $O(h)$, where h is the height of the tree. This low runtime makes refitting a good option for animating scenes. However, refitting does not consider the quality of the resulting BVH and can arbitrarily degrade the BVH. Therefore, Lauterbach et al. [5] proposed a degradation heuristic to show how much a BVH has degraded and determine whether the BVH should be reconstructed or refitted in a particular frame.

2.5.2 Subtree collapsing

Some construction algorithms, such as LBVH, create leaves with a single primitive, which could cause an increase in tracing cost as defined in Section 2.2.1. Therefore, subtree collapsing [21] starts from the leaves and goes up to the root to compare the cost of a node being an internal node or a leaf. If a leaf's cost is lower than an internal node's, we collapse the internal node to make it a leaf. Meister and Bittner [8] proposed a GPU-based version of subtree collapsing that uses several passes of the parallel bottom-up traversal for refitting proposed by Karras [15].

2.5.3 Wide BVH

An w-wide BVH contains a branching factor of w, so a binary BVH with branching factor 2 is a 2-wide BVH. These wide BVHs allow more efficient utilization of SIMD/SIMT units by testing a single ray against multiple bounding boxes during traversal [30]. Furthermore, wide BVHs have a lower depth and require less memory and fewer interior nodes than binary BVHs. However, only some internal nodes will have exactly *w* children; usually, this number is lower around the leaves, and therefore, it causes more algorithmic complexity to handle the additional cases.

There are two ways of building a wide BVH: constructing a binary BVH and collapsing internal nodes bottom-up or constructing a wide BVH from scratch. Wald et al. [30] proposed a collapsing fashion using three operators to minimize the cost function: merging a child node into a parent node, two leaf nodes, or two interior nodes. The other way is to construct a wide BVH during construction, such as using a radix-*i* algorithm for sorting and building an LBVH [6].

2.5.4 Top- and Bottom Level Acceleration Structures (TLAS/BLAS)

TLAS/BLAS is an idea introduced by Wald et al. [31]. The idea is to build a Bottom-level acceleration structure (BLAS) for every object and use a Top-level Acceleration structure (TLAS) to enclose all BLASes. For a BVH, this means that the root node of the BLASes can be the leaves of the TLAS. This two-level hierarchy has some advantages:

- **Different builders:** The objects in a scene can now use an optimal builder for that specific object. A static object can use a high-quality builder, while dynamic objects can use faster builders.
- **Smaller updates for Rigid Body Animation:** If an object has some Rigid Body Animation (animation where the object rotates or moves but does not deform), then only the TLAS requires an update to accommodate the animation.
- **Smaller updates for Animation:** If a dynamic object is animated and deforms, only the BLAS for that object and the TLAS require an update to accommodate the animation, while other BLASes in the scene can remain untouched.

The TLAS also ensures that in cases with distant primitives, the BLASes of the other objects can still be of high quality, and only the quality of the TLAS decreases. However, distant or large objects will still make for a low-quality TLAS, which is undesirable for performance.

2.5.5 Re-braiding

The TLAS has one flaw: highly overlapping objects in the world can significantly reduce the quality of the TLAS, which is not uncommon in real-world scenarios. Benethin et al. proposed a partial re-braiding scheme to ’open and merge’ BLASes during the TLAS build to reduce this issue [32], which reduces overlap and improves SAH cost. The method avoids excessive re-braiding of BLASes by only applying this step where it would provide the most gain in terms of the SAH quality. One consequence of this method is that the TLAS now contains multiple openings to the same object.

2.6 Conclusion from Literature Review

From this literature review, we can conclude that builders exist who can handle large and distant primitives in the scene. When using high-quality builders, such as the binned top-down builder from Bittner et al. [25] from Section 2.3.1 or the agglomerative clustering method from Walter et al. [9] from Section 2.3.2, the builders can handle scenes with distant primitives. The binned top-down builder can do this by recognizing a significant decrease in the heuristic score when splitting the distant primitives from the other primitives in the scene, and the agglomerative clustering method has a distance function that will be large when trying to merge the distant primitives. However, both methods require unfeasible construction times for interactive applications that require many rebuilds because of animations due to the number of splitting planes or nearest neighbor searches required to get a good result. Therefore, for interactive applications, we require the construction times to be low.

The fast construction algorithms use Morton Codes as a speedup for nearest neighbor searches (AAC [14]) or as a layout of a spatially split BVH (LBVH [6]). However, Morton Codes suffer from increased scene bounds size when distant or large primitives exist. So, methods such as AAC [14] and PLOC [8] from Section 2.3.2, LBVH [6] and HLBVH [7] from sections 2.4.1 and 2.4.2 all suffer from the same issue when the scene bounds grow. EMC [16] from Section 2.4.3 does help with these instances by assigning more bits to larger axes, but if the scene's bounds grow out of proportion, it still suffers.

Extending our BVH with a hierarchical TLAS/BLAS structure helps mitigate the issues of large and distant primitives at the object level, allowing for high-quality BVHs for every object in the scene. However, the quality of the TLAS still deteriorates with distant or large objects at the edge of the scene, and a bad TLAS could still be a bottleneck for an interactive application.

Therefore, from all of this evidence, we conclude that a new method is required to build a better-quality BVH quickly in scenes with large or distant primitives. As all fast builders use Morton Codes, the best location to increase the quality of these builders is by improving the Morton Code. Furthermore, improving these new Morton Codes will also improve all construction methods that require these codes.

A full comparison of all discussed builders in this Section is visualized in Table 2.

Name	GPU	Speed	Quality	Uses MC	Can Handle Large Scenes
Binned Top-Down	Yes	Very Slow	Very Good	No	Yes
Agglomerative Clustering	No	Very Slow	Good	No	Yes
AAC	No	Slow	Good	Yes	No
PLOC	Yes	Fast	Good	Yes	No
LBVH	Yes	Very Fast	Medium	Yes	No
HLBVH	Yes	Medium	Good	Yes	No

Table 2: Contains a comparison of all discussed builders

3 Proposed Methods

This chapter will explain our new proposed methods to overcome the issues with the current EMC implementation [16]. Section 3.1 will list the requirements for our new methods. Then, Section 3.2 explains our novel occupation grid, followed by our two new methods that use our occupation grid to optimize scenes with distant primitives, mipping and binning, described in Section 3.2.2 and 3.2.3 respectively. Lastly, Section 3.3 will explain our minor changes to scene bounds computation to enhance the positional accuracy for scenes with large primitives at the edge of the scene, which can be implemented immediately in any existing computation of the codes.

3.1 Requirements

As we saw in Section 2, a fast builder does not handle scenes with distant or large primitives well, as many fast builders [6] [7] [8] [14] rely on Morton Codes [11], which suffer from the teapot in a stadium problem, while builders that can handle scenes with distant or large primitives have high construction times. Therefore, we propose improving Extended Morton Codes [16] for distant or large primitives so that fast builders can create BVHs of higher quality in these cases. Our new codes must fulfill the following criteria:

- **Efficiently Handling Distant Primitives:** Our method should perform equal or better than EMC in cases where the scene bounds stretch into one or more directions. The easiest way to asses this is by adding distant primitives to existing scenes and using both EMC and our new method to compare their respective LBVHs in terms of rendering performance.
- **Efficiently Handling Large Primitives:** Our method should perform equal or better than EMC in cases where the scene bounds stretch because of large primitives at the edge of the scene. The easiest way to assess this is to use a large scene with large objects and use both EMC and our new method to compare their respective LBVHs regarding rendering performance.
- **Generality:** Our method should work for any scene or object.
- **Low Total Build Times:** The overall build time compared to LBVH should not increase significantly, making our method feasible for real-time applications such as games.
- **Black Box Behaviour:** Our new codes must be directly usable for existing construction algorithms that rely on Morton Codes without changing the construction algorithms.
- **Low Global Memory Requirement:** As games use many objects in their scenes, we want to use as little global memory as possible so that the data necessary for constructing a BVH stays as low as possible.
- **Simplicity of Use:** Our new codes should be easy to use without tuning many complex parameters per scene.

3.2 Improvements for Distant Primitives

3.2.1 Occupation grid

While investigating low-bit EMCs of 3D objects, we found that many objects use less than 50% of the possible codes. For instance, a 9-bit MC has 287 cells that do not contain any primitives from the 512 available cells for the Stanford bunny. The Stanford bunny is a relatively dense object, so many objects will have more empty cells, mainly because primitives are rarely evenly distributed over a scene, as that would look like noise or the inside of an object contains unnecessary primitives. Also, many objects are concave in shape (they curve inwards, like spoons, bowls, or starfish) instead of convex (curve outwards, like spheres, dice, or capsules), reducing the occupied space. Furthermore, some objects inherently create many empty cells, especially when an object has an L- or T-shape somewhere in its design, such as the objects in Figure 5.

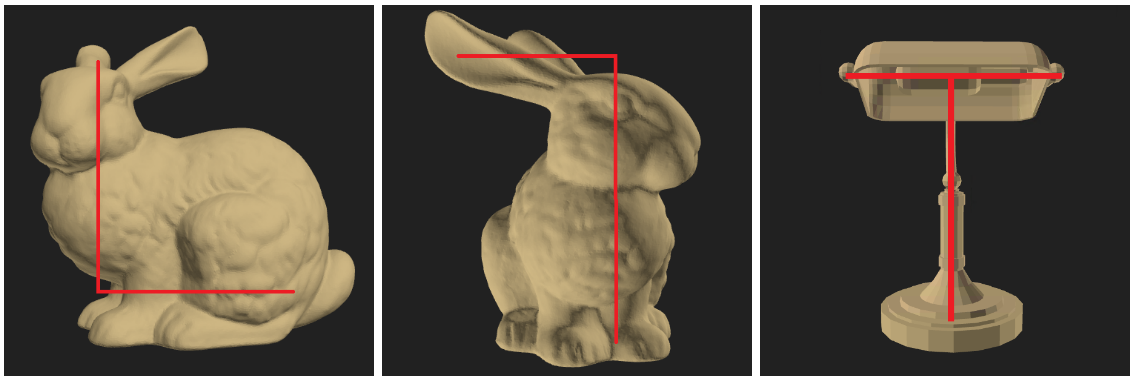

Figure 5: Visualization of the Stanford Bunny's L-shape and a lamp's T-shape.

In the case of a TLAS with re-braid enabled, objects could be very dense at one point in the scene but distant at another due to the 'opening' of BLASes from re-braid. Therefore, re-braid requires more accuracy in parts of the scene where many BLASes overlap.

Therefore, considering that the low-bit EMCs (without size bits) make up a grid and the low-bit EMCs are indexes of different cells, we can store information about the scene in the cells. The low-bit EMC can then serve as an index to the cell of a primitive, which can be a single-bit boolean (true or false) that indicates that a primitive occupies the cell or a 32-bit dword, where a dword is a 32-bit unsigned integer, containing the number of primitives in the current cell. We can use this information to enhance the code by removing or moving splits. Examples of these different occupation grids are given in Figure 6.

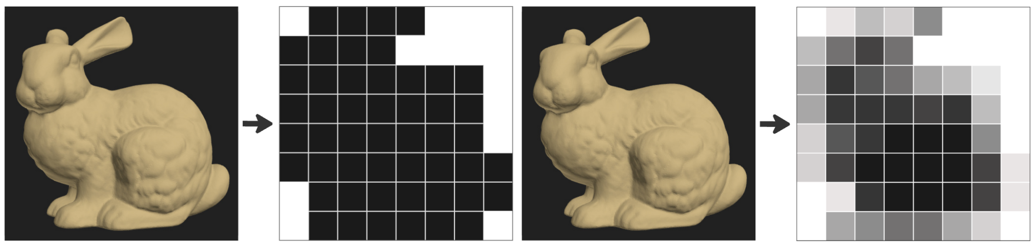

Figure 6: Shows the two different types of occupation grids. The left Figure is a binary bitmap containing black cells that show occupied cells, while the right Figure counts the number of primitives in a cell.

The global memory requirement for such a grid will scale exponentially by the number of bits used in our occupation grid. We will describe the memory in bits β and dwords u.

For an occupation grid that uses a boolean bit to indicate the occupation of a cell (1 for occupied, 0 for empty), the memory requirement is $\beta = \Theta(2^b)$, where b equals the number of bits used for the grid. For single bits, $u = \lceil \frac{\beta}{32} \rceil$, as every bit fits into a dword. For an occupation grid that stores a count of the number of primitives in a cell, $u = \Theta(2^b)$ and $\beta = u * 32$. Table 3 shows the analysis for low-bit occupation grids.

Bits for Occupation Grid	3	4	5	6	7	8	9
1-bit boolean (β)	8	16	32	64	128	256	512
1-bit boolean (u)	1	1	1	2	4	8	16
32-bit counters (β)	256	512	1024	2048	4096	8192	16384
32-bit counters (u)	8	16	32	64	128	256	512

Table 3: Shows the global memory requirement for using a specific type of occupation grid.

As bits and counts can come from any primitive scene, our occupation grid will use atomics for its construction. Per primitive, the work required is $\Omega(1)$. However, atomics could increase the build time as writing to the same memory requires serializing access to the memory address, which could be $O(g)$ time, where g is the number of workgroups.

3.2.2 Mipping

With *Mipping*, we create different layers of our occupation grid to compute which bits we need from the code quickly, similar to mipmaps used in games to create different levels of details of textures. Our method starts by assigning a cell to each primitive and flipping a corresponding bit to the cell in our occupation grid to a 1 with atomic operations to indicate a primitive in that cell. We then create mip layers in parallel by merging 8 bits of the previous layer into 1 bit in the next layer. If any of the previous 8 bits are on, the bit of the next layer will also be on. The outcome of this process would be the occupation grid if we used $3m-3$ bits, where m is the number of mip layers. We do this process *m* times, one time for each layer in shared memory. If we let every thread read eight 32-bit dwords, then every thread can construct one dword for the next layer, which prevents the necessity of atomic operations in higher layers. We show a 2D visualization of this process in Figure 7.

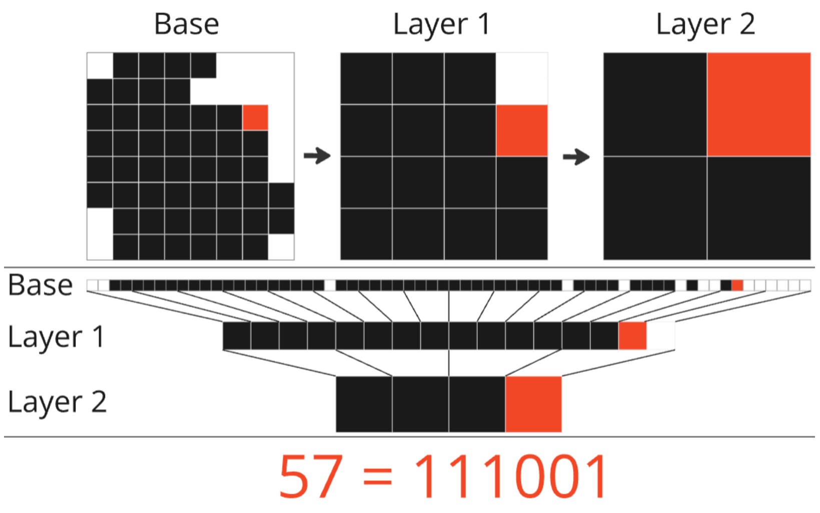

Figure 7: Shows the different layers of an occupation grid, where black means occupied and white means empty. At the top, we have a 2D visualization of the occupied cells, and below, we have a representation of the 2D map in memory. Every layer merges 4 bits of the previous layer, which increases the size of blocks in the previous layer by 4. The cell indicated by red has a Morton Code value of 57, which is 111001 in binary. We will use this red cell as an example to reduce the bits necessary for all primitives in this cell.

When we want to compute the code for a primitive, we move through the mips to figure out which bits are necessary. We start at the topmost layer and consider the 8 bits that a primitive is in, which can be figured out from its cell code. We can then check which bits were necessary by looking at a lookup table that encodes the possible bit maps shown in Figure 8.

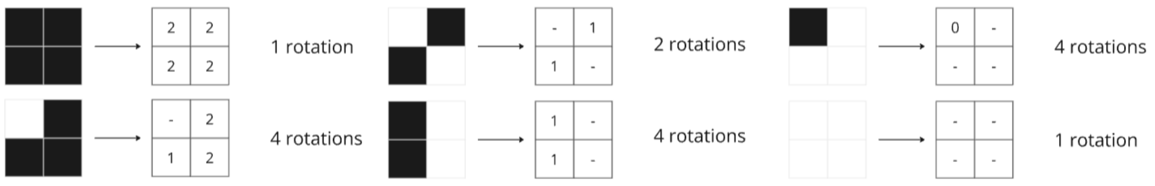

Figure 8: Shows the different possible 2D bit maps. Some of these bit maps have multiple rotations, and there are 16 possible bit maps, one of which is empty. For 3D, there are 256 possible bitmaps.

With these lookup tables, we can quickly find the necessary bits for a cell (An example is the red cell of Figure 7), as shown in Figure 9.

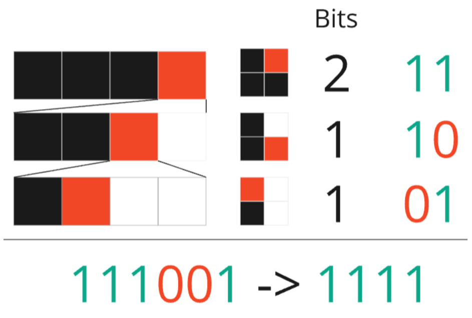

Figure 9: Shows the reduction in necessary bits for the red cell in Figure 7.

After figuring out the concise cell index containing only the necessary splits, we can compute the rest of the code within the cell. We then concatenate the cell index and the code within the cell to obtain the final code.

Our method's benefit is removing all unnecessary splits in the topmost part of the code and adding bits to specific sections. Furthermore, because the cells' bounds are smaller than the bounds of the scene, the final float calculation of the code within the cell will be more accurate. One disadvantage of our method is that it will give many bits to some parts of the scene, which might differ from the parts that require more accuracy.

Analysis: For generating the mip occupation grid, every thread in a workgroup handles a single primitive. Every thread then finds the cell for each primitive in $O(1)$ time and, in the best case, merges the occupied cells locally using an atomic bitwise or in shared memory in $\Omega(1)$ time. Then, every bit in shared memory merges with the global memory using an atomic bitwise or in $\Omega(1)$ time. So, the grid generation is in the best-case $\Omega(1)$ time, but the hardware will serialize accesses to the same memory addresses. Therefore, a more precise worst-case estimation is $O(t)$ time for the merge in shared memory and $O(g)$ time for the merge in global memory, where t is the number of threads in a workgroup and g is the number of workgroups. Alternatively, as a 32-bit dword describes $32 = 2^5$ cells and a grid has 2^b cells, where b is the number of bits used for the grid, we can have a complexity of $O(2^{b-5})$ time for grids that have $t > 2^{b-5}$ by assigning every thread with its piece of shared memory and merging the cells with a for loop and wave intrinsics. Therefore, the worst-case time complexity for generating the grid is $O(g+t)$ or $O(g+2^{b-5})$ based on the size of the grid.

Next, every thread finds the concise index of a primitive by finding its grid cell in $O(1)$ time and traversing the different mip layers, checking 3 bits per layer, in $\Theta(b/3) = O(b)$ time, where b is the number of bits used in the grid. Therefore, the total added complexity in the worst-case is $O(g+t+b)$ or $O(g+2^{b-5}+b)$ time, based on the size of the grid. We added the pseudocode of the whole method in Appendix 8.1.

3.2.3 Binning

Binning is similar to the binning method described by Bittner et al. [25], which used a fixed number of bins for faster top-down construction with SAH, with one key difference. We describe our binary binning method, where we use the occupation information from the scene and the surface area of the cells as an approximation of the surface area.

With Binning, we create a 7-bit occupation grid with one 32-bit integer for every cell, which keeps count of the number of primitives in that cell using atomic increments. We then create bit maps of the grid that describe the occupation with less accuracy but with better storage and efficiency of the computation.

To create these bit maps, we first convert our occupation numbers as if one would generate an axis code in an MC. We find the maximum and (non-zero) minimum occupation cell, and we convert our occupation to a a-bit value, where a is the desired accuracy of the occupation.

We use the a-bit value in $a+1$ bitmaps that describe the scene's occupation. We prepend one bitmap similar to the occupation grid of Mipping, where the bit is set to one if there is at least one primitive in the cell, making the search for the minimum and maximum occupied cell easier. The remaining bit maps will store every cell's different a-bit values. We visualize this process in Figure 10.

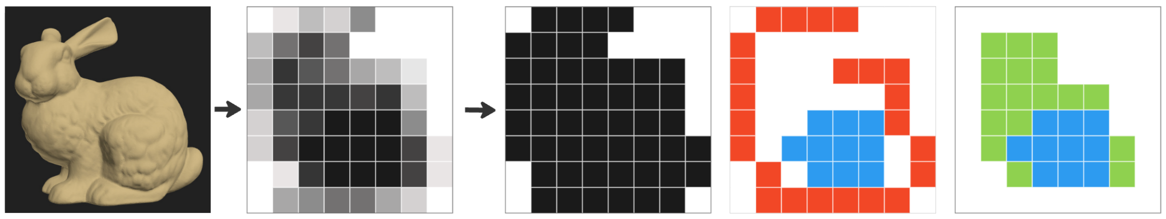
Figure 10: Figure showing the split of a counted occupation grid to a binary occupation grid with two layers for a 2-bit occupation value. The blue cells are the densest, followed by green and red.

A critical part of our method is finding the mask of the current split. We suggest changing the bit layout of the cell code to follow a different layout instead of the standard MC or EMC layout. First, find out the order of the different axes and sort them; we call the longest axis l, the middle m, and the smallest axis s. Instead of interleaving the bits, we start with all of the s bits, followed by all of the m bits, and lastly, the l bits. We use 7 bits, as this reduces the number of cases of different bits used within a single 32-bit dword. For 7 bits, the number of cases comes down to $mmlll$, $mllll$, and $lllll$. For 6 bits, we would also have the $smmll$ case when all axes are approximately the same length. Having an s bit in the 32-bit dword in the 7-bit case is impossible as we would need more than two bits for s, which would mean that s has more bits than m, contradicting that it is the smallest axis. As we know the number of bits l uses, we can create the mask and the minimum and maximum occupied cell of the current split with bitwise operations in $O(1)$ time.

To compute the approximate occupation o can then be computed from the a bit maps in the following fashion:

```
1: $a$ = #bits used for bitmaps
2: $bitmaps$ = bitmaps containing the occupation
3: $mask$ = mask of current split
4: $count$ = 0
5: for $layer = 0$; $layer < m$; $layer = layer + 1$ do
6:     $bitmap = bitmaps[layer]$ & $mask$
7:     $count = count + countbits(bitmap) * (2^{layer})$
8: end for
```

Where *countbits()* counts the number of non-zero bits in a bitmap, or in our case, the number of occupied cells in the current layer.

To compute the approximate SAH value of a split, we create a left and right mask for the split and calculate the value of the following function:

$$SAH_{cell}(x,l,r) = count_l \frac{SA(min_l, max_l)}{SA(min_x, max_x)} + count_r \frac{SA(min_r, max_r)}{SA(min_x, max_x)} \tag{8}$$

Where x is the parent node, l, r is the left and right split, respectively, and the function SA computes the surface area of the cells between a min and max value.

While evaluating the above cost function, we noticed that the cost function could make more rectangle-like boxes, which is undesirable for MCs or EMCs. Therefore, we decided to add a squareness factor called the 'Stretch factor,' which is the difference between the length of the smallest and largest axes. We define the cost of having a more rectangle-like box as follows:

$$SF(min,max) = \frac{\max_d((max - min)_d) - \min_d((max - min)_d)}{\max_d((max - min)_d)} \tag{9}$$

Where $\max_d$, $\min_d$ returns the maximum and minimum length of an axis inside a vector $\vec{v}$. By normalizing the scene extent by the largest axis, this function will return a value between 0 and 1.

To add the Stretch Factor to the SAH_{cell} score, we normalize the SAH_{cell} score to become:

$$SAH'_{cell}(x,l,r) = \frac{count_l * SA(min_l, max_l) + count_r * SA(min_r, max_r)}{(count_l + count_r) * SA(min_x, max_x)} \tag{10}$$

Which also accounts for normalization. We can then combine the scores with a parameter λ that determines the importance of the Stretch Factor:

$$Score(x,l,r) = (\lambda - 1) * SAH'_{cell}(x,l,r) + \lambda * (SF(min_l, max_l) + SF(min_r, max_r)) \tag{11}$$

We implemented binning by using a queue to efficiently and evenly distribute the workload of a single level over the different threads in a thread group.

Queue design: At any point, the maximum number of possible splits for a 7-bit grid is 127, which happens when the bit layout only consists of bits from the largest axis, which follows from the observation that the grid contains only 128 cells and, therefore, having more than 128 bins is impossible. This observation limits the complexity of the queue that we have to use. Therefore, we will only use a single wave (64 threads) to handle the queue, where each thread processes a maximum of 2 tasks, and no atomic increments are needed to track which task needs to be processed.

We have a queue containing tasks that are related to the splitting of a section. Each task contains z subtasks: $z = (bins_l - 1) + (bins_m - 1) + (bins_s - 1)$ Where z can be 0 if the split reaches a single cell, this single cell will also be a single subtask that pushes this cell and an empty cell to the next queue. Every thread finds subtasks in the queue and pushes them on its local stack. Every thread then works on the tasks to compute the score and performs a min operation if two subtasks have the same parent task. Then, an atomic min in shared memory finds the lowest scoring subtask for each parent task. The thread with the lowest scoring task enqueues the next sub-tasks. We perform q passes to create 2^q different splits, and we store the split that contains the current cell in the location that stored the occupancy of each cell to reuse and save memory. We added some pseudocode in Appendix 8.2.

Analysis: For generating the counting occupation grid, every thread in a workgroup handles a single primitive. Every thread then finds the cell for each primitive in $O(1)$ time and, in the best case, merges the occupied cells locally using an atomic increment in shared memory in $\Omega(1)$ time. Then, every bit in shared memory merges with the global memory using an atomic add in $\Omega(1)$ time. So, the grid generation is in the best-case $\Omega(1)$ time, but the hardware will serialize accesses to the same memory addresses. Therefore, a more precise worst-case estimation is $O(t)$ time for the merge in shared memory and $O(g)$ time for the merge in global memory, where t is the number of threads in a workgroup and g is the number of workgroups. We can use the same process as mipping, where we assign shared memory per thread, but as we use a 7-bit grid here, we will not cover the complexity for that version. The total time complexity for generating the grid is $O(g+t)$ time.

Then, we have to generate the $s = 2^d$ splits, where s is the total number of splits and q is the split depth. The number of processed tasks is $O(2^{q+1} - 1) = O(2^{q+1})$ time. The subtasks subdivide over the different threads in a for loop over all tasks per level in $O(s_l)$ time, where $s_l = 2^{q_l}$ is the number of splits at the current level l. Every thread can find the min and max in $O(1)$ time and compute the count of the primitives in $O(m)$ time of a subtask, where a is the number of bits for the approximated count. The number of subtasks a thread needs to process is at most 2 per level, so every thread needs to perform $O(2a) = O(a)$ work per level. The subtasks merge with an atomic in $O(t/s_l)$ time, and the threads write the new tasks in $O(1)$ time. So, for a specific level l, the complexity of that level is $O(t/s_l + s_l + m)$ time for q levels so a final complexity of $O(\sum_{l=0}^{q} t/s_l + s_l + a)$ time. We can approximate $O(\sum_{l=0}^{a} t/s_l)$ to $O(2t) = O(t)$ as the sum unrolls to $t + t/2 + t/4 + ...t/2^q \leq 2t$ for a final complexity of processing the queue of $O(t + 2^{q+1} + qa)$ time.

Lastly, to find the split that the current primitive lies in, we can read the split stored in the occupancy counter of the current cell in $O(1)$ time. Therefore, the total time complexity of our method is

$$O(g+t) + O(t + 2^{q+1} + qa) = O(g + t + 2^{q+1} + qa) \tag{12}$$

g is the number of workgroups, t is the number of threads in a workgroup, q is the split depth, and a is the number of bits used for the approximated count. Following this analysis, we want to keep the depth q low.

3.2.4 Expected Benefit Mipping and Binning

We expect that the occupation grid, combined with our Mipping and Binning strategies, will be able to delay a performance drop in large scenes when we grow the scene bounds by multiplying the bounds by 2^e, where e is the extension value. We expect our methods to delay the performance drop by the following equation:

$$move_d(f) = \log_2(f)/d \tag{13}$$

Where $d \in \{1,2,3\}$ is the number of dimensions the scene bounds grow and f is the number of grid cells in our occupation grid from Section 3.2.1. Then, if a performance drop happens at a specific e value, Mipping and Binning will have the performance drop at the following equation:

$$drop_d(f) = e + move_d(f) \tag{14}$$

Where e is the extension value, where the bounds is extended by 2^e, $d \in \{1,2,3\}$ is the number of dimensions the scene bounds grow and f is the number of grid cells in our occupation grid.

3.3 Improvements for Large Primitives at the Scene's Extent

Lastly, we propose a slight change to how to compute the scene's bounds. Usually, this is done by taking the bounding boxes of all primitives and computing the max and min of all bounding boxes. One could consider using the centroids as new bounds. However, we tested this strategy, and it came back with mixed results, likely due to a significant shift in splits for objects with large primitives, which might be bad for LBVH, which highly benefits from having the first split in the middle of the scene. Therefore, we calculate the max and min of all bounding boxes and the max and min of the centroids and then calculate the minimum difference of these bounds for each axis, which makes the bounds of the scene more concise while keeping the first splits in the center of the scene. We show the calculation of the concise bounds in Equation 15, and a visualization of the calculation in Figure 11.

$$\begin{aligned} difference &= vmin(centroid_{min} - bounds_{min}, bounds_{max} - centroid_{max}) \\ concise_{min} &= bounds_{min} + difference \\ consise_{max} &= bounds_{max} - difference \end{aligned} \tag{15}$$

Where *vmin* computes the minimum value of each dimension given two vectors, *bounds* are the standard scene bounds and *centroid* are the centroid bounds.

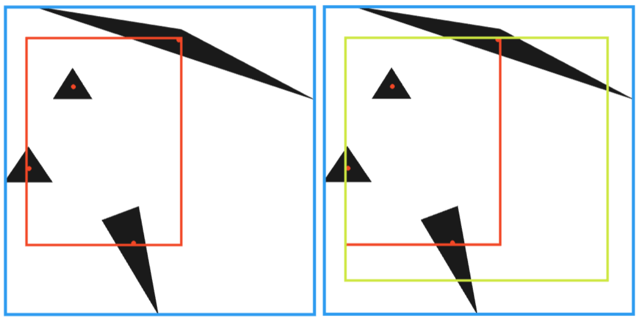

Figure 11: Shows the location of our concise scene bounds in 2D. The blue box is the normal scene bounds, the red box is the centroid bounds, and the green box is our new concise bounds.

4 Results

In this Section, we will look at the results of our new strategy of generating codes and other improvements presented in Chapter 3 for the fast construction and querying of ray tracing in real-world scenarios. To test these scenarios, we implemented our methods of Section 3 within the AMD DirectX 12 driver, which allows us to run the experiment on existing games and applications that use the DXR API [33] and use AMD tooling to generate results. Furthermore, we can lock the core clock speed of the GPU in the driver to create minimal variance in the results.

For this experiment, and with the requirements listed in Section 3.1, we are interested in the following:

1. **Efficiently Handling Distant Primitives:** How well do our Mipping and Binning strategies perform in large scenes with distant primitives? And what about regular scenes that do not contain distant primitives?

2. **Efficiently Handling Large Primitives**: How well do our new scene bounds perform in large scenes with distant primitives? And what about regular scenes that do not contain distant primitives?

3. **Low Total Build Times:** How much extra build time do our Mipping and Binning strategies require?

4. **Simplicity of Use:** Is there an optimal Stretch Factor parameter for our Binning strategy?

We do not require more experiments for the other requirements as they are fixed or based on a parameter of the strategy:

1. **Generality:** All of the strategies work for any scene or object

2. **Black Box Behaviour:** Our new strategies are all immediately usable for all existing builders that rely on Morton Codes

3. **Low global memory requirement:** For our new scene bounds, this is $6*4 = 24$ bytes; for Binning, this is $128*4 = 512$ bytes, and for Mipping, we set the number of grid bits $b \leq 12$, which is at most $4,096$ bits 512 bytes.

We will measure these metrics by building a 4-Wide LBVH with subtree collapsing for all TLAS/BLAS in the scene that uses our new codes to describe the hierarchy. Furthermore, the TLAS uses Re-braid, which creates extra dense regions in the TLAS due to the opening up of BLASes. Because we use AMD hardware, we can measure build times from the Radeon Developer Panel™[34], combined with the Radeon GPU Profiler™[35]. Furthermore, AMD provided us with an internal tool that captures game scenes and allows us to test the rendering performance of different BVHs for different ray types. The ray types that we are interested in, and that are most of the rays in a raytracer, are:

1. Primary Rays, which originate from the camera
2. Global Illumination Rays, which originate from surfaces, lighting up the environment
3. Ambient Occlusion Rays, short rays from surfaces adding more shadow due to closely positioned objects
4. Soft Shadow Rays, extra shadow rays sent to light sources to capture partially occluded light sources more accurately

There are other ray types, such as Reflection and Refraction rays, but these are more special ray types similar to Global Illumination Rays, originating from surfaces.

To measure build times, we will run the strategies 100 times over various scenes that differ in the number of primitives. These scenes will consist of a single BVH that encapsulates all of the primitives in the scene. This way, we can see how Mipping and Binning compare to the construction time of LBVH with EMC. We do not measure the increased construction time of the scene bounds strategy as it only updates the scene bounds, which is a negligible increase in the construction time. The scenes used are listed in Table 4

	Teapot	White Oak	Bunny	Sponza	Chestnut	Conference
#Primitives	16k	37k	144k	262k	317k	330k
In EMC [16]	No	No	No	Yes	No	Yes
	Dragon	Buddha	Levi	Hairball	San Miguel	Powerplant
#Primitives	871k	1,087k	1,710k	2,880k	9,981k	12,759k
In EMC [16]	No	No	No	Yes	Yes	Yes

Table 4: List of all standard scenes used to measure build times, with the number of primitives per scene, and whether the scene was also in the EMC paper [16]

To find the optimal importance value λ of the Stretch Factor for Binning, we will run Binning 3 times over the different scenes with different values for the importance of the Stretch Factor. We will measure the change in tracing performance for 7 seconds over the different ray types in different game scenes listed in Table 5. To measure how well Mipping and Binning from Sections 3.2.2 and 3.2.3 respectively deal with large scenes that either do or do not contain distant primitives, we will inject a distant triangle into game scenes to grow the scene bounds, which creates a worse TLAS. We will multiply the scene's size by a factor of 2^e, where e is the extension value, along a single axis and all three axes, which essentially removes e-bits of precision along an axis for EMC but should be less for Mipping and Binning. We will compare our codes with the 64-bit versions of EMC, but we will not use size bits for EMC as we are only interested in improving the code for the position. Mipping will use an occupation grid of 9 bits (512 cells), and Binning will use 7 bits (128 cells). The game scenes used for this raw tracing performance data are listed in Table 5.

Game	Game Studio	Publisher
Cyberpunk 2077 [1]	CD Project Red	CD Project Red
Control [36]	Remedy Entertainment	Remedy Entertainment
Deathloop [37]	Arkane Studios	Bethesda
Dying Light 2 [38]	Techland	Techland
Plague Tail Requiem [39]	Asobo Studio	Focus Entertainment
Port Royal (3D Mark) [40]	UL Solutions	UL Solutions
Resident Evil [41]	Capcom	Capcom

Table 5: List of all scenes used for raw tracing performance data.

As a frame in a game consists of both a construction phase and a tracing phase, we will also measure the performance within some games that support ray tracing and have a benchmark. Table 6 lists the games used for this data.

Game	Game Studio	Publisher
Cyberpunk 2077 [1]	CD Project Red	CD Project Red
Forza Horizon 5 [42]	Playground Games	Xbox Game Studios
Port Royal (3D Mark) [40]	UL Solutions	UL Solutions

Table 6: List of all real-time applications used for full frame performance data.

For this experiment, we will run our tests on a single test machine with the following hardware:

- CPU: AMD Ryzen 7 7700
- GPU: AMD RX 7800XT (Navi 3)
- RAM: 32 GB DDR5
- OS: Windows 11

However, we can only provide numbers that are relative to existing methods. Due to an NDA signed between the author and AMD, these can be differences in build or render times in percentages, not absolute time or frames per second.

4.1 Added Construction Time

From Table 7, we can see that both Mipping and Binning create some overhead in the construction time. For both Mipping and binning, the relative construction time decreases when the number of primitives in the scene increases because the relative size of the Morton Code generation phase compared to the LBVH construction phase becomes smaller for larger scenes. Interestingly, both scenes seem to struggle more with the White Oak and Bunny scenes. The compactness of both scenes is likely a factor for this result, but we are unsure what the root cause is.

For Mipping, the construction time logically increases when b increases. We also see a significant increase in construction time when using $b = 12$, likely because the memory significantly increased and more random accesses are necessary, which is terrible for caching. A good balance between information and construction time seems to be $b = 9$, as you have eight times more cells than $b = 6$ but only 4-6% more construction time.

For Binning, the construction time is significantly higher when using a larger q, which also follows from the analysis in Section 3.2.3, which stated that the runtime complexity is $O(g + t + 2^{q+1} + qa)$. We see that the difference in relative construction time between $q = 2$ and $q = 4$ is roughly 25%, while between $q = 4$ and $q = 6$ this is roughly 40%. Therefore, $q = 4$ should be a good balance between better splitting and construction time. We also see that the build time linearly increases as a increases, which also follows from the analysis, but in some cases, it only increases after $a = 4$ (such as for teapot BIN($q = 4$)). The reason for this could be that the differences in information create a different splitting in which more threads can exit earlier, but we are unsure whether this is the root cause.

	Teapot	White Oak	Bunny	Sponza	Chestnut	Conference	Dragon	Buddha	Levi	Hairball	San Miguel	Powerplant
MIP (b=6)	17.63	22.1	24.54	18.14	18.66	17.87	12.1	10.52	11.44	11.46	11.06	11.16
MIP (b=9)	21.42	26.37	28.83	23.41	24.34	23.7	16.67	15.22	16.84	16.72	16.04	16.39
MIP (b=12)	50.73	56.44	65.47	61.25	61.25	60.67	49.38	45.38	47.36	43.67	42.04	43.51
BIN (q=2) (a=0)	38.89	41.72	43.36	38.64	38.95	37.89	30.26	26.97	27.98	25.51	24.08	25.84
BIN (q=2) (a=1)	39.72	42.7	44.52	39.39	40.06	39.67	30.31	27.71	28.8	26.29	24.79	26.72
BIN (q=2) (a=2)	40.04	43.39	45.64	40.37	41.54	40.92	31.25	28.63	29.67	27.24	25.63	27.57
BIN (q=2) (a=3)	40.38	44.08	45.99	41.92	42.32	41.51	32.22	29.5	30.46	27.5	26.38	28.37
BIN (q=2) (a=4)	41.54	45.16	47.01	42.46	43.47	42.69	33.16	30.38	31.38	28.36	27.46	28.87
BIN (q=2) (a=5)	42.36	45.81	47.83	43.79	44.59	44.25	33.93	31.17	32.21	29.74	28.24	29.63
BIN (q=2) (a=6)	43.18	46.45	48.91	44.78	45.47	44.99	34.91	32.03	32.97	30.55	28.95	30.41
BIN (q=2) (a=7)	43.6	47.32	49.74	45.78	46.62	45.9	35.88	32.94	33.92	31.75	29.75	31.25
BIN (q=2) (a=8)	44.04	48.22	50.9	46.81	47.66	46.71	36.76	33.77	34.78	32.59	30.52	31.99
BIN (q=2) (a=9)	44.84	48.85	51.83	47.9	48.91	47.82	37.73	34.67	35.66	33.45	31.3	32.78
BIN (q=4) (a=0)	59.03	61.11	71.68	70.0	70.19	69.33	57.37	51.95	52.96	49.63	45.53	48.7
BIN (q=4) (a=1)	58.35	61.7	72.11	70.76	71.2	70.27	57.39	52.91	53.8	49.89	46.4	49.38
BIN (q=4) (a=2)	60.25	63.83	73.41	72.03	72.47	71.38	58.58	54.2	55.12	51.13	47.56	50.54
BIN (q=4) (a=3)	59.95	64.8	75.78	73.55	74.78	72.25	59.84	55.21	55.9	52.17	48.48	51.52
BIN (q=4) (a=4)	61.07	65.68	76.79	75.02	75.28	73.43	60.99	56.33	56.96	53.18	49.51	52.62
BIN (q=4) (a=5)	62.08	66.73	78.07	76.66	76.77	74.98	62.39	57.38	58.14	54.34	50.48	53.54
BIN (q=4) (a=6)	63.56	67.95	79.22	77.84	78.04	75.54	63.5	58.47	59.14	55.34	51.38	54.55
BIN (q=4) (a=7)	63.87	68.44	80.63	78.12	79.39	77.35	64.61	59.55	60.13	56.37	52.37	55.47
BIN (q=4) (a=8)	65.7	69.85	81.88	79.95	81.0	78.67	65.8	60.63	61.34	57.42	53.37	56.52
BIN (q=4) (a=9)	66.73	71.06	83.28	81.48	82.59	79.84	66.96	61.71	62.49	58.53	54.35	57.56
BIN (q=6) (a=0)	92.9	99.95	120.79	120.62	122.35	116.38	101.65	92.59	94.6	89.57	77.79	82.87
BIN (q=6) (a=1)	95.42	101.36	121.87	122.95	124.27	118.48	102.46	95.03	95.97	90.84	79.57	84.03
BIN (q=6) (a=2)	95.44	103.2	124.32	124.11	126.93	121.5	104.26	96.32	97.63	92.41	80.86	85.3
BIN (q=6) (a=3)	95.26	104.41	127.15	126.53	128.3	123.17	105.65	97.53	99.06	93.92	82.13	87.07
BIN (q=6) (a=4)	96.66	105.77	128.87	127.39	130.38	120.78	107.49	99.09	100.49	95.2	83.15	88.12
BIN (q=6) (a=5)	97.84	106.57	130.78	129.75	131.62	125.96	109.25	100.56	101.96	96.53	84.42	89.39
BIN (q=6) (a=6)	100.41	108.04	132.01	130.77	133.31	127.68	110.79	101.84	103.52	98.02	85.65	90.61
BIN (q=6) (a=7)	101.74	109.27	134.25	132.42	135.65	125.65	112.34	103.31	104.98	99.41	86.73	91.76
BIN (q=6) (a=8)	100.76	110.63	135.5	135.09	137.5	126.99	113.98	104.73	106.29	100.82	88.06	93.11
BIN (q=6) (a=9)	101.67	111.98	137.35	136.35	139.37	128.31	115.65	106.36	107.93	102.36	89.31	94.9

Table 7: Shows relative construction times to LBVH with EMC when using our new Mipping (MIP) and Binning (BIN) strategies from Section 3.2.2 and 3.2.3 respectively. The left column shows the different parameters for our methods. b is the number of bits used for the occupation grid in Mipping. q is the split depth, and a is the number of bits used for the approximate count in Binning.

4.2 Different Parameters Binned Splitter

4.2.1 Visualization of Binned Splitter

From Figure 12, we can see that the Binned splitter can significantly reduce the amount of space in the initial splits of LBVH with EMC in cases where objects have an L- or T-shape. In the case of the Bunny, it correctly recognizes that the top-right is empty and, therefore, can reduce the size by splitting the ears and the head instead of having a small piece of the right ear with a small piece of the main body. For the Lamp, we see that Binning splits the head and the base of the Lamp first, and then both sections are divided further without much overlap, which is much better than the initial four splits of LBVH with EMC.

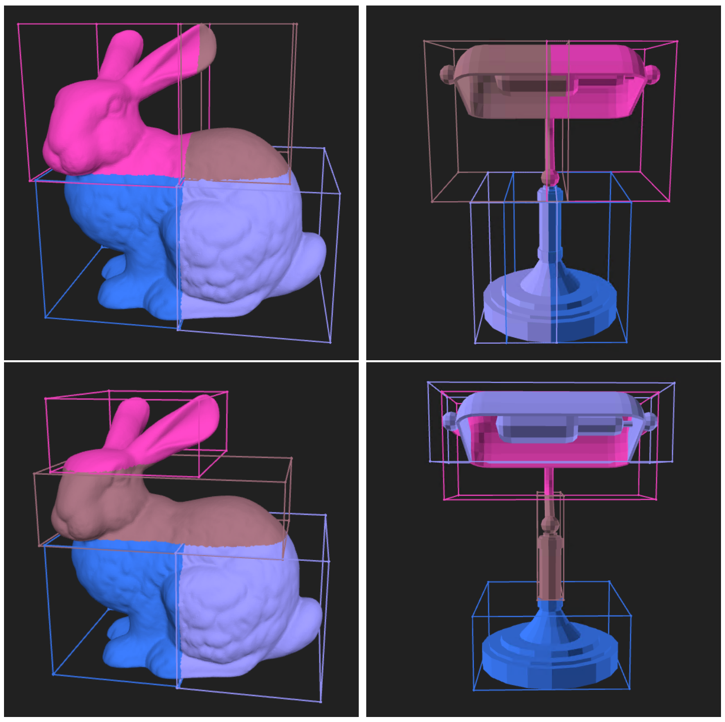

Figure 12: Shows the difference in splits when using standard LBVH with EMC (top) and the Binned Splitter (bottom) for the Stanford Bunny scene and a Lamp.

4.2.2 Performance Data of Binning

From the Tables 8, 9, and 10, we see that the Binned SAH splitter has a mostly positive effect on the rendering performance of almost all ray types except for Soft Shadow Rays, shown in Table 11 in Cyberpunk, Plague Tail Requiem, and Port Royal. Control and Deathloop show mixed results, while Dying Light and Resident Evil show a drop in relative performance.

However, the data does not reveal an optimal value for all games. For instance, for Cyberpunk, Plague Tail Requiem, and Port Royal, we can see that an importance value for the Stretch Factor of

$\lambda = 1/8$ shows a good improvement for all different ray types. However, Control benefits more from $\lambda = 1/2$ in its views, showing that Control benefits more from more square boxes than Cyberpunk, Plague Tail Requiem, and Port Royal, which might also be related to the fact that Control benefits less from our method. Therefore, if a developer wants to use our method inside their game, they should adequately investigate what value of λ works for their game by testing many different views inside a scene. This result differed from what we hoped for, but we will use the value of $\lambda = 1/8$ for the other experiments.

As the data in the Tables 8, 9, 10, and 11 does not include build times, and the additional build times from Table 7 in Section 4.1 shows that the relative build time is worse for smaller objects, we think that these gains will not hold up against the loses in additional build times. If this is the case, this will be shown in Section 4.5.

	1	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256	1/512	1/1024	1/2048
Cyberpunk V0	9.47	4.16	4.66	6.8	15.12	15.06	14.93	14.99	14.99	15.0	14.91	14.93
V1	7.08	3.04	2.85	2.46	6.27	6.28	6.16	6.1	6.36	5.61	5.67	5.69
V2	0.3	3.24	3.03	4.24	4.25	4.47	4.35	4.34	4.22	4.27	4.29	4.21
V3	13.07	5.18	5.33	6.69	13.16	13.05	12.95	12.96	12.67	11.43	11.34	11.34
V4	2.13	4.53	4.88	6.62	4.46	4.26	4.26	4.14	4.21	3.54	3.44	3.49
Control V0	12.46	11.98	-1.98	-1.84	-1.92	-0.72	0.04	-0.01	0.07	-0.01	0.06	0.04
V1	13.82	10.18	2.68	3.73	3.82	2.07	1.96	1.21	1.26	1.31	1.18	1.33
V2	-3.24	-2.78	3.66	5.84	6.07	8.57	8.78	8.53	8.71	8.73	8.7	8.61
V3	-7.05	-10.89	-8.79	-7.67	-7.82	-7.79	-7.66	-7.71	-7.71	-7.73	-7.81	-7.67
V4	-12.71	-7.4	-12.63	-12.18	-12.11	-9.49	-9.34	-9.45	-9.38	-9.41	-9.42	-9.42
Deathloop V0	-1.05	2.41	1.29	3.79	0.18	1.45	2.61	3.5	3.63	2.44	2.31	2.36
V1	-2.19	2.49	1.95	-1.05	-3.4	-3.79	-4.79	-4.83	-4.89	-4.72	-4.79	-4.84
V2	-1.42	-0.81	2.37	14.57	6.46	6.32	6.37	3.71	3.75	3.7	3.96	3.66
V3	-1.74	3.24	3.9	-2.45	-2.39	-3.07	-3.08	-2.96	-2.67	-2.75	-2.96	-2.94
V4	2.21	-3.71	-8.84	7.83	7.22	7.16	7.19	6.55	6.46	6.55	6.47	6.58
V5	-4.94	0.96	5.8	6.32	5.1	4.22	2.86	1.85	1.71	1.9	1.82	1.82
V6	-1.12	-0.91	2.36	-3.88	0.15	0.1	-0.11	-0.12	-0.08	-0.04	-0.13	-0.17
V7	1.98	-5.92	-8.29	2.55	-3.13	-3.24	-3.15	-3.25	-3.22	-3.18	-3.25	-3.31
Dying Light V0	-0.46	-0.32	-0.65	-1.0	-2.54	-2.51	-2.53	-2.57	-2.83	-2.76	-2.62	-2.59
V1	-7.87	-5.34	-5.38	-5.4	-3.71	-3.76	-3.88	-3.84	-4.88	-5.07	-5.11	-5.19
V2	2.92	3.88	4.05	4.12	4.02	-0.01	0.04	0.06	-1.02	-0.87	-0.79	-0.82
V3	-8.43	-9.01	-8.69	-9.78	-8.73	-7.18	-7.35	-7.23	-8.43	-8.69	-8.64	-8.69
Plague Tail V0	2.04	1.98	-2.95	-1.11	-1.14	-0.66	-0.45	-1.18	-1.21	-1.14	-1.42	-1.35
V1	-0.46	-1.0	-6.84	-2.69	-2.57	-2.73	-2.5	-4.45	-4.45	-4.38	-5.26	-5.18
V2	-0.35	-0.38	0.05	3.21	3.4	3.0	2.84	-0.38	-0.33	-0.42	-0.52	-0.38
V3	6.19	5.83	-0.93	1.61	1.64	1.55	1.45	-0.29	-0.19	-0.32	-0.35	-0.52
V4	2.44	2.19	-5.68	0.52	0.46	-0.29	0.56	-1.25	-1.23	-1.27	-1.33	-1.25
V5	-2.06	-2.24	-8.43	4.36	4.39	3.82	3.93	-2.02	-1.65	-1.58	-1.71	-1.62
Port Royal V0	-0.01	2.05	2.35	5.28	6.61	7.44	7.0	6.98	6.79	6.91	6.68	6.84
V1	1.64	0.99	1.4	6.07	4.48	4.55	4.03	2.31	1.57	1.7	1.46	1.62
V2	-1.05	6.69	5.93	3.71	4.02	2.5	1.86	1.95	1.87	2.07	1.7	1.89
V3	2.83	2.94	1.32	3.0	3.78	2.89	2.85	1.78	1.18	1.06	1.08	1.09
V4	6.33	6.36	5.79	9.12	8.33	8.47	9.12	8.58	8.43	8.18	8.38	8.27
V5	0.41	1.4	3.15	4.46	4.14	3.28	2.25	1.58	1.62	1.65	1.57	1.67
Resident Evil V0	0.7	0.48	-2.04	1.93	-2.56	-5.99	-5.81	-4.62	-4.89	-4.86	-4.98	-4.9
V1	-7.91	-7.31	-5.05	-2.05	-3.98	-6.75	-6.63	-6.94	-7.2	-7.11	-7.2	-7.07
V2	25.23	21.94	25.46	9.65	17.03	7.02	6.71	8.59	8.5	8.58	8.49	8.64
V3	1.17	4.28	5.96	4.79	-1.7	-4.57	-4.39	-4.65	-4.61	-4.62	-4.71	-4.67

+30

-30

Table 8: Shows the relative performance improvement/decrease of Primary Rays when using the Binned SAH splitter for different values of the importance of the Stretch Factor. The top row shows the importance factor λ. For many games and scenes, the rendering performance increases when using Binning (highlighted in green), while some scenes also show a performance decrease (highlighted in red). See Section 4.2.2.

	1	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256	1/512	1/1024	1/2048
Cyberpunk V0	5.37	10.34	10.67	10.91	8.84	9.11	9.9	5.98	7.37	7.46	7.59	7.15
V1	4.51	8.63	8.19	10.54	5.31	7.09	8.42	7.21	5.8	5.08	5.54	4.91
V2	8.65	9.88	9.39	11.3	11.91	12.42	12.2	9.99	11.06	11.06	9.85	10.2
V3	9.01	10.35	10.96	11.13	12.51	12.9	11.05	13.17	11.77	12.05	11.34	11.47
V4	9.64	8.5	7.58	11.31	9.5	8.3	9.73	8.95	8.69	8.5	7.56	9.16
Control V0	7.53	8.87	3.46	3.55	3.45	4.13	4.38	4.32	4.37	4.35	4.37	4.43
V1	8.42	12.2	8.03	8.75	8.71	7.85	7.79	7.67	7.67	7.78	7.74	7.73
V2	-1.8	-0.89	2.59	3.23	3.7	4.3	4.64	4.47	4.66	4.59	4.58	4.55
V3	-2.89	-3.73	0.09	0.22	0.13	-1.57	-1.54	-1.35	-1.48	-1.52	-1.55	-1.47
V4	-6.11	-2.59	-7.26	-6.8	-6.55	-1.0	-0.86	-0.79	-0.78	-0.63	-0.57	-0.6
Deathloop V0	3.74	6.24	7.03	7.03	8.04	7.07	7.89	4.52	4.9	5.46	4.04	3.29
V1	-0.02	2.82	2.89	0.42	0.99	0.3	0.57	0.02	0.57	0.67	0.35	0.42
V2	2.82	2.62	2.06	6.28	0.74	0.12	0.66	-1.4	-0.07	-1.03	-0.59	1.81
V3	6.5	11.14	3.24	13.88	-1.46	-6.41	-2.97	-4.58	-5.39	-4.55	-5.98	-5.36
V4	-8.94	-2.67	-3.95	3.08	-3.91	-3.26	-4.95	-5.99	-4.95	-8.11	-3.33	-4.47
V5	8.86	4.74	5.96	6.61	8.89	8.57	9.54	5.16	7.35	5.61	6.41	3.54
V6	13.41	5.11	3.08	2.98	11.58	6.91	7.38	7.89	6.4	7.18	7.96	8.06
V7	7.39	5.04	0.43	-6.33	0.98	7.27	4.89	3.52	4.93	1.45	0.16	1.88
Dying Light V0	-0.71	-0.55	-0.32	-1.0	-0.62	-1.46	-1.42	-1.52	-2.27	-1.75	-1.55	-1.52
V1	2.9	3.13	2.63	2.32	3.67	4.02	3.75	4.1	2.47	2.67	2.86	3.17
V2	4.29	4.77	4.92	1.29	4.53	6.08	5.37	5.19	3.24	5.61	5.11	5.29
V3	-1.55	-0.84	-0.71	-1.31	-1.48	0.37	0.94	0.81	0.0	0.2	0.3	-0.47
Plague Tail V0	-2.94	-2.58	-3.13	1.17	1.96	0.8	1.35	-4.6	-3.25	-3.56	-3.74	-2.88
V1	-3.84	-3.84	-4.28	2.77	1.83	1.89	2.83	-1.7	-2.96	-3.72	-3.15	-3.09
V2	0.79	0.04	-11.41	-4.45	-2.07	-3.83	-2.86	-6.43	-6.52	-7.84	-4.89	-5.55
V3	-1.59	-3.23	-6.68	3.34	3.72	0.82	0.11	-3.45	-2.9	-3.61	-0.38	-3.34
V4	-2.72	-3.7	-6.23	1.18	0.49	-1.57	-0.92	-3.05	-2.72	-4.59	-3.67	-3.51
V5	0.1	0.03	-2.55	-1.38	-1.1	-0.92	-0.79	-0.99	-0.84	-0.87	-0.87	-1.02
Port Royal V0	6.09	6.96	6.85	7.26	7.57	7.6	8.3	6.76	6.15	6.93	5.9	7.24
V1	1.1	1.46	2.69	4.73	4.48	3.68	3.94	1.82	2.99	3.07	3.05	3.1
V2	2.18	4.26	3.36	4.14	2.77	4.11	4.14	2.96	4.17	2.55	4.01	3.55
V3	4.43	5.89	5.77	7.44	7.08	6.15	6.35	6.5	5.46	6.53	6.2	5.49
V4	5.71	6.47	4.78	6.11	6.17	6.17	6.93	6.2	5.71	7.2	6.88	6.39
V5	-1.75	-1.77	-0.63	0.5	0.46	-0.13	-0.57	-1.61	-1.43	-1.38	-1.43	-1.39
Resident Evil V0	-0.66	-1.27	-2.61	-4.62	-5.36	-7.1	-7.72	-6.51	-7.08	-6.64	-6.85	-7.04
V1	1.35	2.94	3.98	3.88	2.1	-0.18	0.06	-0.37	-0.16	-0.06	-0.14	-0.45
V2	3.24	3.21	7.11	4.43	5.42	4.55	4.66	6.29	6.61	6.51	6.56	6.62
V3	0.43	2.07	3.25	2.7	1.62	-0.63	-0.81	-0.79	-0.81	-0.71	-0.57	-0.75

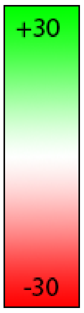

Table 9: Shows the relative performance improvement/decrease of Global Illumination Rays when using the Binned SAH splitter for different values of the importance of the Stretch Factor. The top row shows the importance factor λ. For many games and scenes, the rendering performance increases when using Binning (highlighted in green), while some scenes also show a performance decrease (highlighted in red). See Section 4.2.2.

	1	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256	1/512	1/1024	1/2048
Cyberpunk V0	0.53	1.55	2.05	3.58	2.07	1.76	2.16	1.77	2.12	1.12	1.3	1.1
V1	1.49	2.86	3.02	3.36	1.56	1.45	1.92	1.57	2.27	0.96	0.44	0.83
V2	1.75	1.78	1.83	3.75	3.68	3.93	3.58	3.59	3.65	2.2	2.77	2.34
V3	4.93	1.97	2.38	3.03	6.54	6.54	5.94	6.74	6.1	5.09	4.66	5.47
V4	2.62	3.24	3.75	5.76	2.79	2.9	3.05	3.11	2.98	1.99	1.99	2.13
Control V0	6.67	7.32	3.22	3.34	3.21	3.13	3.25	3.05	3.13	3.16	3.44	3.38
V1	6.48	6.97	5.61	5.77	5.98	4.51	4.36	4.26	4.17	4.21	4.28	4.33
V2	-5.08	-4.91	-0.57	-0.93	-0.42	-0.5	0.09	-0.03	0.13	0.11	0.05	0.08
V3	-10.38	-12.18	-10.56	-11.0	-11.0	-11.92	-11.83	-11.85	-11.91	-11.94	-11.94	-11.87
V4	-8.15	-3.63	-5.45	-5.37	-5.37	-5.83	-5.98	-5.79	-5.64	-5.69	-5.74	-5.85
Deathloop V0	5.75	4.32	4.73	4.08	5.22	3.06	4.62	1.41	1.63	2.8	0.53	0.92
V1	-1.26	1.41	1.77	0.62	-2.71	-2.97	-3.04	-3.31	-3.04	-2.98	-3.17	-3.1
V2	-0.03	0.46	0.52	0.32	-1.4	-1.8	-1.54	-2.54	-1.32	-2.91	-2.21	-1.16
V3	4.94	5.66	2.72	-1.54	1.39	-0.69	0.89	-0.14	-1.42	0.03	-0.69	-0.3
V4	-4.69	-4.42	-4.63	-1.75	-2.39	-1.1	-3.27	-3.7	-3.31	-5.7	-1.24	-2.42
V5	1.98	0.16	1.13	0.25	2.19	2.0	2.23	0.21	1.22	-0.25	0.35	-1.34
V6	2.0	-0.92	-3.44	-2.6	1.26	-2.31	-1.79	-2.23	-3.32	-1.98	-1.84	-0.49
V7	7.5	-0.67	-2.62	0.38	0.24	3.79	1.51	0.82	2.83	0.07	-0.81	0.14
Dying Light V0	-2.58	-1.73	-1.69	-2.37	-1.73	-2.67	-2.58	-2.54	-3.19	-2.97	-2.88	-2.84
V1	-1.89	-1.12	-1.32	-1.3	-0.83	-1.38	-1.47	-1.34	-2.42	-2.2	-2.16	-1.93
V2	-2.51	-2.51	-1.95	-2.15	-1.79	-2.83	-2.68	-2.83	-3.75	-3.36	-3.29	-3.24
V3	-2.54	-1.58	-1.48	-1.42	-1.46	-2.42	-2.32	-2.3	-2.54	-2.54	-2.44	-2.62
Plague Tail V0	3.75	3.72	3.98	5.31	5.2	5.26	5.74	5.34	5.43	5.31	5.06	5.26
V1	4.52	4.24	3.68	5.05	4.85	5.35	5.49	6.64	6.58	6.64	5.83	5.77
V2	1.55	1.29	1.38	2.11	1.89	2.9	2.84	3.18	3.26	3.26	3.18	3.07
V3	2.7	2.59	2.7	3.31	3.37	3.87	3.87	4.09	4.03	3.78	3.95	3.84
V4	-1.24	-0.95	0.19	1.0	1.05	0.9	1.31	0.95	1.12	1.0	0.69	1.02
V5	-0.81	-1.1	-1.1	1.6	1.71	1.02	1.3	1.42	1.6	1.75	1.42	1.52
Port Royal V0	0.57	1.78	2.94	3.18	3.41	3.45	3.72	3.74	3.15	3.51	3.72	3.45
V1	2.64	2.42	3.7	6.9	6.76	5.92	6.48	4.55	5.24	5.36	5.45	5.31
V2	-1.0	0.85	-0.49	0.2	0.12	0.94	0.43	0.09	0.84	0.06	0.71	0.52
V3	0.25	1.3	2.04	4.42	4.25	2.83	3.9	3.21	3.0	3.02	3.31	2.8
V4	-1.1	1.23	2.36	2.55	2.45	2.21	3.03	2.94	2.58	2.51	2.54	2.07
V5	-1.89	-1.94	-4.03	-2.87	-3.18	-3.4	-3.6	-4.35	-4.32	-4.32	-4.36	-4.29
Resident Evil V0	-5.7	-7.69	-6.46	-7.45	-7.93	-9.53	-9.7	-7.83	-8.19	-8.04	-8.07	-8.39
V1	-2.58	-1.85	-0.5	0.56	-1.33	-3.42	-3.14	-3.14	-3.13	-3.64	-3.25	-3.59
V2	1.81	1.76	3.47	1.99	1.44	0.34	0.46	1.67	1.03	1.87	1.83	1.64
V3	-2.86	-2.35	-0.55	-0.31	-2.07	-3.98	-3.69	-3.62	-3.75	-3.77	-3.69	-3.62

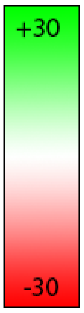

Table 10: Shows the relative performance improvement/decrease of Ambient Occlusion Rays when using the Binned SAH splitter for different values of the importance of the Stretch Factor. The top row shows the importance factor λ. For many games and scenes, the rendering performance increases when using Binning (highlighted in green), while some scenes also show a performance decrease (highlighted in red). See Section 4.2.2.

	1	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256	1/512	1/1024	1/2048
Cyberpunk V0	-7.59	13.96	13.88	13.16	-10.17	-10.32	-10.27	-10.31	-10.2	-10.32	-10.22	-10.31
V1	-4.04	13.52	13.46	13.14	-6.26	-6.34	-6.43	-6.79	-6.49	-6.75	-6.71	-6.73
V2	-4.69	14.93	14.98	14.6	-7.75	-7.87	-7.85	-7.87	-7.82	-7.87	-7.84	-7.89
V3	-5.06	14.84	14.84	14.71	-8.99	-9.13	-9.22	-9.21	-9.2	-9.16	-9.14	-9.19
V4	-5.72	15.05	15.14	14.96	-8.98	-9.07	-9.13	-9.13	-8.97	-9.09	-9.1	-9.13
Control V0	-23.93	-23.56	-27.02	-26.91	-26.81	-3.18	-3.23	-3.14	-3.16	-3.1	-3.18	-3.22
V1	-20.16	-20.2	-17.72	-17.04	-17.1	-12.7	-12.76	-12.86	-12.83	-12.79	-12.85	-12.82
V2	40.83	40.37	26.03	28.64	28.6	19.54	19.6	19.51	19.65	19.67	19.54	19.64
V3	-6.61	-7.86	2.06	1.76	1.77	2.08	2.14	2.01	2.1	2.06	2.11	2.03
V4	9.66	5.97	17.43	19.44	19.53	0.7	0.78	0.62	0.83	0.66	0.66	0.66
Deathloop V0	-1.38	1.25	1.71	-3.62	-2.11	-1.05	-0.03	-0.13	-0.59	-0.4	-0.66	-0.99
V1	1.13	-2.61	-3.83	-0.55	-18.69	-19.2	-19.45	-19.34	-19.37	-19.23	-19.32	-19.29
V2	2.26	2.89	2.66	0.26	-2.84	-2.56	-3.05	-3.05	-3.64	-3.29	-3.08	-3.1
V3	0.26	1.11	1.81	0.37	-5.01	-4.64	-4.37	-5.3	-4.73	-5.65	-5.12	-4.9
V4	-2.38	-1.5	0.36	7.8	-3.57	-3.82	-3.64	-4.08	-4.19	-4.24	-4.24	-4.44
V5	2.78	3.1	2.78	0.37	1.89	1.98	1.38	1.38	0.95	1.66	1.23	0.98
V6	1.66	0.21	0.86	-0.62	0.43	-0.05	-0.32	-0.48	0.11	0.0	-0.67	-0.43
V7	-1.01	-3.84	-16.55	-1.77	-6.11	-6.27	-5.99	-6.35	-6.05	-6.37	-6.09	-6.31
Dying Light V0	3.04	3.74	3.97	4.34	3.92	3.78	3.55	3.5	3.88	3.97	3.88	3.97
V1	25.26	26.29	27.28	27.84	27.96	12.59	12.5	12.37	23.8	23.93	23.71	23.67
V2	2.14	2.77	2.77	2.17	2.87	3.89	4.18	4.13	4.52	4.57	4.41	4.7
V3	6.09	6.24	6.96	7.59	7.25	6.09	6.19	6.14	10.35	10.25	10.3	10.35
Plague Tail V0	-2.09	-2.46	1.96	2.37	2.14	0.68	1.14	4.05	4.05	4.0	3.69	3.87
V1	1.91	1.37	0.82	0.27	0.23	-0.64	-0.73	1.41	1.41	1.46	0.59	0.68
V2	1.76	1.65	-0.96	-2.1	-2.26	-2.03	-2.07	0.77	0.61	0.54	0.38	0.38
V3	-0.86	-1.29	-0.67	-0.82	-0.94	-1.69	-1.49	1.8	1.61	1.84	1.53	1.73
V4	-1.19	-0.97	-3.46	-2.95	-3.31	-3.6	-3.21	0.36	0.4	0.29	0.36	0.32
V5	-0.78	-0.57	-3.28	-6.63	-6.66	-7.13	-7.06	0.18	0.71	0.75	0.82	0.78
Port Royal V0	-0.02	-0.61	0.88	1.15	2.06	2.11	2.18	1.99	1.4	1.25	1.45	1.62
V1	-0.85	0.17	-1.07	0.11	-0.56	-0.64	-0.89	-1.42	-1.59	-1.56	-1.62	-1.59
V2	4.52	-13.51	-14.34	-10.72	-13.46	-13.67	-15.36	-15.7	-15.95	-15.03	-16.03	-15.15
V3	1.51	1.73	2.64	3.73	4.07	4.54	3.54	3.28	3.11	3.05	3.09	3.11
V4	-0.06	-0.15	2.13	2.64	3.21	3.18	3.69	3.0	2.7	2.73	2.64	2.82
V5	5.7	6.08	-0.62	0.35	0.25	0.34	0.07	-0.37	-0.24	-0.31	-0.35	-0.27
Resident Evil V0	-26.52	-21.62	-25.73	-26.4	-16.57	-11.36	-11.24	-12.75	-15.34	-15.24	-15.19	-15.13
V1	-12.71	2.65	1.22	3.2	4.02	-1.9	-1.75	-0.58	-0.78	-0.87	-0.87	-0.89
V2	-9.91	-19.54	-18.43	-16.89	-16.55	-17.56	-17.33	-17.96	-17.97	-17.9	-18.0	-17.9
V3	-16.35	2.89	0.12	1.11	3.44	-3.08	-3.02	-2.17	-2.2	-2.3	-2.37	-2.33

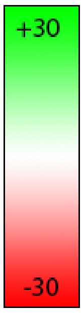

Table 11: Shows the relative performance improvement/decrease of Soft Shadow Rays when using the Binned SAH splitter for different values of the importance of the Stretch Factor. The top row shows the importance factor λ. For many games and scenes, the rendering performance increases when using Binning (highlighted in green), while some scenes also show a performance decrease (highlighted in red). See Section 4.2.2.

4.3 Growing Scene Bounds

In this Section, we will show some Figures of performance degradation when the scene bounds grow. We will only show the Figures for Cyberpunk [1] and Control [36] here, as all Figures follow the same trend. The other games' Figures can be viewed in Appendix 8.3 and 8.4.

When looking at Figure 13, 14, and all Figures in the Appendix 8.3, when growing the scene bounds over a single axis, we see a significant performance drop between $20 \leq e \leq 30$ in the standard base case, where e is the extension value. We also see that both Mipping and Binning move this performance drop by the same value as the outcome of the Equations 13 and 14 from Section 3.2.4. For Mipping, we used $b = 9$ and therefore has $move_1 = \log_2(2^b)/1 = b$, so the drop moved by 9; for Binning, we used 7 bits (128 cells), so the drop moved by 7. Furthermore, in Figure 14, we see that Control has a baseline at roughly 40% of the base performance. We think this is due to the smaller size of the TLAS in Control, so even if the TLAS is as bad as it can be, looping over all the different BLASes in random order is only 60% worse than using a TLAS. We also see some drops and bumps in performance from the Soft Shadow Rays, likely due to more beneficial splits in certain growing cases.

When looking at Figure 15, and all Figures in the Appendix 8.4, we see a significant performance drop between $10 \leq e \leq 20$, where e is the extension value, which is later than we expected. We still see that Mipping and Binning move the performance drop by the same value as the outcome of the Equation 13 from Section 3.2.4, but the benefit is three times lesser than when the bounds increase over a single axis. This is an expected outcome as $move_3$ is three times smaller than $move_1$, or more formally $move_3 = \frac{1}{3}move_1$.

Growth over a single axis

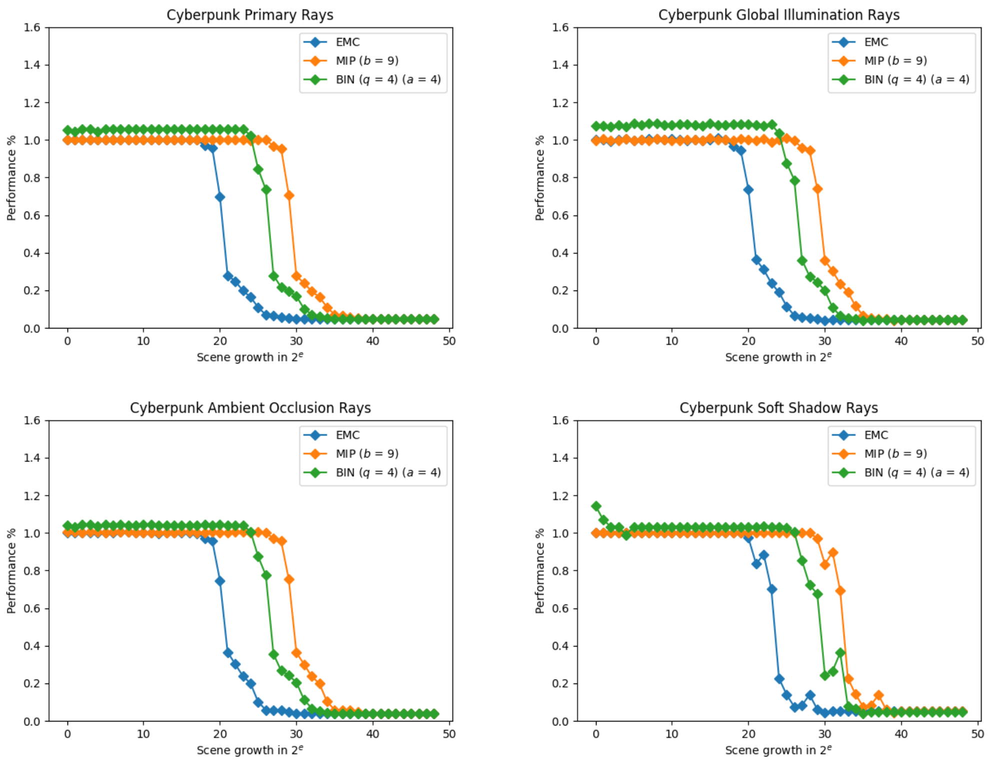

Figure 13: Graphs showing the relative performance in Cyberpunk [1] of our strategies compared to LBVH with EMC without any scene bound growth of the different methods when the scene bounds grow by 2^e in a single axis, where e is the extension value represented on the x-axis. See Section 4.3.

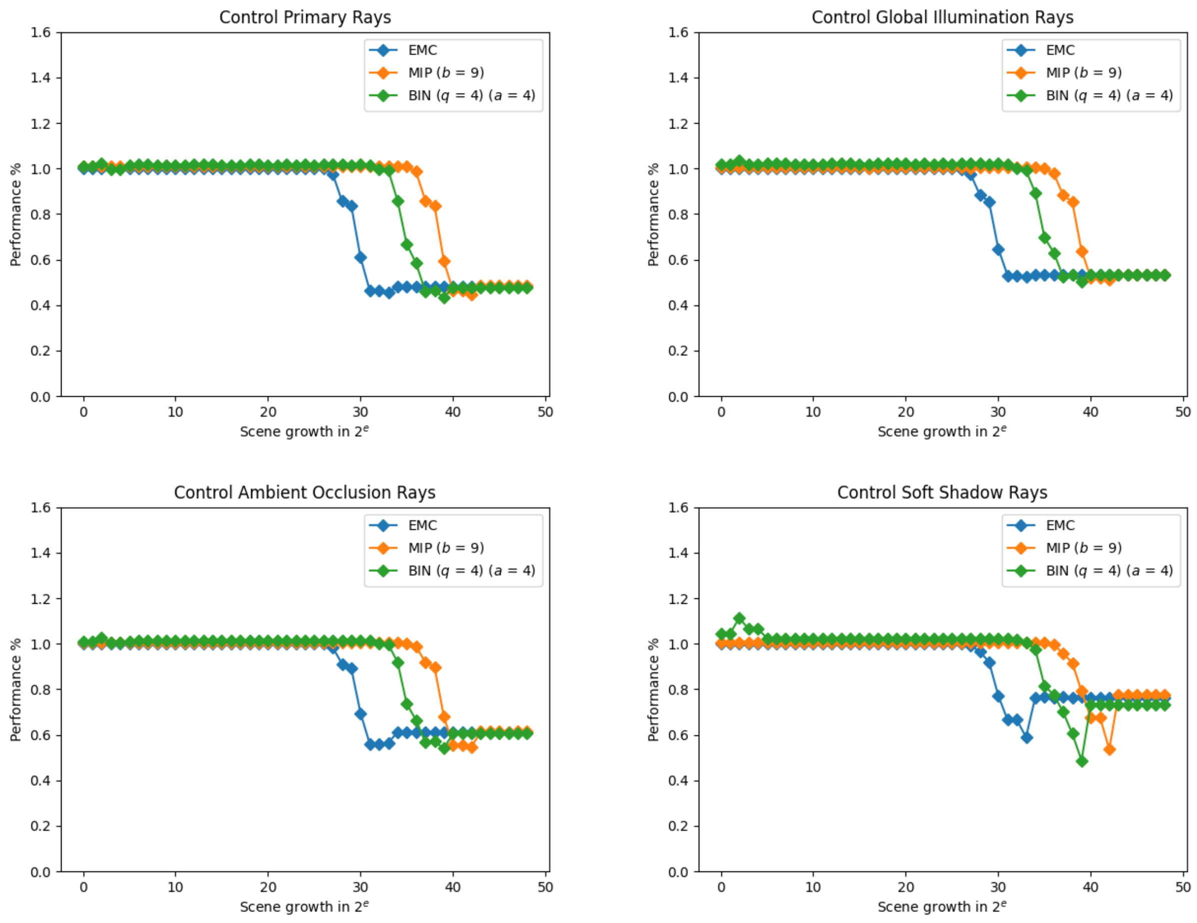

Figure 14: Graphs showing the relative performance in Control [36] of our strategies compared to LBVH with EMC without any scene bound growth of the different methods when the scene bounds grow by 2^e in a single axis, where e is the extension value represented on the x-axis. See Section 4.3.

Growth over three axes

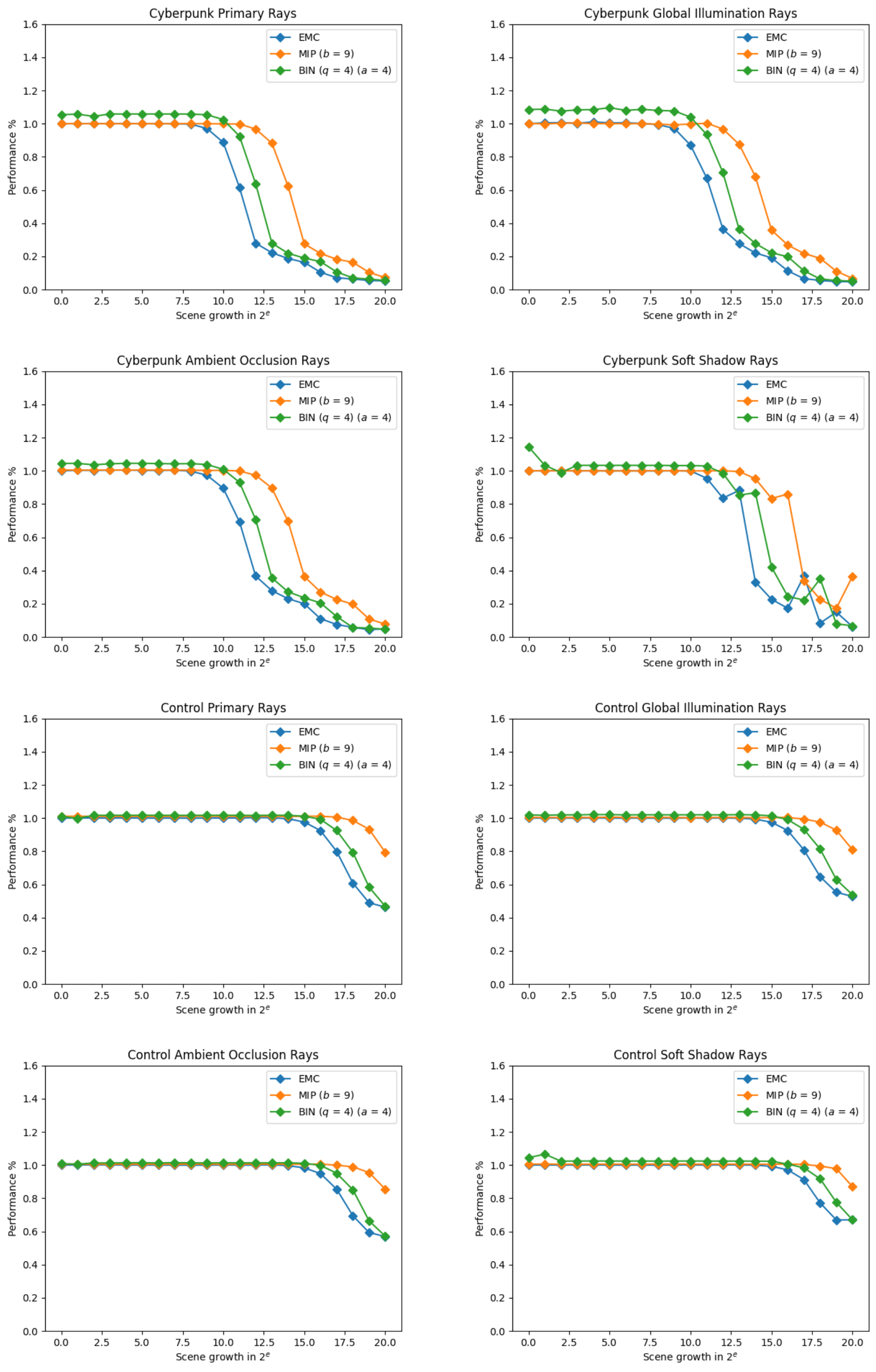

Figure 15: Graphs showing the relative performance in Cyberpunk [1] and Control [36] of our strategies compared to LBVH with EMC without any scene bound growth of the different methods when the scene bounds grow by 2^e in all three axes, where e is the extension value represented on the x-axis. See Section 4.3.

4.4 Smaller Scene Bounds

Table 12 shows an excellent general improvement for many scenes and all ray types. Port Royal and Dying Light were the only games that did not benefit from our method because these scenes do not have large objects in the game's background. However, seeing a 2-10% drop in performance in Global Illumination Rays in Port Royal was unexpected but might be related to a change in the EMC bit order when the scene bounds change. We will see if this drop in performance propagates to the Game Benchmarks in Section 4.5. All the other games seem to benefit from using these more concise scene bounds, which is a good result as the change proposed in Section 3.3 does not change the existing code much. We, therefore, suggest any game developer try this solution out for their game if they are running a ray tracer.

	PR	GI	AO	SSR
Cyberpunk V0	19.99	1.26	2.87	-4.42
V1	16.54	6.9	9.26	0.44
V2	0.89	5.81	4.72	-6.71
V3	27.94	9.09	7.16	-11.3
V4	6.86	11.25	7.39	-14.16
Control V0	3.11	3.76	4.47	1.86
V1	15.86	9.11	6.64	-3.81
V2	11.62	3.53	4.59	9.08
V3	-6.58	3.96	2.08	21.04
V4	5.15	9.7	9.92	12.16
Deathloop V0	6.58	1.64	-1.01	-2.32
V1	2.86	-0.48	-0.75	31.54
V2	-6.03	3.42	0.64	2.01
V3	-4.74	18.29	2.07	-0.02
V4	-4.3	17.04	4.13	3.18
V5	8.63	0.18	3.16	4.04
V6	-3.15	5.47	-3.56	-2.11
V7	6.76	0.96	4.19	-5.59
Dying Light V0	2.38	-1.75	-0.84	-0.46
V1	0.79	-0.6	0.48	-0.59
V2	-0.01	-4.73	0.64	-0.43
V3	-2.29	-1.43	0.38	-0.38
Plague Tail V0	-1.03	2.46	2.95	1.5
V1	-0.96	3.19	2.51	0.31
V2	-0.83	2.06	0.88	1.01
V3	1.44	3.88	2.3	0.08
V4	0.34	0.21	1.37	-0.41
V5	2.14	0.59	3.53	-2.97
Port Royal V0	2.22	-5.69	-0.1	-1.12
V1	-2.74	-3.41	0.13	-3.43
V2	-9.57	-3.91	2.21	7.22
V3	-3.98	-5.2	0.66	-2.13
V4	4.35	-1.53	-0.68	-1.51
V5	-16.8	-9.59	-3.09	29.79
Resident Evil V0	-1.06	-0.27	0.41	7.06
V1	1.79	0.04	0.05	20.22
V2	3.65	0.15	0.45	2.89
V3	-0.88	0.29	0.06	19.92

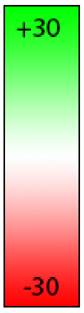

Table 12: Shows the rendering improvement of using smaller scene bounds as discussed in Section 3.3.

4.5 Game Benchmarks

First, we want to clarify the results in the Port Royal column in Tables 13 and 14. The benchmark provided by 3D Mark has a single build phase at the start of the benchmark, which is why it shows similar values for the different methods, as the benchmark does not consider the build times of the various techniques. Therefore, Binning shows a slight improvement in this benchmark.

Next, from Tables 13 and 14, we see a drop in performance for both Mipping and Binning. We expected this result as we are preventing a case that is not happening in these benchmarks, and if we were to grow the scene bounds significantly, our new codes would outperform the LBVH with EMC. However, Binning decreases the framerate of Cyberpunk [1] by a considerable margin, which we think is due to the numerous small objects in the scene, and from Section 4.1, we know that the construction time is worse for small objects. Therefore, the increase in raw tracing performance of Section 4.2.2 does not outweigh the added construction time of the method. We see a similar trend for Mipping and Binning Forza Horizon 5 [42] to a lesser degree because Forza Horizon 5 only uses ray-traced reflections on the racer's car [43].

For our scene bounds from Section 3.3, we see a good improvement of 0.4 - 0.5% on average in Cyberpunk [1] and 0.9% in Forza Horizon 5, showing that these scene bounds also work well in real-world applications.

	Cyberpunk			Forza			Port Royal
	avg	min	max	avg	min	max	avg
MIP ($b = 9$)	-3.13	-2.59	-3.14	-0.31	-1.18	0.0	0.0
BIN ($q = 4$) ($a = 4$)	-19.54	-18.43	-19.46	-0.73	-0.53	-0.83	0.33
Scene Bounds	0.41	1.57	1.3	0.92	0.73	0.92	0.22

Table 13: Shows the improvement in the Cyberpunk [1] and Forza Horizon 5 [42] games when using Mipping, Binning, and the more concise Scene Bounds from Sections 3.2.2, 3.2.3, and 3.3 respectively, with a screen size of 1920x1080.

	Cyberpunk			Forza			Port Royal
	avg	min	max	avg	min	max	avg
MIP ($b = 9$)	-3.22	-3.09	-4.11	-0.23	-0.29	-0.57	0.0
BIN ($q = 4$) ($a = 4$)	-17.36	-15.97	-17.35	-0.35	-0.43	-0.29	0.3
Scene Bounds	0.48	1.88	0.96	0.93	0.81	0.65	-0.03

Table 14: Shows the improvement in the Cyberpunk [1] and Forza Horizon 5 [42] games when using Mipping, Binning, and the more concise Scene Bounds from Sections 3.2.2, 3.2.3, and 3.3 respectively, with a screen size of 2560x1440.

5 Conclusion and Discussion

Let's restate the research question we had in Chapter 1 of the thesis:

Can we improve the Extended Morton Codes further to construct BVHs of equal or higher quality in real-time applications? In particular in situations where the scene's bounds grow out of proportion due to distant or large primitives at the extent of the scene?

We then listed a set of requirements in Chapter 3, which we repeat and discuss here for the sake of exposition:

- **Efficiently Handling Distant Primitives:** Our method should perform equal or better than Extended Morton Codes in cases where the scene bounds stretch into one or more directions.
  We assessed this by adding distant primitives to existing scenes and using both EMC and our new method to compare their respective LBVHs in terms of rendering performance.
  Mipping and Binning from Sections 3.2.2 and 3.2.3, respectively, can help in cases where the scene's bounds grow out of proportion due to distant primitives by keeping the tracing performance high for a longer time than standard LBVH with EMC.

- **Efficiently Handling Large Primitives:** The method should perform equal or better than Extended Morton Codes in cases where the scene bounds stretch because of large primitives at the edge of the scene.
  We assessed this using a large scene with large objects and with EMC and our new method to compare their respective LBVHs regarding rendering performance.
  Our new concise scene bounds from Section 3.3 can adequately improve the performance in these cases. It even shows a good improvement in Cyberpunk [1], a real-world game, where the performance increased by 0.41%. Although this improvement appears to be small, it correlates with a couple of extra frames per second in a game, which is significant according to the key stakeholder (AMD). Furthermore, developing efficient ray query implementations has been a topic of study for many years, hence it is impressive that our methods improve the state of the art.

- **Generality:** Our method should work for any scene or object, which is satisfied as all of the strategies presented work for any scene or object.

- **Low Total Build Times:** The overall build time compared to LBVH should not increase significantly, making our method feasible for real-time applications such as games.
  Our new concise scene bounds only increase the build time negligibly, as it only has a different computation for the scene bounds. Mipping increases the build times slightly when using a low b value, but it could be improved if it was possible to have bitmaps in which programmers can turn on bits without atomics. This change is possible on current day hardware if we had used a single 32-bit dword per cell instead of a single bit. However, we did not test this as this

would significantly increase the global memory requirement for the method. On the other hand, Binning significantly increases the build times of the BVH and, therefore, does not meet this requirement.

- **Black Box Behaviour:** Our new codes must be directly usable for existing construction algorithms that rely on Morton Codes without changing the construction algorithms, which we showed by using LBVH for all tests.
- **Low Global Memory Requirement**: As games use many objects in their scenes, we want to use as little global memory as possible so that the data necessary for constructing a BVH stays as low as possible.
  For the new scene bounds, this is $6*4 = 24$ bytes; for Binning, this is $128*4 = 512$ bytes, and for Mipping, we set the number of grid bits $b \leq 12$, which is at most $4,096$ bits 512 bytes.
- **Simplicity of Use:** Our new codes should be easy to use without tuning many complex parameters per scene.
  Mipping and our new scene bounds satisfy this requirement, as Mipping only requires a single parameter (b, the number of bits for the grid), and our new scene bounds don't have a parameter. However, Binning also requires tuning the importance factor λ of the stretch factor, which can differ for every game and scene. Therefore, this requirement is not satisfied for Mipping.

So, from these results, we can conclude that there is room for improvement regarding large primitives at the scene bounds, which is not uncommon in games. Using our new concise scene bounds, we can handle such cases better without significantly increasing the build times of existing methods.

However, these results conclude that Mipping and Binning might not be the best methods to optimize distant primitives. This conclusion mostly comes from the significant overhead required to generate these new codes. Therefore, we suggest only using these methods in contexts with so much space between two sections that the methods could make a difference. However, developers should realize that the two sections are so far apart that they could also consider building two BVHs for the different sections, rendering these methods obsolete.

In the future, we could investigate whether we might be able to catch multiple distant sections automatically during primitive encoding, such that we can decide to build various BVHs automatically for the different sections with their scene bounds. We can efficiently encode the split for these BVHs at the start of the EMC, so we would need to find a heuristic that can filter primitives into different sections during encoding.

Next, we could investigate adding other heuristics to the EMC code. The authors of EMC already mention this in their paper [16], but they do not hint at any possible heuristics. Possible heuristics as additions could be the distance a primitive extends into a direction (and might leave its cell) instead of a fixed one-dimensional value for the size and merge primitives in cells that overlap the same planes.

6 Acknowledgements

First and foremost, I want to thank Jacco Bikker for seeing my potential and inviting me to a meeting with AMD to see if we could work out a master's thesis. Then, I want to thank Nathan McDaniel and Sinha Pranabesh for also seeing my potential and hiring me to work at AMD. Furthermore, I want to thank John Tsakok and Jalil Ameer from AMD for helping me with all my driver and code issues, introducing me to the infrastructure of AMD, and helping me think of solutions to problems I encountered.

Next, I want to thank some people who helped by listening to my rambles about difficulties with my thesis, may it be legal or unfortunate setbacks of results. So, big thanks to Sanne de Baar, Tom Simmelink, Lisa Kwast, Nadie Smit, Iddo Riemersma, Sonja Riemerma, Michel Riemersma, Sjoerd Riemersma, Julio Rosua, and Lars de Kwant.

Lastly, I want to thank my current primary supervisor, Alexandru Telea, for pushing me to create a master's thesis that I am proud of.

7 References

[1] CD Project Red. *Cyberpunk*. Accessed in 2024. URL: `https://www.cyberpunk.net/nl/en/`.

[2] S. Woop, C. Benethin, I. Wald. "Exploiting Local Orientation Similarity for Efficient Ray Traversal of Hair and Fur". In: *Proceedings of High-Performance Graphics*. 2014, pp. 41–49.

[3] I. Wald et al. "Using Hardware Ray Transforms to Accelerate Ray/Primitive Intersections for Long, Thin Primitive Types". In: *Proceedings of the ACM on Computer Graphics and Interactive Techniques 3, 2*. 2020.

[4] I. Wald et al. "State of the Art in Ray Tracing Animated Scenes". In: *Computer graphics forum 28, 6*. 2007, pp. 1691–1722.

[5] C. Lauterbach et al. "Interactive Ray Tracing of Dynamic Scenes using BVHs". In: *Proceedings of Symposium on Interactive Ray Tracing*. 2006, pp. 39–46.

[6] C. Lauterbach et al. "Fast BVH Construction on GPUs". In: *Computer Graphics Forum 28, 2* (2009), pp. 375–384.

[7] J. Pantaleoni, D. Luebke. "HLBVH: Hierarchical LBVH Construction for Real-time Ray Tracing of Dynamic Geometry". In: *Proceedings of High-Performance Graphics*. 2010, pp. 87–95.

[8] D. Meister, J. Bittner. "Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction". In: *IEEE Transactions on Visualization and Computer Graphics 23, 3* (2017), pp. 1345–1353.

[9] B. Walter et al. "Fast Agglomerative Clustering for Rendering". In: *Proceedings of Symposium on Interactive Ray Tracing*. 2008, pp. 81–86.

[10] K. Garanzha, J. Pantaleoni, D. McAllister. "Simpler and Faster HLBVH with Work Queues". In: *Proceedings of High-Performance Graphics*. 2011, pp. 59–64.

[11] G. M. Morton. *A Computer Oriented Geodetic Database and a New Technique in File Sequencing*. Tech. rep. 1966.

[12] N.K. Govindaraju et al. "GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management". In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. 2006, pp. 325–336.

[13] A. Adinets, D. Merril. "Onesweep: A Faster Least Significant Digit Radix Sort for GPUs". In: *ArXiv* (2022).

[14] Y. Gu et al. "Efficient BVH Construction via Approximate Agglomerative Clustering". In: *Proceedings of High-Performance Graphics*. 2013, pp. 81–88.

[15] T. Kerras. "Maximizing Parallelism in the Construction of BVHs, Octrees, and Kd-Trees". In: *Proceedings of High-Performance Graphics*. 2012, pp. 33–37.

[16] M. Vinkler, J. Bittner, V. Havran. "Extended Morton Codes for High Performance Bounding Volume Hierarchy Construction". In: *Proceedings of High-Performance Graphics*. 2017.

[17] J.H. Clark. "Hierarchical Geometric Models for Visible Surface Algorithms". In: *Communications of the ACM 19, 10*. 1976, pp. 547–554.

[18] M. Vinkler, V. Havran, J. Bittner. "Performance Comparison of Bounding Volume Hierarchies and Kd-Trees for GPU Ray Tracing". In: *Computer Graphics Forum*. 2016.

[19] J. Bikker. *How to build a BVH – Part 1: Basics*. Accessed in 2024. URL: `https://jacco.ompf2.com/2022/04/13/how-to-build-a-bvh-part-1-basics/`.

[20] D. Meister et al. "A Survey on Bounding Volume Hierarchies for Ray Tracing". In: *Computer Graphics Forum 40, 2*. 2021, pp. 683–712.

[21] T. Karras, T. Aila. "Fast Parallel Construction of High-Quality Bounding Volume Hierarchies". In: *Proceedings of HighPerformance Graphics*. 2013, pp. 89–100.

[22] J.D. MacDonald, K.S. Booth. “Heuristics for Ray Tracing Using Space Subdivision ”. In: *The Visual Computer 6, 3* (1989), pp. 153–165.

[23] T. Aila, T. Karras, S. Laine. “On Quality Metrics of Bounding Volume Hierarchies”. In: *Proceedings of High-Performance Graphics*. 2013, pp. 101–108.

[24] S. Popov et al. “Object Partitioning Considered Harmful: Space Subdivision for BVHs”. In: *Proceedings of High-Performance Graphics*. 2009, pp. 15–22.

[25] I. Wald. “On Fast Construction of SAH based Bounding Volume Hierarchies”. In: *Proceedings of Symposium on Interactive Ray Tracing*. 2007, pp. 33–40.

[26] C. Apertrei. “Fast and Simple Agglomerative LBVH Construction”. In: *Proceedings of Computer Graphics and Visual Computing*. 2014.

[27] Crytek. *Crytek Sponza*. Accessed in 2024. URL: `https://apartridge.github.io/OppositeRenderer/images/sponza.html`.

[28] Frank Genarri. *San Miguel*. Accessed in 2024. URL: `https://3dworldgen.blogspot.com/2017/01/san-miguel-scene.html`.

[29] L. Domigues, H. Pedrini. “Bounding Volume Hierarchy Optimization through Agglomerative Treelet Restructuring”. In: *Proceedings of High-Performance Graphics*. 2015, pp. 13–20.

[30] I. Wald, C. Benthin, S. Boulos. “Getting Rid of Packets Efficient SIMD Single-Ray Traversal using Multi-Branching BVHs”. In: *Symposium on Interactive Ray Tracing*. 2008, pp. 49–57.

[31] I. Wald, C. Benthin, P. Slusallek. “Distributed Interactive Ray Tracing of Dynamic Scenes”. In: *Proceedings of Symposium on Parallel and Large-Data Visualization and Graphics*. 2003, pp. 77–86.

[32] C. Benethin, S. Woop, I. Wald. “Improved Two-Level BVHs using Partial Re-Braiding”. In: *Proceedings of High Performance Graphics*. 2017.

[33] Microsoft. *DirectX Raytracing (DXR) Functional Spec*. Accessed in 2024. URL: `https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html`.

[34] Inc. Advanced Micro Devices. *Radeon Developer Panel*. Accessed in 2024. URL: `https://radeon-developer-panel.readthedocs.io/en/latest/`.

[35] Inc. Advanced Micro Devices. *Radeon GPU Profiler*. Accessed in 2024. URL: `https://gpuopen.com/manuals/rgp_manual/rgp_manual-index/`.

[36] Remedy games. *Control*. Accessed in 2024. URL: `https://www.remedygames.com/games/control`.

[37] Bethesda. *Deathloop*. Accessed in 2024. URL: `https://bethesda.net/en/game/deathloop`.

[38] Techland. *Dying Light*. Accessed in 2024. URL: `https://dyinglightgame.com/`.

[39] Focus Entertainment. *A Plague Tale Requiem*. Accessed in 2024. URL: `https://www.focus-entmt.com/en/games/a-plague-tale-requiem`.

[40] UL Benchmarks. *Port Royal Benchmark*. Accessed in 2024. URL: `https://support.benchmarks.ul.com/support/solutions/articles/44002135553-overview-of-3dmark-port-royal-benchmark`.

[41] Capcom. *Resident Evil*. Accessed in 2024. URL: `https://www.residentevil.com/re4/en-asia/`.

[42] Microsoft. Accessed in 2024. URL: `https://forza.net/horizon`.

[43] Forza Support Team. *NVIDIA DLSS, AMD FSR and DirectX Ray Tracing Improvements in Forza Horizon 5*. Accessed in 2024. URL: `https://support.forzamotorsport.net/hc/en-us/articles/10944080013843-NVIDIA-DLSS-AMD-FSR-and-DirectX-Ray-Tracing-Improvements-in-Forza-Horizon-5`.

8 Appendix

8.1 Pseudocode Mipping

8.1.1 Pseudocode Generating Occupation Grid

Creating the occupation grid starts by setting the base layer of the grid:

```
1: primitive = Primitive for this thread
2: b = #bits for the occupation grid
3: cellCode = EMC(primitive, b)
4: AtomicSetBit(cellCode)
```

Where **AtomicSetBit()** atomically sets a bit in the base layer. **EMC()** computes an EMC code for a primitive with a given length.

The base layer is extended by more layers:

```
1: threadIndex = Index of current thread
2: layers = b / 3
3: for layer = 1; layer $\leq$ layers; layer=layer + 1 do
4:     dwordIndex = threadIndex
5:     if dwordIndex < LayerDwordCount(layer) then
6:         prevDwords = FetchDwords(8 * dwordIndex, layer - 1)
7:         newDword = MergeDwords(prevDwords)
8:         WriteDword(newDword, layer)
9:     end if
10: end for
```

Where **LayerDwordCount()** holds the number of dwords in the current layer. **FetchDwords()** fetches eight dwords from a given layer. **MergeDwords()** merges eight bits of a single dword into one bit; therefore, merging eight dwords creates a new dword. **WriteDword()** Writes a dword in a given layer.

8.1.2 Pseudocode Creating Concise Cell Index

```
1: primitive = Primitive for this thread
2: b = #bits for the occupation grid
3: cellCode = EMC(primitive, b)
4: conciseCellCode = 0
5: layers = b / 3
6: for layer = 1; layer $\leq$ layers; layer=layer + 1 do
7:     currentByte = FetchByteInLayer(cellCode, layer)
8:     AppendNecessarySplits(conciseCellCode, cellCode, currentByte, layer)
9: end for
```

```
10: return $conciseCellCode$
```

Where **EMC()** computes an EMC code for a primitive with a given length. **FetchByteInLayer()** Fetches the byte in a layer the cellCode is in. **AppendNecessarySplits()** appends the necessary splits to a code.

8.2 Pseudocode Binning

8.2.1 Pseudocode Queue

```
 1: $splitDepth$ = The depth of the current splits
 2: $queue$ = The queue containing tasks, starts with a single task
 3: $scores$ = Array of scores for all current tasks
 4: for $depth = 0$; $depth < splitDepth$; $depth = depth + 1$ do
 5:     ClearScores()
 6:     $localQueue$ = SubdivideTasks($queue$)
 7:     for $subTask$ in $localQueue$ do
 8:         ComputeScore($subTask$)
 9:     end for
10:     for $subTask$ in $localQueue$ do
11:         AtomicMin($scores[subTask.index]$, $subTask$.score)
12:     end for
13:     for $subTask$ in $localQueue$ do
14:         if $scores[subTask.index] == subTask.score$ then
15:             WriteNewTasks($subTask$)
16:         end if
17:     end for
18: end for
```

Where **ClearScores()** removes the values from the old scores. **SubdivideTasks()** evenly subdivides the tasks into subtasks for all threads. **ComputeScore()** is described in 8.2.2 and computes the score of the current subtask. **AtomicMin()** applies an atomic min over a memory location. **WriteNewTasks()** Writes out the left and right split of the current task.

8.2.2 Pseudocode for Computing Scores

```
1: $scene$ = Binary occupation grid of the scene
2: $leftBounds$, $rightBounds$ = Left and right bounds of the current split, respectively
3: $m$ = #Bits used for bitmaps
4: $w$ = weight of Stretch Factor
5: $bitmaps$ = bitmaps containing the occupation
6: $leftMask$ = CreateMask($leftBounds$)
7: $rightMask$ = CreateMask($rightBounds$)
```

```
$leftScene = leftMask$ & $scene$
$rightScene = rightMask$ & $scene$
$leftMinMax$ = FindMinMax($leftScene$)
$rightMinMax$ = FindMinMax($rightScene$)
for $layer = 0$; $layer < m$; $layer = layer + 1$ do
    $leftBitmap = bitmaps[layer]$ & $leftMask$
    $rightBitmap = bitmaps[layer]$ & $rightMask$
    $leftCount = leftCount$ + countbits($leftBitmap$) * ($2^{layer}$)
    $rightCount = rightCount$ + countbits($rightBitmap$) * ($2^{layer}$)
end for
$leftSA$ = SurfaceArea($leftMinMax$)
$rightSA$ = SurfaceArea($rightMinMax$)
$leftSAH$ = Normalize($leftCount$ * $leftSA$)
$rightSAH$ = Normalize($rightCount$ * $rightSA$)
$leftSF$ = StretchFactor($leftMinMax$)
$rightSF$ = StretchFactor($rightMinMax$)
$leftScore = (w - 1)$ * $leftSAH + w$ * $leftSF$
$rigthScore = (w - 1)$ * $rightSAH + w$ * $rightSF$
return $leftScore + rightScore$, $leftMinMax$, $rightMinMax$
```

Where **CreateMask()** and **FindMinMax()** Create the mask and find the Min Max of the current split, respectively. **countbits()** counts the number of non-zero bits in a bitmap, or in this case, the number of occupied cells in the current layer. **SurfaceArea()** Computes the bounds between a min and max value. **Normalize()** normalizes the SAH score. **StretchFactor()** Computes the stretchfactor between a min and max value.

8.3 Additional Graphs over a Single Axis

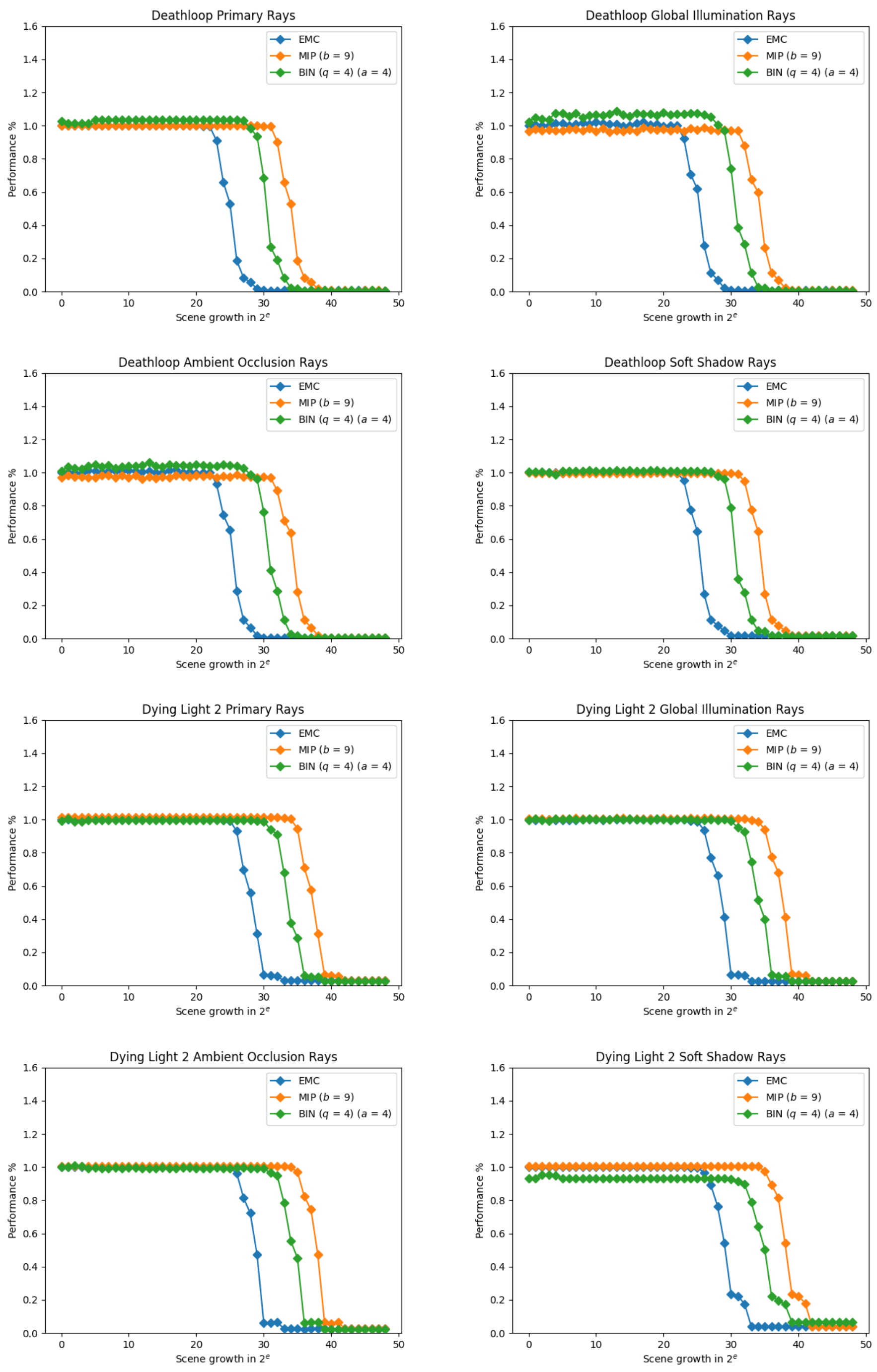

Figure 16: Graphs showing the relative performance in Deathloop [37] and Dying Light [38] of our strategies compared to LBVH with EMC without any scene bound growth of the different methods when the scene bounds grow by 2^e in a single axis, where e is the number on the x-axis. See Section See Section 4.3.

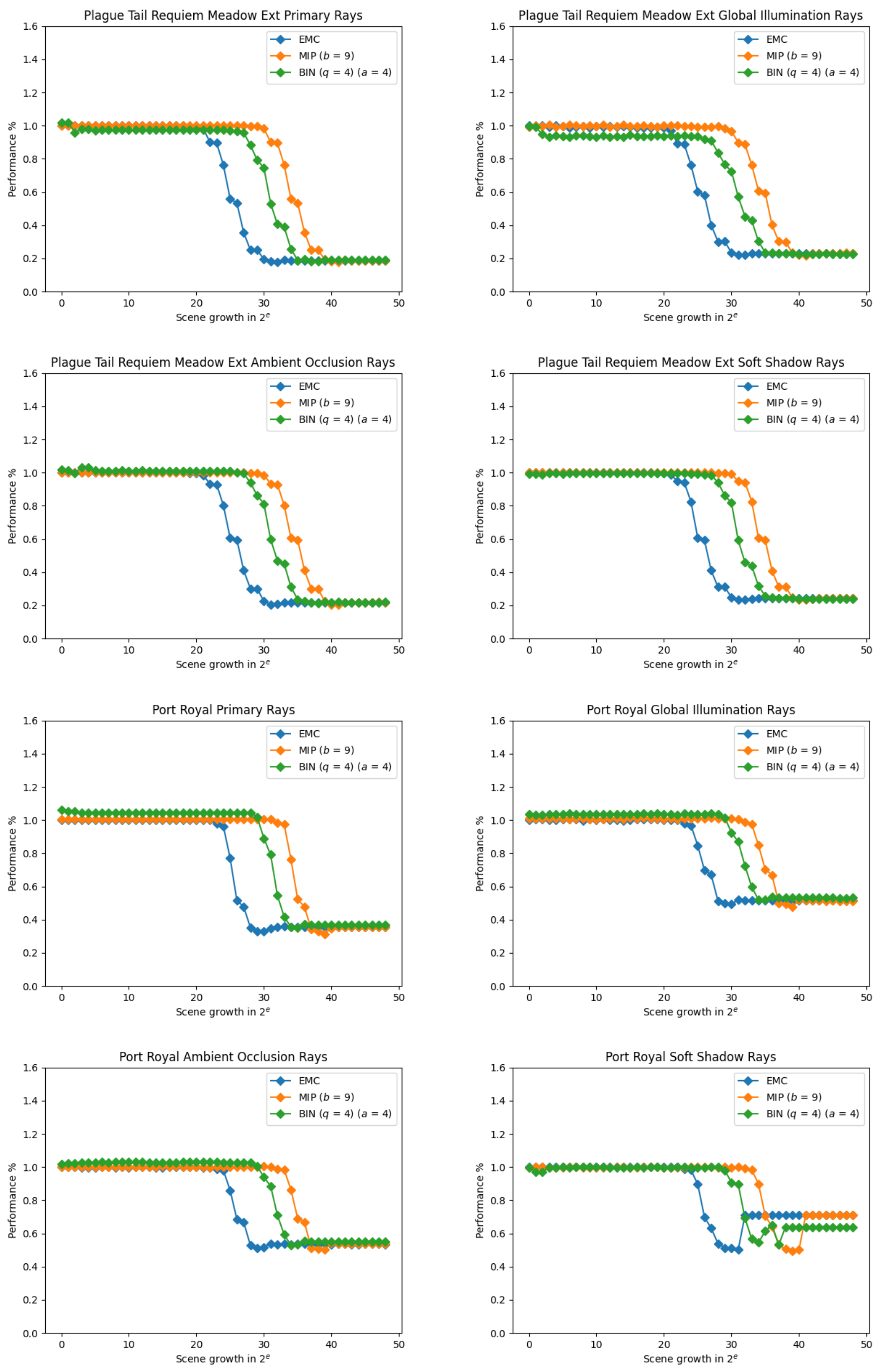

Figure 17: Graphs showing the relative performance in Plague Tail Requiem [39] and Port Royal [40] of our strategies compared to LBVH with EMC without any scene bound growth of the different methods when the scene bounds grow by 2^e in a single axis, where e is the number on the x-axis. See Section See Section 4.3.

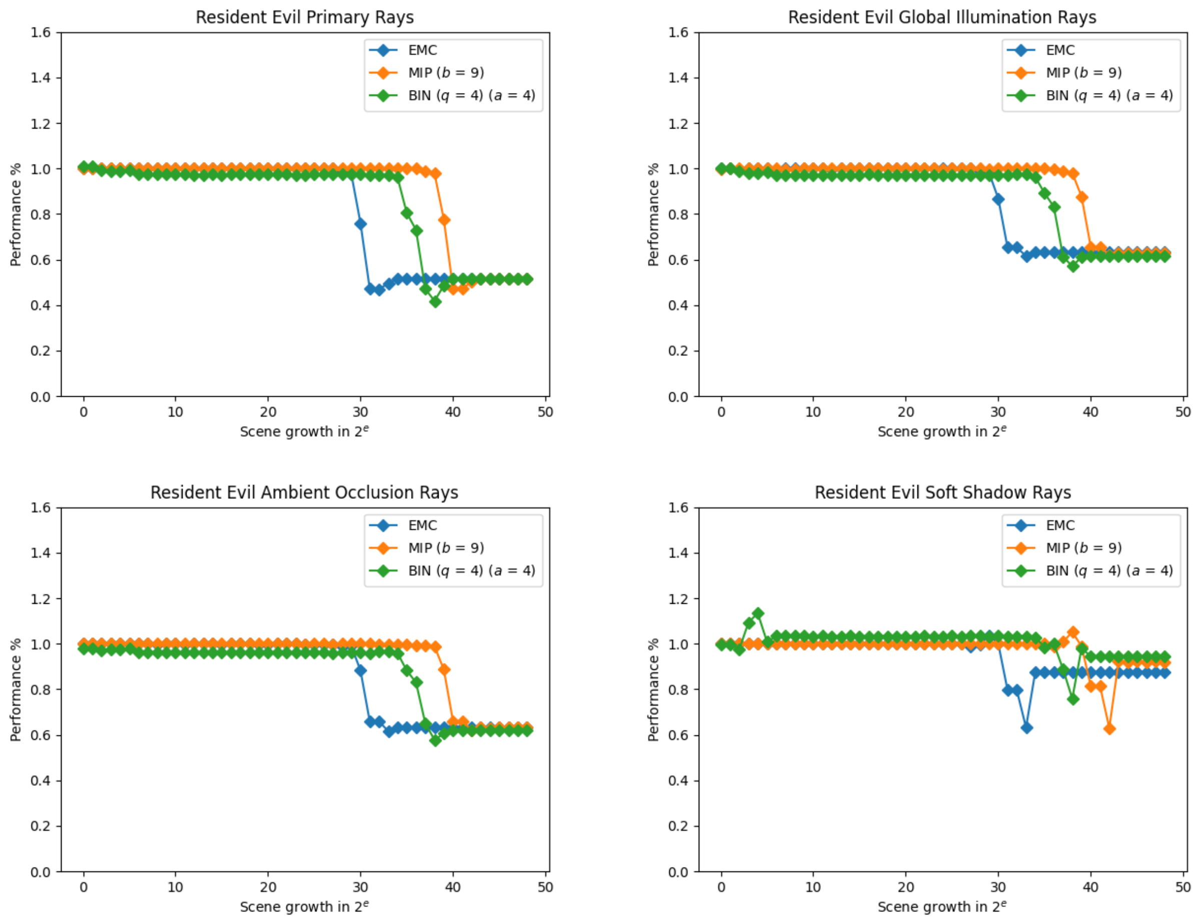

Figure 18: Graphs showing the relative performance in Resident Evil [41] of our strategies compared to LBVH with EMC without any scene bound growth of the different methods when the scene bounds grow by 2^e in a single axis, where e is the number on the x-axis. See Section See Section 4.3.

8.4 Additional Graphs over Three Axes

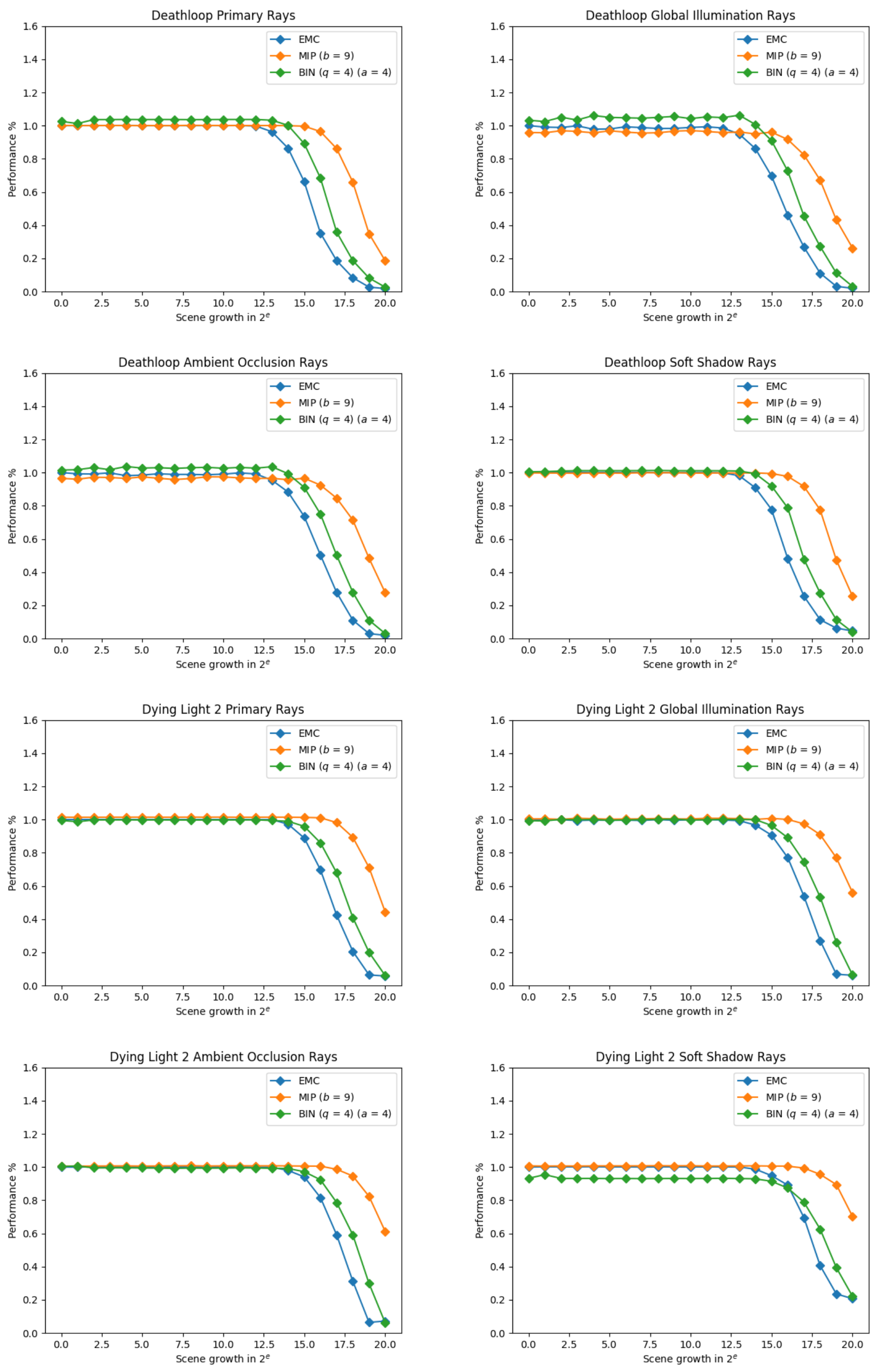

Figure 19: Graphs showing the relative performance in Deathloop [37] and Dying Light [38] of our strategies compared to LBVH with EMC without any scene bound growth of the different methods when the scene bounds grow by 2^e in all three axes, where e is the number on the x-axis.

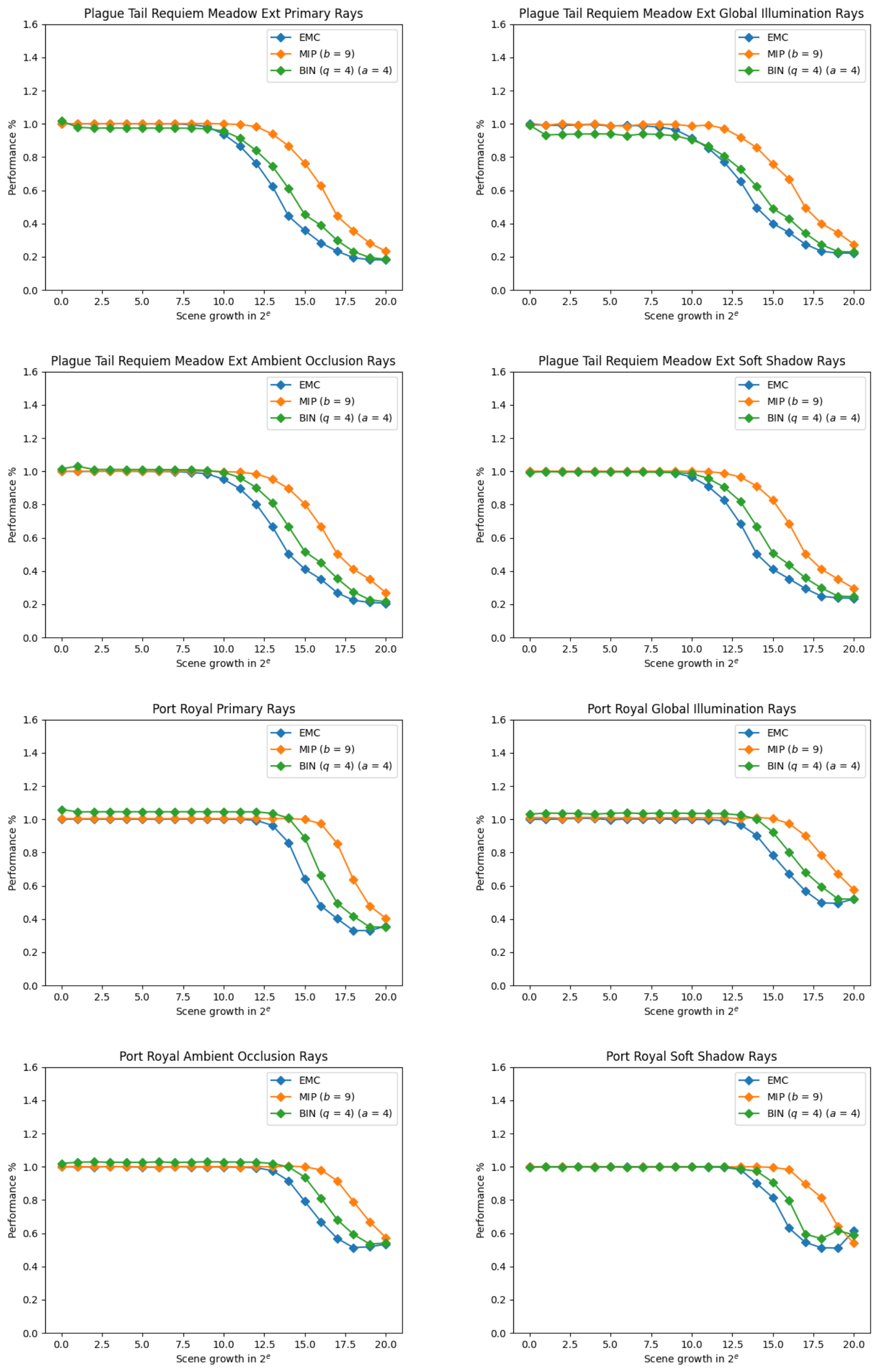

Figure 20: Graphs showing the relative performance in Plague Tail Requiem [39] and Port Royal [40] of our strategies compared to LBVH with EMC without any scene bound growth of the different methods when the scene bounds grow by 2^e in all three axes, where e is the number on the x-axis. See Section See Section 4.3.

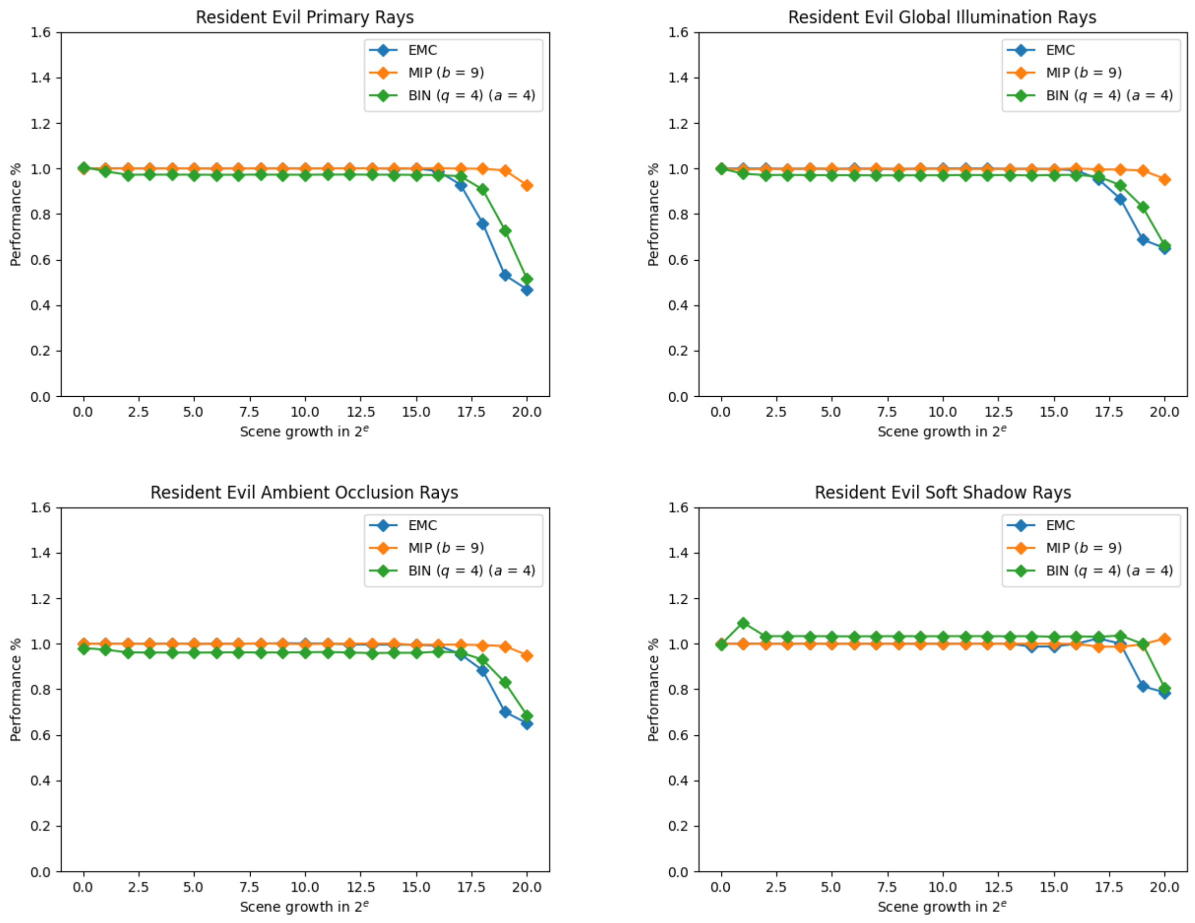

Figure 21: Graphs showing the relative performance in Resident Evil [41] of our strategies compared to LBVH with EMC without any scene bound growth of the different methods when the scene bounds grow by 2^e in all three axes, where e is the number on the x-axis. See Section See Section 4.3.