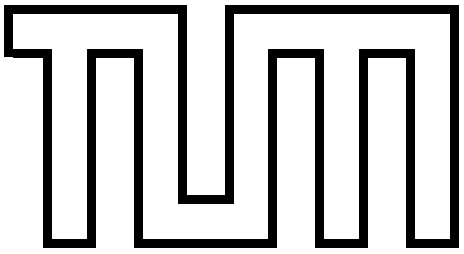


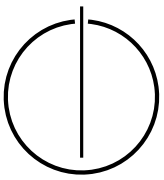
Department of Informatics

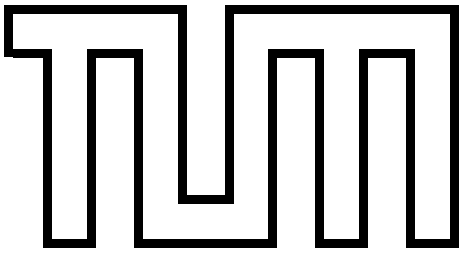
Technical University of Munich

Master's Thesis in Informatics: Games Engineering

Analysis of Different Bounding Volume Hierarchies and Ray Tracing Algorithms

Andreas Leitner





Department of Informatics

Technical University of Munich

Master's Thesis in Informatics: Games Engineering

Analyse Verschiedener Bounding Volume Hierarchies und Ray Tracing Algorithmen

Analysis of Different Bounding Volume Hierarchies and Ray Tracing Algorithms

Author: Andreas Leitner

Supervisor: Prof. Dr. Rüdiger Westermann

Advisor: Dr. Matthäus Chajdas

Submission: 15. March 2020

Ich versichere, dass ich diese Masterarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15. March 2020

(Andreas Leitner)

Acknowledgments

I would like to thank Matthäus Chajdas and Marcus Rogowsky for their support and guidance through this thesis. Thanks to Professor Westermann for supervising this topic.

In addition I would like to add a special thank you to Sebastian Neubauer and Lukas Jagemann for always having an open ear, and for their occasional help with the struggles one can have when programming with C++.

Abstract

Over the past decade computer games have continuously improved their visual quality and complexity. Due to the rapid increase in the processing power of GPUs, the new visual effects could also become more expensive without affecting the frame rate. However, it was still not possible to correctly implement some realistic effects like reflections, refractions and global illumination. These effects are fundamentally not possible with the rasterization technique, which is used to efficiently render triangular scenes in real time applications. In order to achieve an imitation of these effects, developers have to deal with expensive workarounds, sometimes implemented as a post processing effect with screen space information [5].

Ray tracing is a technique that allows to simulate the behavior of light rays in scenes, and can be used to correctly implement visual effects such as reflections. This technique is mainly used for high quality offline renderers and is often implemented using the general purpose computing power of the GPU. This is usually too expensive for a purely visual effect in computer games.

The newest evolution of GPUs now includes fixed function pipelines for real time ray tracing [14]. With hardware acceleration it now begins to be feasible to trace a few rays per pixel per frame.

Despite intensive research in the field of ray tracing and bounding volume hierarchies (BVH) over the last years, there are no comprehensive analyses over all different BVH configurations. The few papers that try out new BVH configurations usually choose them since they fit the single instruction, multiple hardware (SIMD) capabilities of their hardware. While they usually compare their results with existing configurations such as binary BVH and QBVH [8] a comparison with all possible BVH configurations would be interesting.

The motivation behind this thesis is to explore all reasonable BVH configurations and to extensively test their effects on ray tracing, with the goal of finding out what kind of configurations are suitable for use in ray tracing on fixed function hardware on the GPU. For the ray tracing algorithm, this means that it can be designed for arbitrary SIMD widths. To evaluate a BVH configuration we implement an instrumented renderer, a cache simulator, and a normal, for performance optimized, renderer. While the benchmarks are computed on ordinary hardware they can still be used to validate our assumptions.

The main contributions of this thesis are:

- Evaluation of different bounding volume hierarchies.
- A detailed comparison between two ray tracing algorithms with special focus on cache efficiency.

Analysis of Different Bounding Volume Hierarchies and Ray Tracing Algorithms

Chapter 1

Introduction

This section explains what ray tracing is and introduces the different aspects of ray tracing we investigate in this thesis.

1.1 Ray Tracing

Ray tracing is a technique to simulate how a light ray would interact with the surrounding geometry. It can be used to accurately compute realistic visual effects such as reflections and global illumination.

The fundamental principle is to shoot a ray into the scene for each pixel that has to be rendered and calculate the closest intersection of the ray with the geometry. At each collision point, the properties of the surfaces are sampled for shading. Depending on the desired effect, new ray can also be created form the collision point.

The biggest limitation of the traditional rendering technique used in real time games, rasterization, is that for each camera perspective the whole geometry of the scene has to be put through a series of transformations. This means that additional cameras, like those used for shadow maps, are quite expensive, and it is not possible to use it for general reflections. This means that developers who want effects such as reflections and refractions have to implement complex workarounds to reach the desired visual effects [5].

For ray tracing the underlying acceleration structure changes only when the geometry move, therefore it is possible to cast rays from arbitrary positions and directions. This enables more accurate computations of shadow effect, and also avoids the problem of visual artifacts due to shadow cascades. Ray tracing can also be used to calculate effects that were previously not possible without workarounds such as reflections and global illuminations.

1.2 Bounding Volume Hierarchies

In order to use ray tracing in real time applications, it must to be fast enough to compute approximately one ray for every pixel. With a naive approach this would result in testing every ray against every triangle. For the later benchmark we calculate $3840 \times 2176 = 8355840$ pixels for scenes ranging from 260 thousand triangles to 5.6 million triangles.

To compute these ray / triangle intersections efficiently a spacial acceleration structure is required to greatly reduce the number of triangle intersections. The most commonly used structure is the bounding volume hierarchy (BVH). A BVH is a tree structure in which each node stores the bounding volumes of the child nodes. This means that if a ray does not collide with the bounding volume of a node it can also not collide with the children of that node. The geometric primitives are stored in the leaves of this tree.

The following bounding volumes are commonly used together with BVHs:

- Bounding Spheres. They have multiple properties that make them excellent for BVHs. They are often used in acceleration structures for collisions since sphere / sphere intersections are very easy to calculate and they are rotational invariant. The main issue is that bounding spheres are mostly empty when representing flat objects, what can lead to many false positives.
- An Axis aligned bounding box (AABB) is a cuboid that is aligned with the axis of the coordinate system. One benefit compared to spheres is that they can represent flat or pointy geometry more efficiently.
- Oriented bounding boxes (OBB) are rotated rectangular cuboids. They can efficiently represent many shapes while still having most of the properties of AABB. For organic structures like plants and hair, OBB can greatly improve the performance. It can be quite complex to compute the best OBB for an given set of primitives.

AABBs are used in this paper since they are very easy and fast to create and intersect.

The BVH can be created with different branching factors, meaning the amount of children each node can have. This is called node size. The leaf size is the number of primitives each leaf can store. The process of testing a ray against a node or leaf then contains multiple AABB or triangle intersections. SIMD can then be used to efficiently calculate the intersection with several AABB or triangles at once.

1.3 Motivation

The main problem to be investigated in this thesis is the behavior of different BVH configurations and also their traversal behavior with SIMD. This requires a BVH builder that can create BVHs with any combinations of node and leaf sizes as well as ray tracing algorithms that utilizes SIMD to efficiently traverse those BVH. To correctly evaluate a BVH an image is rendered with an instrumented renderer and two different performance optimized renderers. In addition to the normal performance benchmarks the cache behavior of the optimized renderers is also measured with a cache simulator.

The general goal is to find BVH configurations and a traversal algorithm that are suitable to be implemented in fixed function hardware with a SIMD. The SIMD width would depend on the configuration, meaning the node and leaf size of the BVH. While we cannot compute with any SIMD width on our CPU the performance benchmarks can still be used to validate most of the intersection results.

The overall focuses lies on the traversal aspect of ray tracing, and therefore anything related to shading or textures is ignored. The reason for this is that realistically only the BVH traversal is implemented as a fixed function pipeline in hardware. The process of shading for ray tracing is very similar as to what the pixel shader is designed for, so there is no reason to build something additional for ray tracing.

In summary, the key aspect of ray tracing to be investigated in this paper is the evaluation of different BVHs and traversal algorithms with respect to their general usefulness for an implementation as fixed function hardware.

Chapter 2

Design

This section introduces the BVH builder and the different traversal algorithms.

2.1 Benchmark Scenes

The scenes for our benchmarks should fulfill the following properties:

- "Game ready" meshes. This means mostly evenly distributed triangles without too many triangles that do not contribute the visible shape of the object.
- The scene should contain enough triangles. The general target is more than one million triangles.
- The scene should allow camera angels that do not show too much background.
- Textures and normal maps do not matter, since they are not used for shading.

This thesis covers only a ray tracer for triangles, since the general idea is to find the best BVH configuration together with a suitable algorithms for potential fixed function hardware on the GPU. The rasterizer on the GPU already uses triangle meshes and it would be unreasonable not to use models with the same type of primitives. Since SIMD is used to compute multiple ray - primitive intersections at once all primitives have to be the same type.

We also do not utilize any additional information about the scene, like a possible object hierarchy or any duplication of objects.

2.2 BVH Builder

The input to the BVH builder is a simple triangular mesh without any additional information about the objects or scenes.

For the structure of the BVH builder we roughly follow the implementation of the PBRT renderer [12]. They use a top-down, greedy surface area heuristic (SAH) based BVH builder. We modify the algorithm to allow arbitrary node and leaf sizes.

The node size is the target number of children each node in the tree should have. The leaf size is target number of leaves primitives.

The SAH is the cost function that is minimized during the creation of the BVH. It will be explained further after the introduction of the BVH builder.

Most state of the art algorithms that build wider BVHs first create a binary tree BVH and than collapse this tree in an SAH-optimal fashion to the desired wide tree [8, 16]. For our problem case, a collapsing algorithm is not feasible because we are interested in all possible node sizes. This includes odd numbers, which would be particularly difficult to collapse from a binary tree.

It also helps to provide a fair foundation for later analysis and comparison of different BVH configurations, if all BVH configurations are created with the same algorithm. In this paper we use a custom algorithm that directly creates nodes with the desired number of children by splitting multiple times.

The BVH builder works on every node individually. To simplify the following explanation the node the BVH builder is currently working on will be called `nodeC`.

The BVH builder starts with the root node which contains all primitives of the scene. The splitting algorithm is then applied to the root node and all newly created nodes.

Splitting Algorithm

The algorithm works by splitting the child nodes of `nodeC`, so the first step is to create a child that contains all the primitives of `nodeC`.

If `nodeC` has multiple children, the BVH builder chooses to split the child with the most primitives, in order to create a reasonably balanced tree.

Before any splits can be computed, the primitive of the child have to be sorted by an axis. The axis to sort is the one where an AABB containing the centers of the

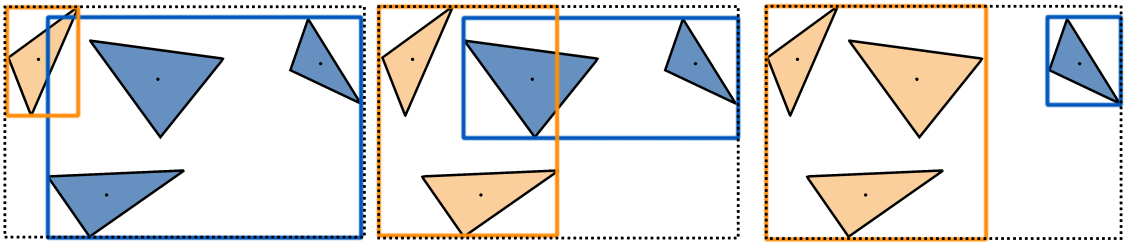


Figure 2.1: Example of the different splits and their respective AABB possible for this configuration of four triangles. The black dotted line shows the AABB of `nodeC`. In this case the right most split has the lowest SAH because both AABBs that the new nodes are relatively small. This split would also be preferred for ray tracing since it has no overlapping area.

triangles is the largest. This axis has a high chance for a gap between the triangles, so that later a split can be found that results in two nodes that do not overlap much.

The splitting algorithm itself tries every possible split, as shown in Figure 2.1, and computes the SAH of both resulting nodes of every split. The split with the minimal SAH cost is then applied, and the new child is added to `nodeC`.

If `nodeC` has not reached the target node size, the next largest child is split. This is repeated until either the target node size is reached or all children have less or equal the target amount of primitives.

If `nodeC` is not filled because the children already have the target leaf size, a second iteration is started. During this step the possible SAH improvement for each child is computed. If if it was possible to improve the SAH the split with the greatest SAH improvement is applied.

If there are no more split possible that improve the SAH, or if `nodeC` has reached the target node size, the algorithm is finished.

When the algorithm is finished it is called recursively on all children of `nodeC`.

Surface Area Heuristic

The surface area heuristic (SAH) is a function that estimates the cost of tracing a ray against a node of the BVH.

The SAH is minimized during the BVH creation in order to build an acceleration structure that is well suited for Ray Tracing. The general approach to optimize the SAH is a greedy top down algorithm that treats each node individually.

This means that the builder starts with the root node containing all primitives. All

primitives are then sorted along an axis and the SAH of each split is calculated. A split means splitting the primitives of a node into two nodes. When splitting at one index, all triangles with a smaller or equal index are placed in the left node and all with a larger index are placed in the right node.

The core concept of the SAH is that the surface area of a node divided by the surface area of the parent node is the chance to hit the node if the parent node has already been hit. This cost is calculated for both nodes of every possible split, and the BVH builder chooses the split with the lowest cost.

The SAH is calculated for each possible split to determine the one with lowest cost.

$$\texttt{SAH} = \texttt{nodeCostFactor} * (\texttt{XL} * \texttt{SAL} + \texttt{XR} * \texttt{SAR})/\texttt{SAP} \tag{2.1}$$

`XL` and `XR` are the number of primitives in the left and right child after a split and represent the cost of further traversing this child. `SAL` and `SAR` are the surface areas of the left and right child. `SAP` is the surface area of the parent node.

This can be summarized to the following: The cost to traverse the children is weighted by the approximated chance of hitting the AABB if the parent node has already been hit. Minimizing this value results in two small, often similarly sized child nodes.

The `nodeCostFactor` is the relative node cost compared to the leaf intersection cost. As an example, a `nodeCostFactor` of 0.5 means that a leaf intersection takes twice as long to compute than a node intersection. This number is relative, so that the cost of a leaf intersection is equal to the number of primitives it contains. The reason for this factor is to compare how expensive a node or leaf is. This is used to decide whether it is better to further split possible leaves. The `nodeCostFactor` is computed by measuring the performance of nodes and leaves of the ray tracer implementation, and might change depending on the hardware. Since it also depends heavily on the BVH configuration there is one `nodeCostFactor` for every combination of node and leaf size used for performance benchmarks.

The SAH does not utilize the target leaf size, so it simply tries to build a good BVH. It turns out that this results in many leaves with only a few primitives. In order to improve the fullness of leaves we use a modified SAH that scales the SAH cost with a step function depending on the number of children. `XL` and `XR` are modified to the following step function: $\texttt{XNew} = Ceil(\texttt{X}/\texttt{LeafSize}) * \texttt{LeafSize}$. This also corresponds to the general concept of computing leaves with SIMD, which means that there is no cost difference for differently filled leaves.

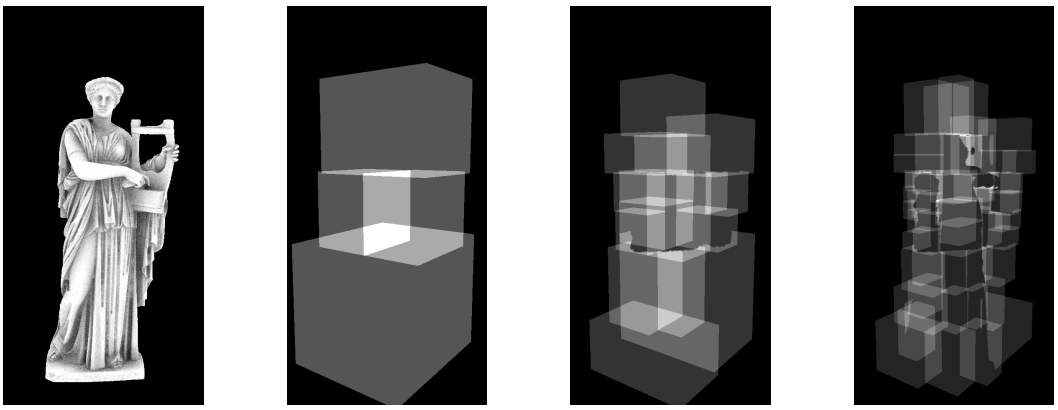


Figure 2.2: Visualization for BVH with node size 4 of the model Erato. The leftmost image shows the render result. Each image after that shows number of AABBs that are hit during the ray tracing for the specific depth of the BVH. From left to right the depth increases by one. This means the first image shows the AABB of the root node. The next image shows the AABB of all children of the root node, and so in. Black represents no collision. The grey scales are normalized to the maximum intersection count for each image.

Using N4L4 as an example, the modified SAH reduced the number of leaves from 655k to 370k and number of nodes from 265k to 140k. So this is an important improvement to the overall memory usage of the BVH, while also improving performance.

Several hybrids that interpolate between the two mentioned SAH functions were tested, but the pure step function had the best results.

BVH example

Figure 2.2 visualizes the nodes and its AABB in the first few levels. The model used for this examples is Erato, downloaded from Morgan McGuire's Computer Graphics Archive [10]. The Appendix also contains BVH visualizations of the same scene for N2, N8, N12, and N16.

Node Traversal Order

In order to improve the BVH traversal time of the ray tracer a heuristic is introduced that guides the traversal towards nodes that are close to the camera. Each node then stores some additional information, so that the ray traversal algorithm knows the order to traverse the child nodes. The normal approach for a binary tree BVH would be to save the axis the nodes were sorted along during the BVH creation.

The traversal order is then determined by the sign of the ray direction for the stored sorting axis.

The paper introducing the QBVH extends this algorithm to nodes with 4 children by saving the 3 split axis [8]. For our case this approach is not applicable, because our nodes can be split unevenly and can also be arbitrarily large. This can be nicely seen in Figure 6.6 in the Appendix.

It would be possible to save the different split axes in some tree structure and then expensively reconstructing everything during runtime to determine the child traversal order. Instead, it is faster to store different traversal orders and determine the child order during traversal depending on the direction of the ray.

The order of the AABB centers of the children are precomputed for all 3 axes(x,y,z). During ray tracing the largest axis of the ray direction is then the axis of which the stored traversal order is used. If the largest axis is negative, the stored order is applied in reverse order.

BVH Memory Layout

After the BVH builder is finished, the BVH is converted into the following cache efficient and compact format that we use for rendering.

A normal BVH implementation that is not specifically designed for SIMD, like the one described in PBRT [12], would store the bounds of a node with the node itself. For a SIMD approach this would result in scattered memory access for every node intersection. Instead, it is better to store all the bounds of the children tightly bundled together in the parent node. The data is stored in structure of arrays (SoA) order for efficient SIMD access. The SoA order is further explained in Chapter 3.2.2 about storage structure.;

One disadvantage from of approach is that leaves have unused memory, since they have no children and therefore no AABB to store.

In summary, each node stores the following information.

- The bounds of all children.
- The index to either the first child or the first primitive.
- The Child traversal order for the 3 axes.
- Which child is a node or leaf.

The nodes are stored in level order. This has the effect that the indices of the siblings

are always incremental, therefore only the first index has to be stored. The primitives are stored in a separate array, in the order their in which respective leaves are stored in the node array.

2.3 Camera

For this thesis we want to test the properties of different ray traversal algorithms.

A modern game would not replace the rasterizer with ray tracing, but would rather use ray tracing for secondary rays, since the rasterizer can not be compute those effects without tricks. Some possible effects would be global illumination, reflections, and shadows. For the purpose of evaluating render results and generalizing them do different scenes, we decided against rendering shadows and reflections, because their computational complexity is very scene dependent and the results would be very hard to generalize.

There are two different types of workloads for ray tracing. Coherent rays all have a similar start position and direction and therefore traverse the BVH with very similar paths. Depending on the size of the triangle and the distance to the start position, coherent rays that spawned together often hit the same triangle. Incoherent rays on the other side have highly varying directions and therefore go different paths through the BVH and most likely will not hit the same primitives.

In order to enable a fair evaluation that is not scene dependent this thesis focuses on the following two types of rays:

- Primary rays are very coherent and traverse the bvh until they find the nearest triangle that hits the camera. This is basically the equivalent to the rasterizer in traditional rendering. Primary rays have similar properties to shadow rays and reflections on glossy surfaces, but their workload is less dependent on the scene and camera angle.
- The incoherent rays we traverse are secondary rays to compute global illumination. These rays are spawned from the surface the primary ray hit and go in a random direction in the hemisphere. They only travel a limited distance and terminate and terminate on the first surface they hit. Depending on how many of the secondary rays hit something the surface is shaded brighter or darker.

The camera creates and manages all the rays that are traced. To do this efficiently, multiple rays are bundled together into work groups and multi threading is used to compute multiple work groups in parallel. This approach is preferred to starting a

thread for each ray due to the overhead of creating and switching between threads.

2.4 BVH Traversal

The general approach of BVH traversal is to do a depth first search through the BVH tree by exploring those child nodes where the ray has a collision with the respective AABB. Each time a ray successfully hits a triangle, the maximum distance of the ray is set to the surface distance. This makes all future node intersections behind this triangle a miss. This means that every triangle hit can heavily reduce the amount of nodes the ray can possibly hit, therefore it is important to use a heuristic to early hit any close surfaces. This allows us to find the correct triangle with about 60 node and leaf intersections for a complex model with 1 million triangles.

The indices of all nodes and leaves that still have to be traversed are stored in the `nodeStack`. Since the BVH is traversed in a depth first manner its a last in, first out data structure. The `nodeStack` is initialized with zero, the index to the root node.

There are two types of intersections that can happen during BVH traversal. Node intersections, that decide on the subtree to further traverse and leaf intersections, that compute the actual ray - primitive intersection.

Node Intersection

A node intersection is the process of testing a ray against all child AABBs of a node in the BVH. It is the most common operation of the ray tracer and SIMD is used to compute the intersection against multiple AABBs at once. For each AABB that is hit the index of this node is added to the nodeStack.

When a ray successfully hits multiple AABBs in one node, the order the child nodes are stored on the `nodeStack` has to be determined. The theoretically best heuristic would be to sort the AABBs by the intersection distance, but this would be to expensive to do in real time. Instead it is faster to use the precomputed traversal orders that is stored in the nodes to determine the traversal order. This means the depth first search through the tree has a heuristic that guides it towards nodes that are closer to the camera.

Normally when doing a node intersection without SIMD one would only compute the intersection with the closest child and store the untested children on the `nodeStack`. The nodes stored in the `nodeStack` would then be untested nodes. Our SIMD approach tests all AABB of a node at once, so it stores the successfully tested nodes. This reduces the amount of nodes that have to be stored. It also means that if the

maximum distance of the ray is updated during the traversal it could lead to nodes in the `nodeStack` that would not be hit anymore if the intersection with their parent were recomputed. When the intersection with those nodes is later performed no child can be hit since the node itself cannot be hit anymore. This means the that the SIMD approach to compute node intersections has slightly more node intersections than necessary.

A possible solution to this problem would be to store the AABB intersection distance together with the successfully hit node. Before doing an intersection the stored distance can be compared against the ray distance, and when the stored distance is larger the node can be discarded. This approach was implemented but it did not improve the performance for most BVH configurations. Only for Node Sizes of 16 and larger the overall render time improved slightly. We decided to not use it, also because the implementation was not feasible for the wide traversal algorithm.

Leaf Intersection

While the nodes contain AABB together with the index to the first child, leaves only store the index to the first primitive it contains. For a leaf intersection the ray is tested for a collision against all primitives of the leaf. Similar to nodes this is done with SIMD. For the case when a ray hits multiple triangles during a leaf intersection only the index of the closest triangle should be returned.

For every successful leaf intersection the maximum distance of the ray also has to be updated to distance of the successful hit triangle.

Primary Rays

Since primary and secondary rays have fundamentally different behaviors, their traversal is implemented separately to improve performance.

For each pixel one primary ray is cast to find the closest surface. They basically do the task the rasterizer performs for the conventional rendering approach. The traversal can be described as search for the closest triangle hit in the BVH.

Secondary Rays

For every primary ray that hits a surface a secondary ray is cast in a random direction in the hemisphere of this surface. The normal for the hemisphere is determined with flat shading, so it only requires position the of the triangle edges to be computed.

These secondary rays simulate a very simple version of global illumination by shading

the surface depending on how many rays hit a surface. The maximum distance of the ray is limited. The more secondary rays hit something the darker the original pixel is shaded, since it is occluded in close proximity and therefore should receive less light from its surroundings.

The special property of those secondary rays is that the BVH traversal can stop once it successfully hits the first triangle, so it can be described as an any hit search through the BVH. This allows a few optimizations of the node and leaf intersection code that can speed up the BVH traversal.

The main reason to use the heuristic for the child traversal order for primary rays is to find a triangle close to the camera as early as possible. For secondary rays this does not apply, since it not continues the traversal after the first hit. Therefore it is faster to do the children in an arbitrary order.

The leaf intersection problem changes from closest hit to any hit problem. Therefore the SIMD algorithm no longer needs to return the closest hit, but only if of the triangles are hit. This simplifies the SIMD implementation and improves the intersection performance.

2.4.1 Traversal Algorithms

The following paragraphs explain how the general concept of BVH traversal is applied for the two different traversal algorithms discussed in this paper.

Single Ray Traversal

This is the straight forward approach of calculating the ray results. Each ray is computed separately, from the ray creation to the final intersection that finishes the BVH traversal. Only when the previous ray is finished will the next ray be started.

First, all primary rays of the work group are computed. For a normal ray tracer the next step would depend on the surface material properties and textures. We only compute ambient occlusion for secondary rays, so only the surface position and surface normal are required to continue.

Primary and secondary rays are separated. This means computing the result for one work group consists of two parts. First calculating all primary rays, and then from those results computing all the secondary rays. This separation improves the cache behavior since the primary rays are highly coherent and traverse the tree in a similar manner. Therefore the next primary ray will most likely also traverse a similar tree and can reuse some of the cache lines of the previous ray.

The big problem with single ray traversal is that it requires a cache big enough to store all nodes and leaves used during the tree traversal. Otherwise it would start to evict the oldest nodes what would lead all of the cache lines being evicted before they can be reused.

Wide Traversal

Wide Traversal is an approach to reorder the node and leaf intersections done to compute all rays of a work group to be more cache efficient.

The general strategy is to compute one intersection for each ray in the work group. After all rays have computed one step the second step is started, and so on, until all rays are finished.

The benefit from this is that when all rays simultaneously traverse through the BVH, that all the rays should be in similar areas of the tree. While it might be better for coherent rays, for incoherent rays the first view steps of the the BVH traversal should still be able benefit from it. Now the cache only needs to be large enough to store the different nodes needed for one step of all rays.

The disadvantage is that there is a overhead to manage and store all the states of each ray.

Node and leaf intersections are separated. One step consists of first going through all the rays that calculate node intersections, and updating their nodeStack according to the intersection results. Then all rays that are doing leaf intersections are computed. When the first rays terminate the number of intersections that are calculated per step decreases.

The benefits of separating the node and leaf computations is that it is a bit more cache efficient. It would also be possible to delay work on the rays that do leaf intersections until there is a sufficient amount of work to be done in one step.

Figure 2.3: Bistro Interior scene with 50 ambient occlusion samples.

Rendering results

Figure 2.3 shows a rendered image of the Bistro Interior scene with 50 secondary ray samples. The model was downloaded from Morgan McGuire's Computer Graphics Archive [10].

Figure 2.4 visualizes the workload of the different areas of the image. Pixels that nearly miss geometry are expensive, in this image it can be seen in the bright halo around most objects. An interesting effect for the leaf intersections are the rather large AABB that can be seen above the bar and on the bar counter. Those are AABBs that encapsulate very long triangles that are not aligned with an axis. This means the AABB also cover much empty area.

Figure 2.4: Top image: Visualization of the number of node intersections. Bottom image: Visualization of the number of leaf intersections. Both images are rendered with N4L4. The gray values range from white, maximum number of intersections, to black, zero intersections. For the node intersections the maximum is 163, and for leaf intersection it is 53.

Chapter 3

Implementation

This section will go into detail on the implementation of the renderer that is described in the Design Chapter and is later used for the performance benchmarks. It will first go over the process of loading the models and creating the BVH for rendering. Then the implementation of the different ray traversal algorithms will be explained.

The implementation part of this thesis is written in C++ 17 and compiled with x64 msvc 19.24. In the following the used tools and libraries are listed.

- OpenGL Mathematics (GLM) is used as a mathematics library [6]. Overall it provides all mathematic constructs needed for graphics computations, like vectors, matrices, and quaternions.
- For model loading TinyGLTF is used [9].
- Intel SPMD Program Compiler (ISPC) provides a simple way to write SIMD code in a similar fashion to how shaders are written for the GPU [3].
- Since ISPC is missing templates or similar features we use Jinja2 as a templating engine to generate some ISPC code [13].
- OpenMP is used for easily configurable multi threaded loops [11].

3.1 Model Loading

All scenes are stored in the glTF format and loaded with TinyGLTF [9].

Any additional information about the model is not used, so the possibly complex game object structure is converted to a simple triangle list.

After loading the scene all triangles that have no surface area are removed, since these triangles cannot be hit by a ray and could produce bounding boxes without volume. It should be noted that those degenerate triangles are not necessarily a model imperfection, but rather a technique to represent hard edges in a smoothly shaded model. However, since we generally do flat shading, it is preferred to remove those triangles to avoid possible edge cases and reduce the amount of triangles that have to be stored in the BVH and later rendered.

3.2 Bounding Volume Hierarchy Builder

The BVH builder starts with the root containing all primitives. The following splitting algorithm is then recursively applied on the root node and all newly created nodes and leaves until there is no SAH improvement possible. The recursion can use a tasking system to be computed in parallel and when the recursion terminates the BVH structure is finished.

`nodeC` is the node that is currently beeing worked on. For the first split `nodeC` is the root node.

The algorithm can be divided into three parts. The first part consists of creating children from the primitives of `nodeC` and then splitting the largest child until either there are no children left that have more primitives than the target leaf size, or `nodeC` has reached the target node size.

```
if Primitve Count ≥ Leaf Size then
    Create a child that contains all primitives of nodeC;
    while Child Count ≤ Node Size and for at least one child: primitives ≥
     Leaf Size do
        Choose child with most primitives;
        //Do a SAH optimal split
        Choose new split axis;
        Sort primitive by their centers along the split axis;
        Compute SAH of every possible split;
        Perform the split at the position with the minimal SAH. Create a new
         child and give it the triangles removed from the existing child;
    end
end
```

Algorithm 1: Node splitting algorithm first iteration

When the `nodeC` has reached the target node size, it is finished and the algorithm is called for all children. For this case the other parts of the algorithm are not required.

Otherwise, the second and third part of the splitting algorithm come into play and further split nodes that could be leafs according their number of primitives. Only those splits are actually applied that have a lower SAH after the split than before.

There are two different types of splits. The first type is when `nodeC` already has children. Then all splits for all children are tested, regardless of the numbers of primitives they have and the split with the highest SAH improvement is applied.

```
//second iteration, further split children if it improves the SAH while Child
 count ≤ Node Size do
    For all children, try all possible splits out (see first iteration) and compute
     the possible best SAH improvement;
    if Best SAH improvement positive then
        Apply the best split to the leaf;
        //this increases the child count by one.
    else
        //There is no improvement in splitting any leaf, so nodeC is finished.
        break;
    end
end
```

Algorithm 2: Splitting child leaves

When this is finished the recursion continues and the splitting algorithm is called

for all children.

The second type of leaf splits is for `nodeC` that have no children. This means their number of primitives is less or equal the leaf size. Since it might be a SAH improvement to place those primitives into two or more children this has to be tested.

```
//second iteration to split nodeC if it can improves the SAH
Save current primitives of nodeC;
Create a child that contains all primitives of nodeC;
while Child count ≤ Node Size do
    For all children, try all possible splits out (see first iteration) and compute
     the possible SAH improvement of every child;
    if Best SAH improvement positive then
        Apply the split to the child;
    else
        //There is no improvement in splitting any child. break;
    end
end
if SAH did not improve over previous leaf or Child count = 1 then
    Remove the children and restore nodeC to the status before this algorithm;
else

end
```

Algorithm 3: Splitting leaves

If the algorithm performed any splits, the recursion continues with the children. Otherwise, the `nodeC` finished and is a leaf of the BVH. This is the terminating case of the recursion.

The calculation of the SAH improvement for the second type of splits requires the `nodeCostFactor` to compare the cost of multiple leaves against the cost of `nodeC`. The computation of the `nodeCostFactor` is described in Section 4.3 together with the final values.

3.2.1 Computing Best SAH Split

Before a node can by split its triangles must be sorted along an axis. This axis is also called split axis. The spit axis is along the dimension where the bounding box containing the centers of the primitives is the largest. This axis has the most distance between the triangles, so splitting along it should create nodes that do not overlap much.

The computationally expensive part of calculating the SAH for every split is to compute the AABBs of the primitives. A naive approach would have a quadratic runtime depending on the number of primitives.

To compute this efficiently, a sweeping algorithm is used, which divides the problem of computing the AABB of all splits into two steps. We start at the beginning of the primitive list with the first split containing only one triangle for the small side and all the other triangles for the other side. The AABB of the single triangle is then computed and used to calculate the "left" part of the SAH of this first split.

Then with each step one triangle is added. The AABB of the previous step can be reused and is updated to also include the newly added triangle. This is repeated until we have gone through all primitives. The SAH cost of the left side for each split is stored in a list.

To compute the right side, the same algorithm is repeated, starting from opposite side. This computes the right part of the SAH for every split. The SAH results of the reverse order are then added to the these of the first iteration.

The entry in the list of SAHs with the minimal cost is the best possible split.

The SAH function from the PBRT book [12] would be the following:

$$\texttt{SAH} = \texttt{PrimCount} * \texttt{SA} * (1/\texttt{SAP}) \tag{3.1}$$

`PrimCount` is the number of primitives in the new node. `SA` is the surface area of the AABB containing said primitives. `SAP` is the surface area of the parent node.

This function scales linearly with the number of primitives. Our modified SAH function increases the average amount of primitives in the leaves by introducing a step function. This means that the cost of a leaf intersection with one primitive is the same as the cost for a full leaf, what would also correspond to our general SIMD approach.

$$\texttt{SAH} = ceil(\texttt{PrimCount}/\texttt{LS}) * \texttt{LS} * \texttt{SA} * (1/\texttt{SAP}) \tag{3.2}$$

`LS` is the target leaf size. The step function roughly guides the number of primitives inside nodes during the BVH creation to be divisible by the leaf size. Over all scenes and BVH configurations this modified SAH function achieves an average leaf fullness of 98.5% before leaf splitting. The fullness describes the percentage of primitives in the leaf compared to the target leaf size.

3.2.2 Compact BVH Structure

The internal node and leaf structure during the BVH creation is not important since the BHV is copied into a compact form for the performance tests.

The compact representation for rendering uses two arrays, one for all nodes and one for triangles.

Node Data

The data layout for the compact nodes is as follows:

```
1 struct alignas(64) Node{
2   float bounds[2 * 3 * NodeSize];
3   uint32_t idBegin;
4   int8_t traverseOrderEachAxis[3 * NodeSize];
5   std::bitset<NodeSize> childType;
6 };
```

The `Node` struct is padded to multiples of 64 byte for good cache alignment.

The `bounds` store the AABB of the child nodes in Structure of Array (SoA) order for fast SIMD access. If the current node is a leaf this space is not used. The first element of the `bounds` array is set to NaN if the node is a leaf. The exact data layout of AABBs is explained later in this section.

The `idBegin` variable is either the index of the first child node or the index to the first triangle, depending on the type of the current node. The ids of the other children or triangles are not stored, since they are adjacent in memory.

`TraversalOrderEachAxis` describes the order in which the children should be traversed for the traversal of primary rays. It stores the order for each of the dimensions(x, y, z). The ray takes the child order of the dimension where its direction vector has the largest element. If the largest direction value is negative the stored order is applied in reverse.

The bitset `childType` is basically a compact Boolean array that indicates if the child of the same index is a node or a leaf. This is only required by the wide render to separate the different rays by the type of work they will compute in the next step. For the four different node sizes (4,8,12,16) of the performance tests, this bitset does not increase the size of the node, since it fits in the padding.

AABB layout example

The following explains some examples of the AABB layout in the `bounds` array.

The first example is the Array of Structure (AoS) order, that is usually used. The table shows an N4 node with 4 children in the first row and 2 children in the second row.

Min P0	Max P1	Min P2	Max P3	Min P1	Max P2	Min P3	Max P3
x_0, y_0, z_0	x_0, y_0, z_0	x_1, y_1, z_1	x_1, y_1, z_1	x_2, y_2, z_2	x_2, y_2, z_2	x_3, y_3, z_3	x_3, y_3, z_3
x_0, y_0, z_0	x_0, y_0, z_0	x_1, y_1, z_1	x_1, y_1, z_1	$0, 0, 0$	$0, 0, 0$	$0, 0, 0$	$0, 0, 0$

Short reminder: To store an AABB, only the edge closest to the origin and furthest away have to be stored. In the tables they are named Min and Max. X, Y, and Z are the dimensions and the subscript describes its AABB the variable belongs to.

For improved SIMD access the points are stored in Structure of Array(SoA) order. The following table shows the exact same nodes as the above table.

Min x	Min y	Min z	Max x	Max y	Max z
x_0, x_1, x_2, x_3	y_0, y_1, y_2, y_3	z_0, z_1, z_2, z_3	x_0, x_1, x_2, x_3	y_0, y_1, y_2, y_3	z_0, z_1, z_2, z_3
$x_0, x_1, 0, 0$	$y_0, y_1, 0, 0$	$z_0, z_1, 0, 0$	$x_0, x_1, 0, 0$	$y_0, y_1, 0, 0$	$z_0, z_1, 0, 0$

Since half empty nodes are relatively frequent, we add an optimization to store them more efficiently and save some computation. This is only for the case when the hardware SIMD is not as wide as the target node size. The theoretical example below describes how our algorithm would sore a N4 node with a SIMD width of 2 when there are only 2 elements in the node. It should be noted that our CPU does not support a native SIMD width of 2 and this was only chosen so the table has a presentable size. The following equation is used to determine the real size the AABB are stored with:

$$\texttt{storedNodeSize} = ceil(\texttt{childCount}/\texttt{SIMD}) * \texttt{SIMD} \tag{3.3}$$

For the example, this results in 2. It basically treats the bounds array as if the node was 2 wide and indicates it by storing `NaN` in the last elemen. The second last element indicates the actual width the data was stored with. This means that the traversal algorithm has to find the actual node size before computing the node intersection and then do a node intersection as if we had a node that was 2 wide.

The AABB memory would then look like this:

x_0, x_1	y_0, y_1	z_0, z_1	x_0, x_1	y_0, y_1	z_0, z_1	0,0,0,0,0,0,0,0,0,0	2	`NaN`

This optimization is an important performance improvement for wider nodes, since we mostly use SSE, but we still want to compute performance tests with node sizes of 8, 12, and 16. It also brings the performance closer to the performance of hardware with matching SIMD width.

When creating a BVH with a node size that is not a multiple of the SIMD width the node is then stored as if it had the next larger matching SIMD width. Like with partially filled nodes, the empty space is filled with zeroes.

Primitive Data

Similar to the Nodes, the triangle data of a Leaf is stored tightly packed together in one large array. Textures and other information are not required for traversal and shading.

The data structure for the triangles of one leaf consists of $3 * 3 *$ `LeafSize` Floats. This are the 3 edge points for each triangle.

The primitive data is also stored in SoA order for fast SIMD access. The leaves always have the same size, independent of the amount of triangles inside. The empty triangle parts are filled with zeros.

The same technique used to tightly pack half empty nodes is also used for leaves.

3.3 BVH Traversal

The BVH traversal is the performance critical part of the code, so its important that it is implemented efficiently.

Since instrumentation adds too much overhead, there are multiple version of each traversal algorithms, an optimized renderer for performance benchmarks, an instrumented renderer for all intersection results, and a renderer for cache simulations. Since there is no randomness in traversal and shading, it is also simple to verified that all algorithms have the results.

In order to to write high performance code C++ templates are used for the different the node sizes and work group sizes. This is required so that the size of some of the data structures and arrays is known to compile time.

In this section we will first discuss the parts that are shared by single ray traversal and wide traversal and then go into detail how both algorithms work.

3.3.1 Parallelism

The image is divided in work groups of either 8×8, 16×16 or 32×32 rays. OpenMP [11] is used to compute these work groups in parallel:

```
1 #pragma omp parallel for schedule(dynamic, 4)
2 for (int i = 0; i < (width / workGroupWidth) * (height / workGroupHeight); i++)
3 {
4   ray creation and BVH traversal
5 }
```

This means that the OpenMP compiler generates code that bundles 4 work groups together and then executes them in a parallel fashion. Dynamic scheduling is used because the different rays and work groups have highly varying workloads depending on the complexity of the current area to render.

It should be noted that height and width of the image must be dividable by 32 in order to avoid edge cases.

3.3.2 Rays

The memory layout of rays. For wide traversal this is important to be compact since every ray in the work group needs to be saved and they are all cycled trough for every step.

```
1 struct Ray{
2   float tMax;
3   glm::vec3 origin;
4   glm::vec3 invDirection;
5   glm::vec3 direction;
6 };
```

The `direction` is required for the triangle intersection and the `invDirection` for the AABB intersection. The `invDirection` must not be NaN in any dimension, so when the `direction` is 0 the `invDirection` of the respective dimension is set to the largest possible floating point number.

The `origin` is the center of the camera and the point from where the ray is cast into the scene.

The maximum distance of the ray is stored in `tMax`. This is the only ray variable that changes during the traversal.

The primary rays origin and direction can be calculated by the pixel index and the camera position and direction. For each pixel, one ray is spawned for the center of the pixel. The maximum ray distance of primary rays is not limited.

For the secondary ray we sample the hemisphere of the surface hit by the primary ray. The randomly sampled direction is deterministic and computed with the pixel index and sample step. This means that every rendered image always looks the same, as long as the number of ambient samples is the same. The maximum ray distance depends on the size of the scene and is $\sqrt[3]{\texttt{Volume}}/10$, with `Volume` being the volume of an AABB containing the scene. The secondary rays origin is moved 0.001 units along the surface normal in order to prevent the ray from hitting the surface it spawned from.

3.3.3 SIMD Implementation

The general idea of the thesis is to see what BVH configuration would be the best for ray tracing with arbitrary SIMD width. The reference implementation to verify the results from the instrumented renderer runs on the CPU, so it is limited to SSE or AVX, meaning 4 wide or 8 wide SIMD. So for SSE the matching configurations that can be tested are 4, 8, 12, and 16.

To simplify the process of writing SIMD code we use ISPC [3]. It allows us to write C-based code that is easily understandable and has a very similar structure as writing GPU shaders. The code is then compiled by ISPC to utilize SIMD instructions. An additional feature is that one can choose between SSE and AVX in the compiler settings, so it is very simple to swap between them.

For general variables in ISPC code there are two important keywords to explain, uniform and varying. A uniform variable exists once in memory and all SIMD lanes share it. A varying variable has one instance for every SIMD lane.

ISPC offers an simple for writing foreach loops that are executed with SIMD. As an example, a foreach loop over 16 elements, compiled with SSE, is transformed into to four iterations that each execute four lanes in parallel.

The code for the node and leaf intersection is inspired by a ray tracer implementation in the ISPC examples [4]. Their implementation is also based on PBRT [12], but their approach uses SIMD to compute multiple rays in parallel. This means the structure of the intersections has to be modified to instead compute multiple AABBs or triangles with SIMD, but the code to solve the geometrical problem of intersecting rays is mostly the same.

SIMD loops over a number of iteration that is given as a function parameter perform

significantly worse than loops were the number of iterations is know to compile time. This is due to a limitations in the optimizer of ISPC. Since ISPC does not support C++ like templates the template engine Jinja2 [13] is used to generate methods for intersections with hard coded node and leaf sizes.

Loops that are not a multiple of the SIMD width should be avoided.

Node Intersection

A node intersection is the process of testing a ray against all child AABBs of a node in the BVH. It is the most common operation of the ray tracer. To accelerate the intersection process SIMD is used to test one ray against multiple AABBs at once. The node and leaf intersections are the part of the ray tracing process that would be implemented as fixed function hardware. The intersections are very simple geometric problems and therefore well suited for a hardware implementation. The exact implementation and structure used in this paper is optimized for the CPU, so a fixed function hardware version would most likely implement it a bit differently.

They input to a node intersection is one Ray, a pointer to the bounds array inside the node and a pointer to the array where the intersection results are stored in.

The intersection itself consists of a ISPC foreach SIMD loop over each element in the node. If any AABB was hit the function returns true and and the exact hit can then later be read out of the intersection result array.

Leaf Intersection

There are two different types of leaf intersections, those for primary rays and those for secondary rays.

The input of a leaf intersection is a pointer to the triangle data and to the ray data. The intersection returns an integer, either the triangle index of the successful intersection or -1 if nothing was hit.

The major difference to node intersection is, that instead of all collisions only the closest one is needed.

This means that only a varying variable is required to store the intersection distance. This variable is initialized with a very large float. During the SIMD loop each successful intersection stores the collision distance in the variable if it is smaller than what is stored there. This is important since a leaf size of 16 requires 4 iterations with SSE and triangle hits from previous iterations should not be overwritten when they are closer than the new hit.

After the foreach a SIMD reduce min operation is called to get the smallest distance in the varying float variable. This distance is used to update the maximum distance of the ray and the index of the closest hit triangle is then returned by the function.

For secondary rays the triangle idndex is not needed, so the method only returns a boolean to indicate if anything was hit. This also means that it is not required to update the maximum ray distance. The overhead to stop the SIMD loop early if a triangle was hit it is larger than the performance gain.

3.3.4 Single Ray Traversal

The single ray traversal algorithm is very simple and straight forward.

Every ray of the work group traverses the tree individually and the next ray only starts when the previous ray finished.

```
//first primary rays:
prepare memory for primary ray results;
for every ray id in the work group do
    create the primary ray;
    initialize nodeStack with index of root node;
    while nodeStack not empty do
        workNode = top element of nodeStack;
        if workNode is leaf then
            Compute primary ray Leaf Intersection;
        else
            Compute Node intersection;
            update nodeStack according to result;
        end
    end
    update primary ray results with last triangle hit;
end
```

Algorithm 4: Single Ray Traversal, Primary Ray

The `nodeStack` is a stack that contains the ids of all nodes and leaves that still have to be intersected. Internally the stack is implemented as an fixed size array of 48 elements and an index to indicate the size. For most BVH configurations a stack size of 32 elements would suffice, but for wide nodes as N16L16 the complex scenes can require more. When adding elements to the stack the traversal order stored in the node has to be used.

The algorithm for the secondary rays is very similar:

```
//Go through secondary rays to compute ambient occlusion:
for Ambient sample count do
    for every ray id in the work group do
        create the secondary ray according to primary ray results;
        initialize nodeStack with index of root node;
        while nodeStack not empty do
            workNode = top element of nodeStack;
            if workNode is leaf then
                Compute secondary ray leaf Intersection;
                if hit any triangle then
                    //this ray is finished
                    break;
                end
            else
                Compute Node intersection;
                update nodeStack according to result;
            end
        end
        update primary ray results with last triangle hit;
    end
end
```

Algorithm 5: Single Ray Traversal, Secondary Ray

For secondary rays the traversal order stored in the nodes is ignored since it gives no particular benefit for secondary ray traversal.

3.3.5 Wide Traversal

Wide traversal is more complex since all rays are traversed at once.

In order to store the states of all rays of one work group this approach requires some additional local memory. This will often be called memory overhead of the wide traversal algorithm:

```
1 int stackSize = 48;
2
3 //wide data:
4 Ray rays[workGroupSize]
5 int32_t nodeStack[workGroupSize * stackSize]
6 uint16_t currentWork[workGroupSize];
7 uint16_t nextWork[workGroupSize];
```

`WorkGroupSize` is the number of rays in the work group, usually 16×16.

The `nodeStack` has the same function as in single ray traversal but for all rays in the work group. Since the different rays traverse a similar path through the BVH they should also have similar full stacks. To improve the cache access it is ordered as 48 arrays of `workGroupSize` large arrays. This means that when all node stack have the same size it only requires 16 cache lines to load the topmost element for each ray instead of 256. In order know what type of work has to be performed for the stored index the nodes are stored with positive indices and the leaves with negative indices.

`currentWork` and `nextWork` are arrays that indicate what rays will do node intersections or leaf intersections. One is the array that is currently worked on and the other is what will be worked on in the next step. Each arrays has the indices of rays that do node work as a stack growing from the left and the indices of the nodes that do leaf work as a stack growing from the right. This means that we also have to save how many node and leaf elements are in the array. The `currentWork` array is initialized with increasing numbers from zero to `workGroupSize` to indicate that all rays will do a node intersection in the next step. Both arrays are swapped after each step over all rays.

When a ray is finished its index will no longer be stored in `nextWork`.

```
//Primary Rays:
prepare memory for primary ray results;
Initialize all primary rays of the work group;
Initialize nodeStack of every ray with index of the root node;
Initialize currentWork and nextWork;
while currentWork is not empty do
    for each ray index that will do node work in currentWork do
        get current ray according to index;
        Compute node intersection;
        Update nodeStack of ray the ray with intersection results;
        update nextWork of ray index;
    end
    for each ray index that will do leaf work in currentWork do
        get current ray according to index;
        Compute primary ray leaf intersection;
        update nextWork of ray index;
    end
    Swap currentWork and nextWork;
end
```

Algorithm 6: Wide Traversal, Primary Rays

For the `update nextWork` line the sign of the topmost element on the `nodeStack` of the current ray is checked. If the index is positive the index of the ray is added to the node part of the `nextWork` array, otherwise its added to the leaf part. When a ray has no element in the `nodeStack` the ray is finished and not added to `nextWork`.

```
//Secondary Rays:
for each Ambient sample do
    Initialize all secondary rays of the work group;
    Initialize nodeStack of every ray with the index of the root node;
    Initialize currentWork and nextWork;
    while currentWork is not empty do
        for each ray index that will do node work in currentWork do
            get current ray according to index;
            Compute node intersection;
            Update nodeStack of ray the ray with intersection results
            update nextWork of ray index;
        end
        for each ray index that will do leaf work in currentWork do
            get current ray according to index;
            Compute primary ray leaf intersection;
            if no triangle hit then
                update nextWork of ray index;
            else
                //Not update nextWork. This means that the ray index is no
                  longer referenced and therefore not picket up in the next
                  iteration, since it is finished.
            end
        end
        Swap currentWork and nextWork;
    end
end
```

Algorithm 7: Wide Traversal, Secondary Rays

Similarly to single ray traversal the stored traversal order of nodes is not used for secondary rays.

Chapter 4

Evaluation

This section will first go over several properties of the different BVH configurations and explain their possible effects on ray tracing. Then the actual properties of ray tracing the BVHs are investigated with the instrumented renderer and the performance benchmarks. Lastly the cache simulator is introduced and its results are presented.

In order to evaluate the effect of a BVH configurations it is used to render multiple scenes.

For the different benchmarks the following scenes are used:

1. Sponza. 262,267 triangles. Despite being the smallest, it is a quite complex scene that many node and leaf intersections for its size. Sponza is a very common benchmark scene.
2. San Miguel. 5,600,315 triangles.
3. Gallery. 998,822 triangles. This is a scan of the Hallwyl Museum in Stockholm. Since its a scan the topology is different to the other, artist made, scenes.
4. Bistro Interior. 1,020,903 triangles. It is the main testing scene since it is a good game ready model that offers different challenges for ray tracing.
5. Bistro Exterior. 2,833,257 triangles. Similar to the interior scene.

All models were downloaded from Morgan McGuire's Computer Graphics Archive [10] and then converted to glTF.

The camera is placed inside the scene, facing complex geometry. It is mostly avoided to show much background. Showing background results in primary rays missing, what means that there is no secondary ray.

The maximum distance of the ambient rays depends on the size of the scene. The exact formula is the following: $\sqrt[3]{\texttt{Volume}}/10$, with `Volume` being the volume of an AABB containing the scene. This means that it is approximately 10% of the average length of the scene.

The render resolution for performance benchmarks is $3840 \times 2176 = 8355840$ pixels and for intersection and cache measurements it is $1920 \times 1088 = 2088960$ pixels. These are slightly larger than the usual Full Hd or 4K resolutions in order to have an image that is dividable in 32 x 32 chunks. The reason for the large resolution for the performance benchmarks is to give the CPU enough work, so the results are a bit more consistent.

In order to keep this chapter compact and well structured most graphs are averaged over all five scenes. For the BVH analysis and the intersection results the Appendix also shows the individual graphs of each scene.

The images in Figure 4.1 are rendered with 50 ambient samples. For the later analysis all images are rendered with 1 ambient sample. Increasing the sample count does not change the intersection results for the secondary ray since the sample size is already very large.

The hardware used for most performance benchmarks consists of an AMD Ryzen Threadripper 1950X @3.40 GHz (water cooled) 16-Core Processor and 64 GB ram with 2133 MHz.

It should be noted that some additional benchmarks were done with an Intel i7-6700k @4.00 GHz 4 core processor with 16 GB of ram. While the results are obviously slower compared ot the AMD CPU they all show the same trends, so most results should be applicable to other hardware.

The AMD CPU has 128-bit floating point registers. This means that SSE instructions can be executed with one operation, but 256-bit wide AVX instructions are implemented as two 128-bit operations. In order to keep the performance results simple we decided to only use SSE with this CPU. The Intel CPU supports 256-bit floating point instructions, so it is used to demonstrate the effect of increasing the node size.

4.1 BVH Analysis

For BVH configurations we use the following abbreviations: node size of four is N4 and leaf size of four is L4. This BVH would then be called N4L4. The usually tested node sizes go form N2 to N16 and for leaf sizes its L1 to L16. With all combinations

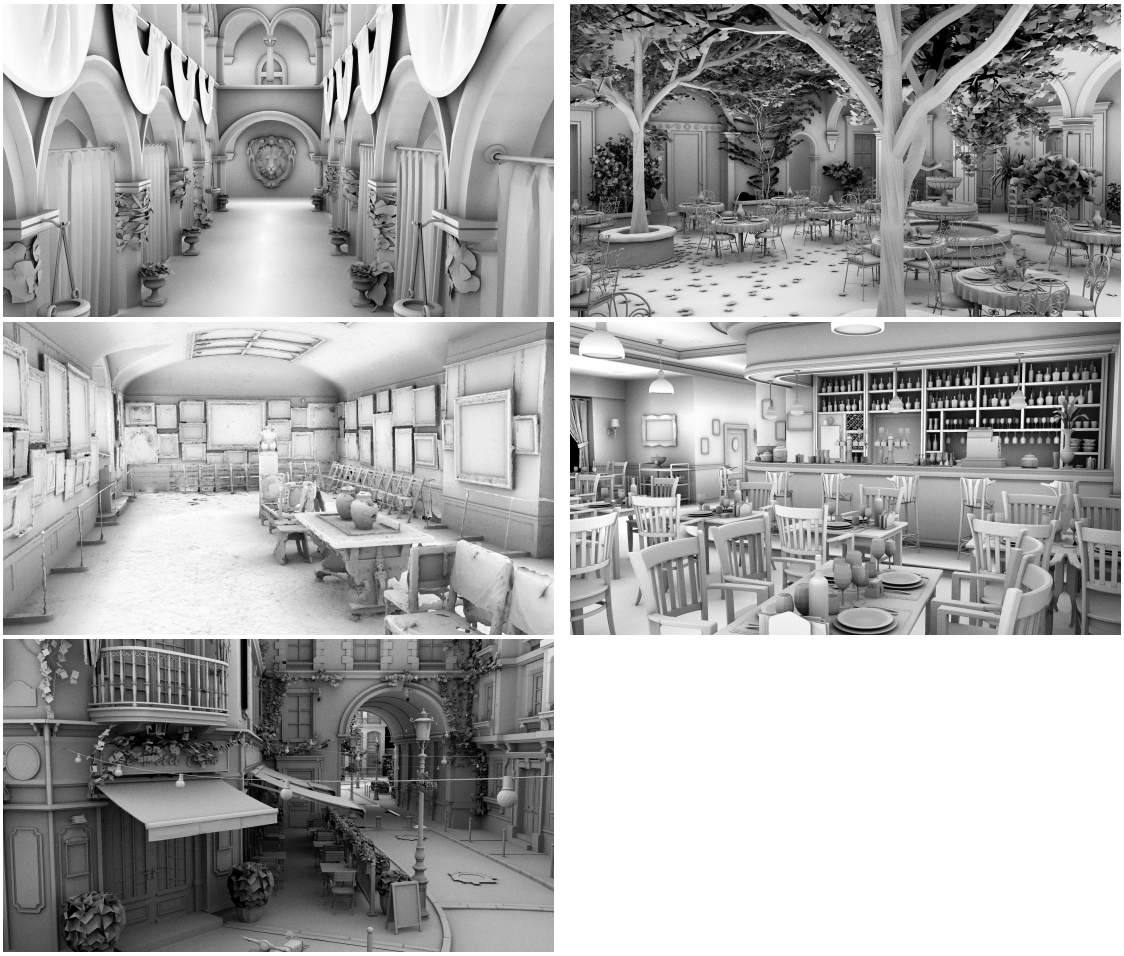


Figure 4.1: All scenes. From left to right, top to bottom: Sponza, San Miguel, Gallery, Bistro Interior, Bistro Exterior

this results in 240 different BVH configurations for every scene.

Firstly we will look at the effects of different leaf and node sizes on the BVH itself.

Figure 4.2 shows the average number of nodes and leaves over all scenes for the different BVH configurations. Before computing the averages of the values each scene is normalized by its N2L1 result, otherwise the graphs would be dominated by larger scenes like San Miguel. Due to the structure of a binary three the N2L1 has the same amount of nodes and leaves. Increasing the node size significantly reduces the amount of nodes. Increasing the leaf size reduces the amount of leaves and therefore also reduces the amount of nodes. The interaction between the node size and the number of leaves is a bit more complex. Normally increasing the number of children for nodes should not change much for the leaves. The reason that the number of leaves changes for all configurations expect L1 is leaf splitting as described in Section 3.2. The visible spikes at N4,L8,L12 originate from a change of the `nodeCostFactor` and L1 is constant since a leaf with one primitive cannot be split.

The node and leaf fullness graphs in Figure 4.2 show how the nodes and leaf storage is utilized. The fullness describes how many elements are in the node, compared to how many could be in it. As an example, a node that has 3 children in a N4 BVH has a fullness of 75%. Since the storage space and the compute cost is independent of the fullness it is generally preferred have full nodes, as long as it has no negative impact on the ray tracing results.

An interesting property of the BVH is the surface area of the leaves. The core concept of the surface area heuristic (SAH) is that a node with an increased surface area results in a higher chance to hit the box with a ray. Figure 4.3 shows that with increasing leaf size the sum of the surface area of the leaves decreases.

In addition to the volume we can also look at the overlap of nodes and leaves. We use a metric called end point overlap (EPO) [2]. It is designed to supplement the SAH by penalizing triangles that overlap with nodes of other sub trees. Overlapping nodes and leaves are something that should be avoided since it forces the ray tracer to traverse into both sub trees. Our expectation was that with increasing node and leaf size the overlap might be more prominent, but the opposite is the case, as shown in the lower part of Figure 4.3.

An interesting effect with the graphs in Figure 4.3 is is that even leaf sizes are often better than odd numbers. This behavior originates from the internal triangle structure of the scene. Most of the scenes have a topology that originally consisted of quads, that were later split into triangles to render them. Two triangles that previously were a quad share one edge and are often shaped in a way that one AABB

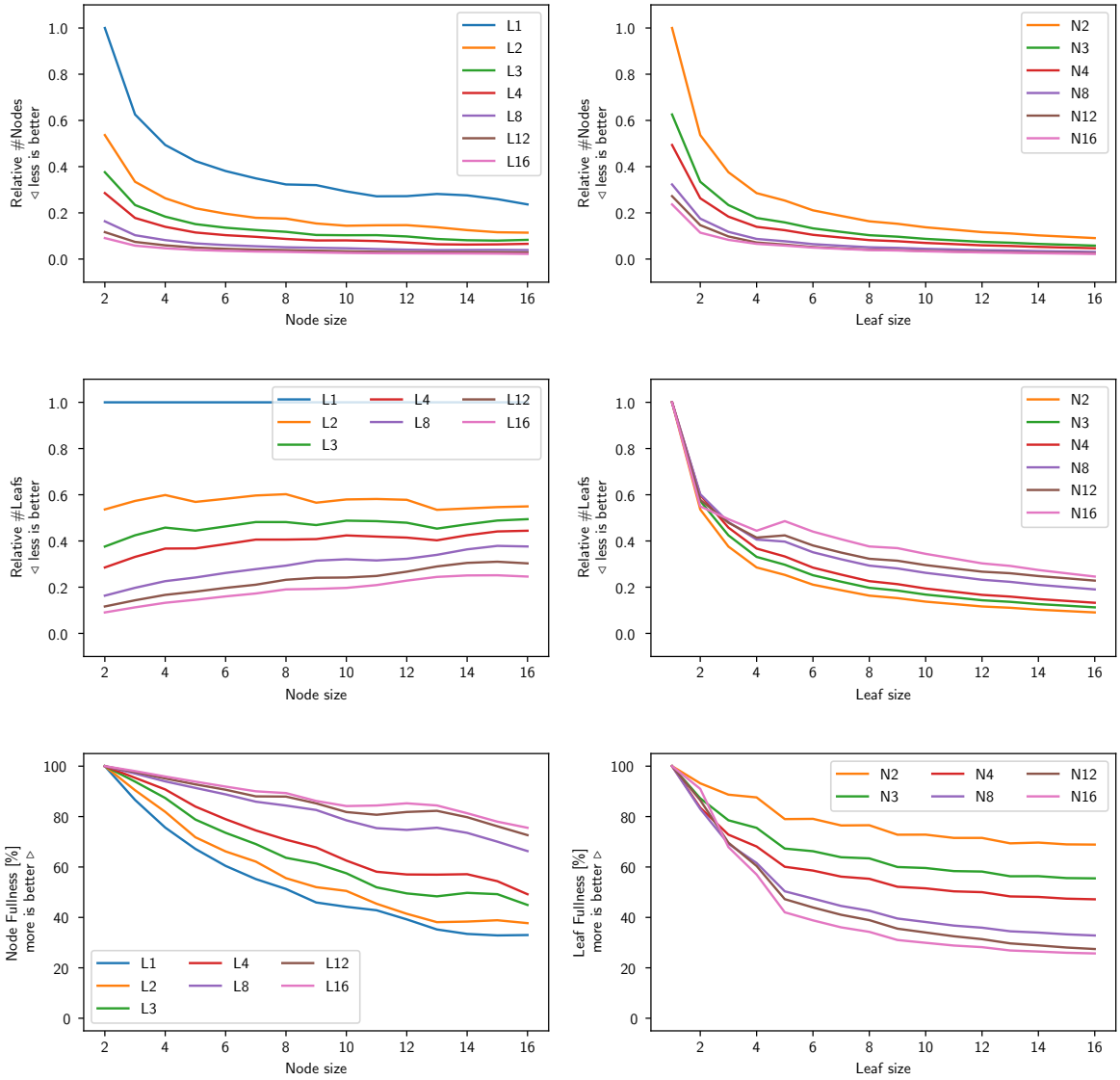


Figure 4.2: Overview of the effect of different node and leaf sizes on the number of nodes and leaves. All graphs are normalized to their respective N2L1 value before averaging over the different scenes.

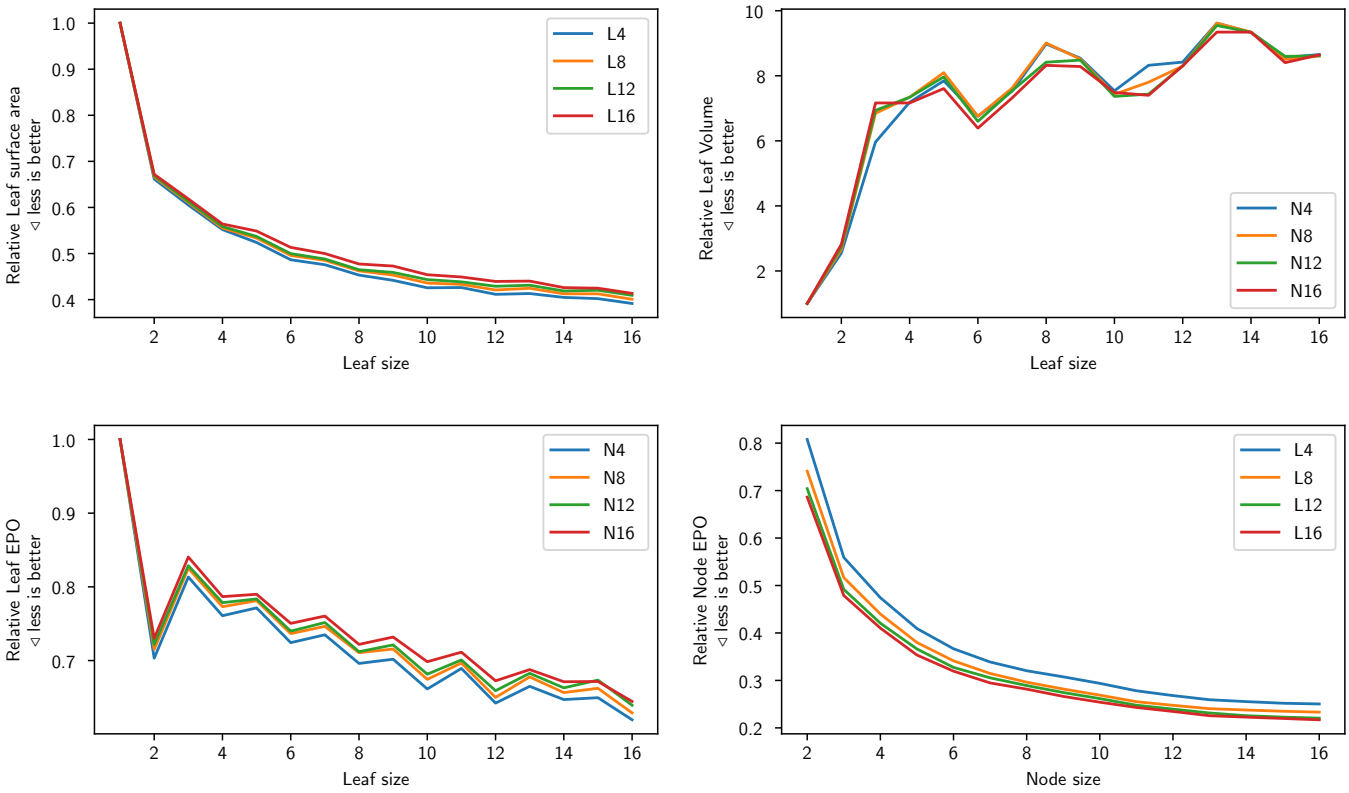


Figure 4.3: BVH properties. All scenes are normalized to their respective N2L1 value before averaging.

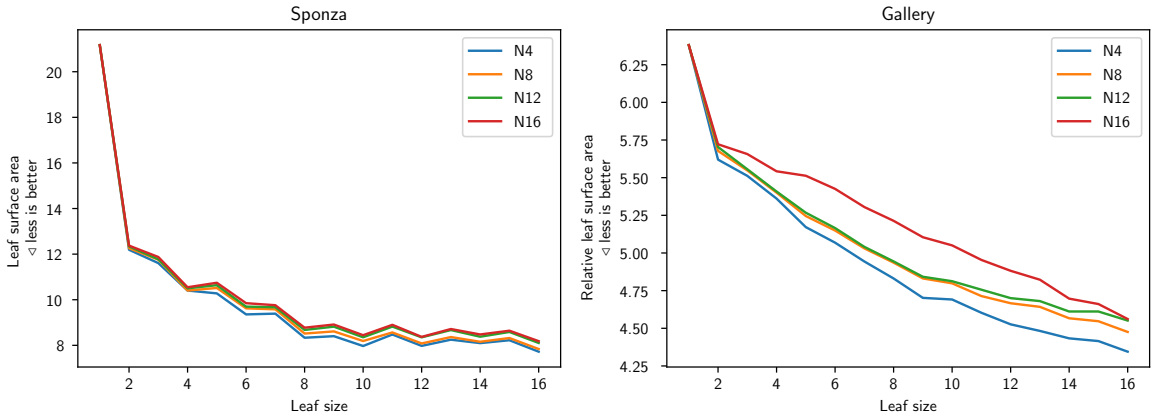


Figure 4.4: Comparison of the surface area of Sponza and Gallery. The surface area is normalized by the surface area of the AABB containing the respective scene.

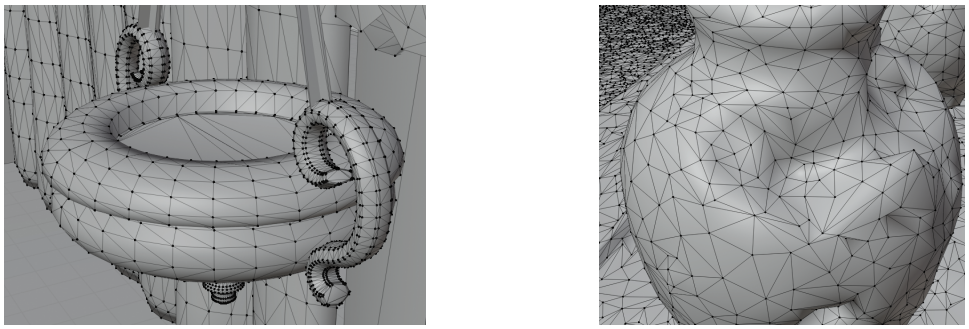


Figure 4.5: Topology of Sponza (left) and Gallery (right)

surrounding them both would only be slightly larger then two individual AABBs. Therefore a L1 BVH would have severely overlapping leaves. For larger odd leaf sizes a similar overlapping can also occurs. This can be shown by comparing the Gallery scene with Sponza. The Gallery scene is a computer generated scan which triangles resemble no quad structure, while Sponza mostly consists of quads, see Figure 4.5. The effect of this is also very prominent in the surface area of Sponza and Gallery, see Figure 4.4. For most artist made models that have quad structure a leaf size of one should be avoided and it is generally better to choose an even leaf sizes in order to take advantage of the topology. nt

An other important property of the BVH that indicates the approximate traversal workload is the average depth of the BVH tree. It allows comparison of how many node intersections have to be performed on average to hit the first leaf, under the assumption that the traversal always chooses the perfect path. There can not be a shorter way to the leaf intersections than going through the BVH. In Figure 4.6 it can be seen that the increasing the leaf size slightly reduces the depth it reduces the

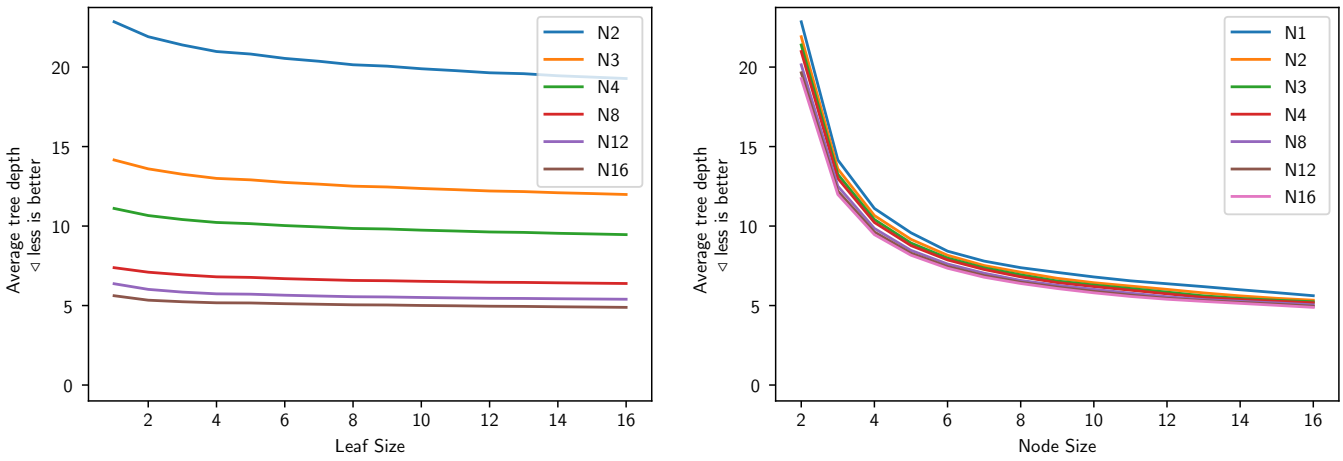


Figure 4.6: Average tree depth over all scenes

amount of leaves. The node size has significantly more impact. Doubling the node size form N2 to N4 approximately halves the depth of the tree, independent of the leaf size. Further increasing the node size is less of an improvement and after N8 it barely reduces the depth of the tree.

4.2 Instrumented Renderer Results

In addition to the performance renderer described in the Implementation Section there are also instrumented traversal implementations to gather statistics about node and leaf intersection.

Those renderer collect different properties of rendering process. The most important information is the number of node and leaf intersections, since they directly represent the amount of work required to render image. An intersection is the process of testing a ray against a node or leaf. A node intersection with a node size of 4 can include up to 4 AABB tests. While SIMD is used to compute those 4 AABB - ray intersections in parallel the memory required for the computation is still required.

The instrumented renderer also gathers some statistics like the average fullness of nodes and leaves and the success rate of aabb and triangle intersections. Most of the data is collected separately for primary and secondary rays since they represent have a fundamental different problem set.

The intersection results from the different scenes are relatively similar and therefore it is not necessary to normalize them before averaging.

Figure 4.7 gives an overview of the primary ray intersection results. The different

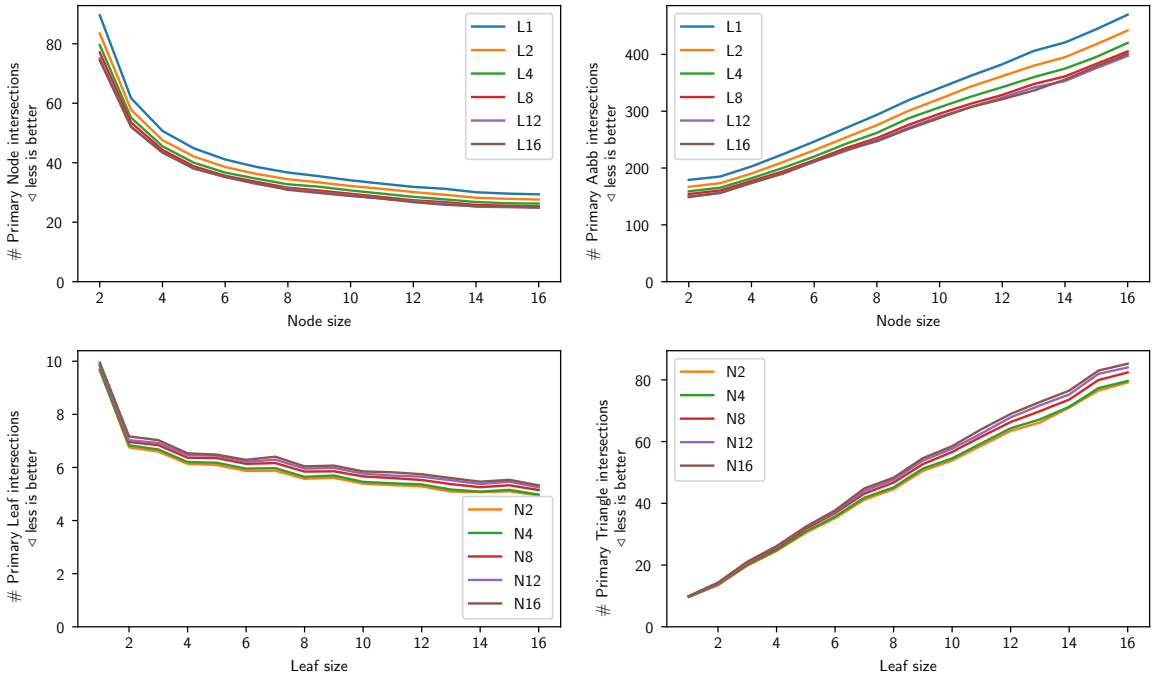


Figure 4.7: Primary ray intersection results, average of all scenes.

node sizes have basically no impact on the number of leaf intersections and the leaf size slightly reduces the amount of node intersections since it also reduces the number of nodes. While increasing the node size reduces the number of node intersections it also increases the amount of overall tested AABBs. This value represents the actual amount of work that is done and how also much memory has is required for the BVH traversal.

For leaves the biggest improvement for the intersections is the step from L1 to L2, further increasing the leaf size has less of an effect. Similarly to the nodes the number of triangle intersections increases with leaf size.

The Table 4.1 and 4.2 show the exact effect of increasing the node or leaf size. Both leaf and node intersections have diminishing returns from increasing the size, while the AABB and triangle intersections mostly increase linearly. Overall the decision can be described as a trade off between performance and low memory bandwidth. This suggest that if we could build a fixed function hardware with arbitrary SIMD width the performance should increase with increasing SIMD width until some memory bottlenecks are reached. The best SIMD width would then depend on the available memory bandwidth. Some aspect of this topic will be further discussed in Section 4.4 where the cache simulator and its results are explained.

Node Size	# Node Intersections	Node Change	# AABB Intersections	AABB Change
N2	79.55	0.00 %	159.09	0.00 %
N3	55.11	-30.72 %	165.34	3.93 %
N4	45.52	-17.40 %	182.09	10.13 %
N5	40.01	-12.12 %	200.04	9.85 %
N6	36.68	-8.31 %	220.09	10.02 %
N7	34.62	-5.62 %	242.34	10.11 %
N8	32.74	-5.43 %	261.93	8.08 %
N9	31.93	-2.49 %	287.33	9.70 %
N10	30.68	-3.91 %	306.77	6.77 %
N11	29.60	-3.51 %	325.62	6.14 %
N12	28.53	-3.63 %	342.34	5.13 %
N13	27.69	-2.95 %	359.93	5.14 %
N14	26.78	-3.26 %	374.98	4.18 %
N15	26.37	-1.56 %	395.52	5.48 %
N16	26.26	-0.39 %	420.23	6.25 %

Table 4.1: Node and AABB intersection overview for L4. The change describes the relative change of the intersection number from the next smaller node size

Leaf Size	# Leaf Intersections	Leaf Change	# Triangle Intersections	Triangle Change
L1	9.66	0.00 %	9.66	0.00 %
L2	6.83	-29.29 %	13.67	41.43 %
L3	6.68	-2.31 %	20.03	46.53 %
L4	6.20	-7.08 %	24.81	23.89 %
L5	6.18	-0.44 %	30.88	24.45 %
L6	5.95	-3.58 %	35.73	15.70 %
L7	5.97	0.31 %	41.81	17.02 %
L8	5.65	-5.46 %	45.17	8.04 %
L9	5.70	0.87 %	51.26	13.48 %
L10	5.46	-4.20 %	54.56	6.44 %
L11	5.40	-0.95 %	59.45	8.95 %
L12	5.36	-0.82 %	64.32	8.20 %
L13	5.17	-3.58 %	67.18	4.45 %
L14	5.09	-1.51 %	71.26	6.07 %
L15	5.15	1.27 %	77.32	8.50 %
L16	4.98	-3.45 %	79.63	2.98 %

Table 4.2: Leaf and triangle intersection overview for N4. The change describes the relative change of the intersection number from the next smaller leaf size

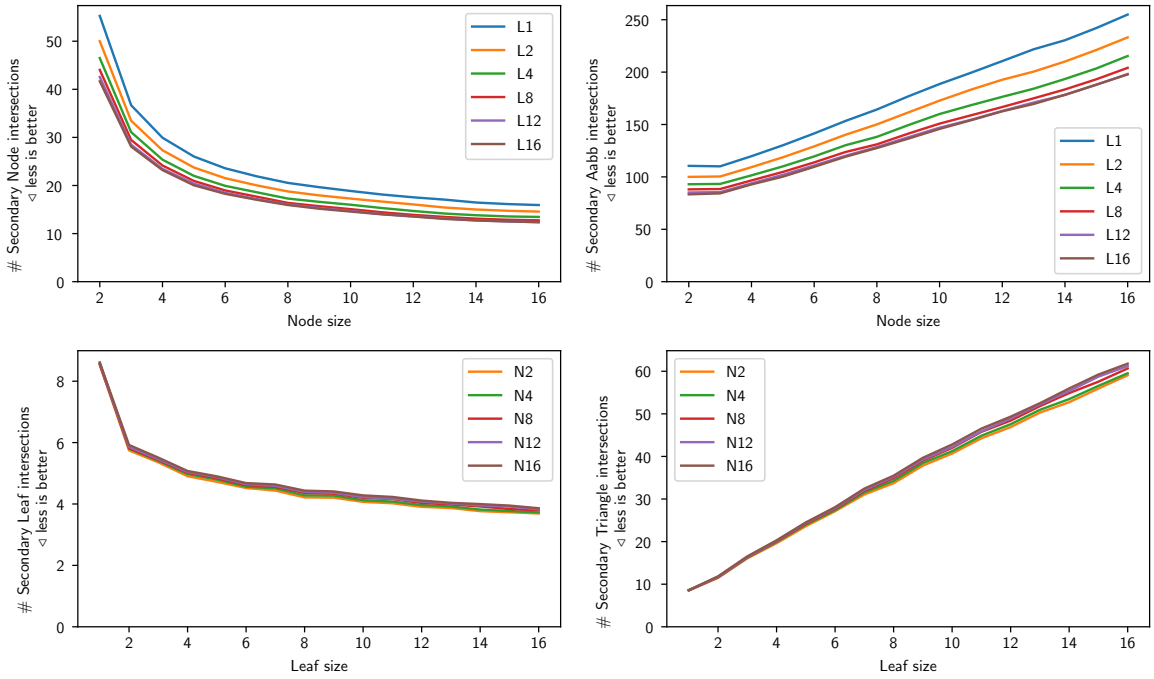


Figure 4.8: Secondary ray intersection results, average of all scenes.

For the secondary rays the intersection results follow the same trends as for the primary rays, as seen in Figure 4.8. The secondary rays have approximately 55% less node intersections and 20% less leaf intersections than the primary rays. While the secondary rays generally have the simpler task of an any hit search the way they are spawned makes them relatively costly. They start from an existing surface, so its quite likely they have to compute an intersection with the leaf they were spawned from. Since the direction sampling is also in the random hemisphere without any weighting it is relatively likely that they have multiple near misses with leaves.

During the BVH analysis Figure 4.2 showed the node and leaf fullness of the BVH tree. In comparison, Figure 4.9 shows the measured nodes and leaf fullness during BVH traversal. The measured fullness for both nodes and leaves is significantly higher during the traversal. The reason for the large difference of node fullness is that all nodes in the upper part of the tree are always full. Only nodes at the bottom of the tree can be can be partially empty. This means that during BVH traversal the first few node intersections are always with full nodes, depending on the tree depth this is somewhere between 20 and 5 nodes. The measured leaf fullness is higher, but the difference is not as large as with the nodes. Here the reason is that it is more likely to hit larger leaves and leaves with more primitives are usually larger.

When the node and leaf size is equal to the SIMD width of the hardware the mea-

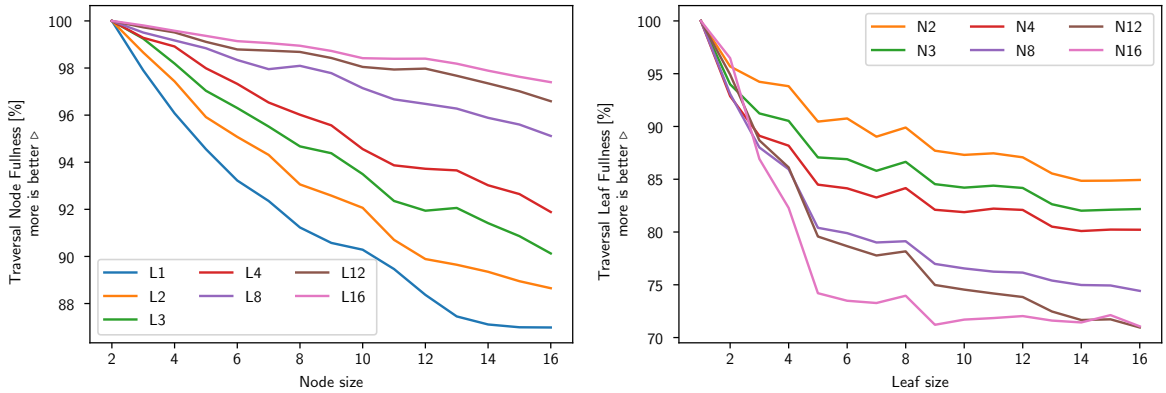


Figure 4.9: Average node and leaf fullness during the traversal

sured fullness can be seen as the SIMD efficiency of the hardware.

4.3 Performance Benchmarks

The performance measurements are computed with single ray traversal and wide traversal, as described in Section 3.3.4 and 3.3.5. For every BVH and traversal algorithm six images are rendered. The first render is to load everything into memory, so its results are discarded. For the final result the median of the next five render times is computed.

The function `std::chrono::high_resolution_clock::now()` is used to query the current time during rendering. It is called at the beginning and end of the code section to measure. For the main benchmark this means that the time is measured before and after the render of an image.

In order to measure the duration of node and leaf intersections, individually timers for intersections are needed. The lightweight approach is to only measure the duration of leaf intersections and the duration of the entire ray traversal. Since the ray traversal mostly consists of leaf and node intersections, the duration of node intersections can then be approximated. The problem is that every thread requires millions of time measurements for single traversal, so it has a significant performance impact. Since the wide renderer bundles multiple leaf intersections together, it only requires a fraction of the time measurements. It is still enough to decrease the overall performance and also alter performance tests of specific algorithms. As an example, when testing an algorithm that reduces the the number of leaf intersections, it might be an performance improvement with the individual timers enabled, but worsen the performance when the timers are disabled.

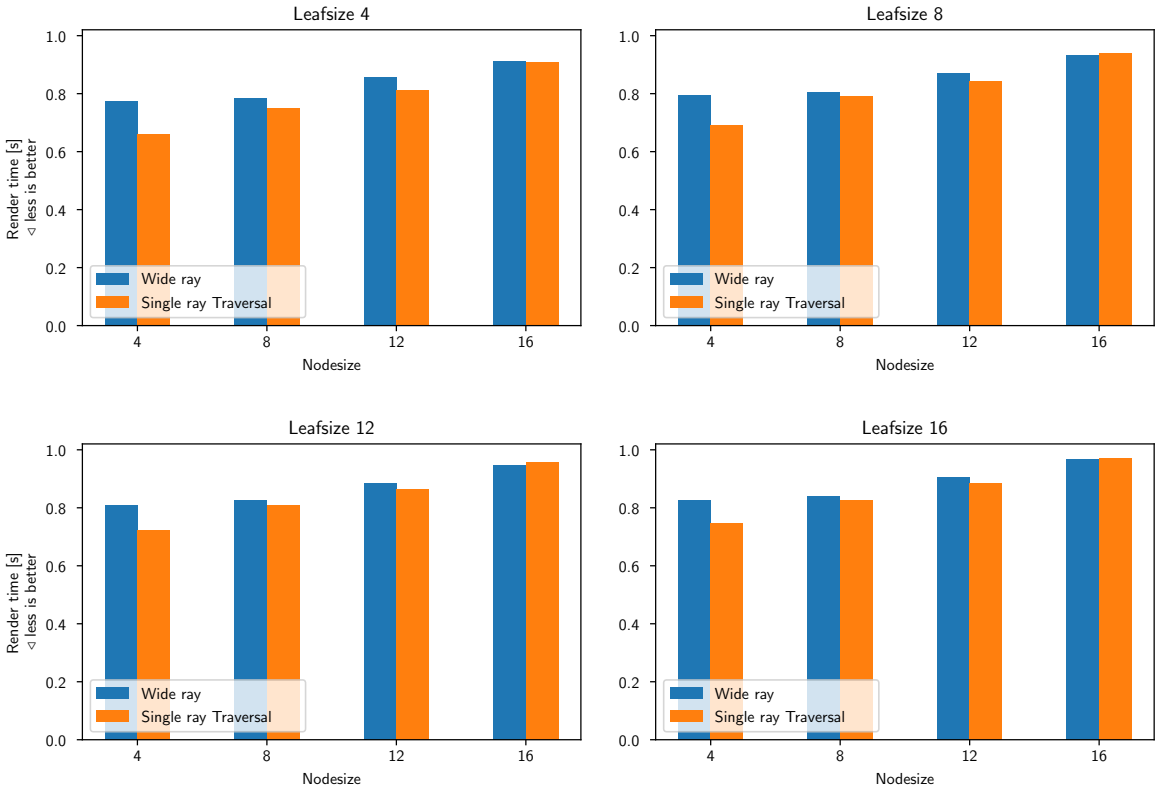


Figure 4.10: Performance overview of all scenes. AMD CPU.

So in order to get accurate time measurements, we only track the overall render time. The node and leaf measurements are only computed once to calculate the `nodeCostFactor` for the BVH creation.

For performance benchmarks multiples of 4 are used for the leaf and node sizes, so all 16 combinations of N4,N8,N12,N16 and L4, L8, L12, L16 are rendered. The goal of this benchmark is to compare the performance of wide traversal and single ray traversal. A work group size of $16 \times 16 = 256$ is used because it is the fastest for both rendering algorithms.

The results of the performance benchmark can be seen in Figure 4.10. It is computed on the AMD CPU with SSE. Both rendering approaches are the fastest for N4L4, what was expected for the CPU after the intersection results. While single ray traversal is faster for small node sizes the performance impact of increasing the node and leaf size is more sever than for wide traversal. For N16L16 Wide traversal is minimally faster than single ray traversal. The exact reasons for this will be explained in the cache analysis.

It is interesting that the performance for N4 and N8 are nearly the same for wide

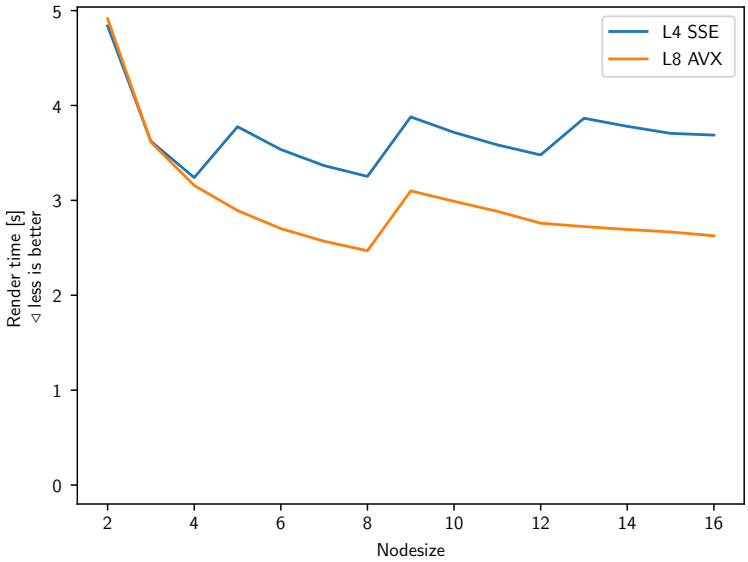


Figure 4.11: Comparison of SSE and AVX performance for Bistro Interior, rendered with wide traversal. Intel CPU.

traversal, even tough the CPU has to do double the work for each intersection step, since it uses a SIMD width of 4. This should mean that N8 would be faster than N4 with AVX.

To test this assumption a similar benchmark was computed with AVX and SSE. For this all node sizes between N2 and N16 are tested. The results are shown in Figure 4.11. This benchmark was computed on the Intel CPU mentioned in the beginning of the chapter. For the leaf size the matching one is chosen, so L8 for AVX and L4 for SSE.

It can be seen that increasing the node size improves the performance, as long as it does not increase the number of iterations of the SIMD loop. Without those steps the slope of both is very similar to the graphs shown in Figure 4.7, that show the number of node intersections depending on the node size.

Overall it can be said, that there is no BVH configuration that has any special properties and should be targeted. For hardware with a given SIMD width the best BVH configuration is the one that matches it for the node and leaf size. For larger nodes the memory bandwidth might be a limiting factor since the amount of different AABBs that are tested increases linearly with the node size.

On our test systems the renderer is compute bound so the increasing memory requirements are no issue.

The `nodeCostFactor` describes the cost of computing a node intersection compared to the cost of computing a leaf intersection. A `nodeCostFactor` of 0.5 means that one node needs half the time to be compute compared to a leaf.

`nodeCostFactor`				
node size \ leaf size	4	8	12	16
4	0.979	0.601	0.461	0.380
8	1.312	0.808	0.635	0.543
12	1.561	0.966	0.747	0.638
16	1.803	1.136	0.873	0.765

4.4 Cache Analysis

In order to better compare the two traversal algorithms we implemented a cache simulator for the Level 1 cache. It allows measuring the number of cache misses of both algorithms. To enable a fair comparison it has separate counters for cache misses from BVH memory and cache misses caused by the memory overhead of the algorithm. The size of the Level 1 cache is configurable in order to simulate how well the algorithms would behave with different hardware and also allow an estimation of the minimal cache size needed to efficiently run the algorithms.

According to the specification of most CPUs the Level 1 cache is implemented with an 8 way associative design. This means that the cache is organized into multiple sets that each hold 8 cache lines. The memory address of a memory block is used to determine the set it is stored in. For the cache replacement strategy inside the sets we decided to use tree pseudo least recently used (Tree PLRU) [7, pg. 14–15], since it is commonly used for modern CPUs [1]. It is basically a hardware efficient version of least recently used storage.

The general approach of the cache analysis is to have a simulated cache for every thread. During the BVH traversal, the cache simulator is notified of all relevant memory addresses that are loaded and then keeps track of what addresses are stored according to the policy and cache size. The number of the overall cache loads and the cache misses is separated for the memory loaded from the BVH and the memory overhead of the traversal algorithm. This is important since it allows to quantify effect of the wide renderer and also enables us to see how large the impact of storing the states of all rays is. The cache is cleared before executing each work group.

The following sections use cache lines as a unit for memory space. One cache line consists of 64 bytes. As a reference, our CPU has 512 cache lines in the Level 1 cache

per core. Since this values is per CPU core it means that two threads share the same cache. The implementation of the cache simulator does not include the cache sharing behavior since our main goal is to estimate how much cache is required for efficient BVH traversal and cache sharing would only add additional layers of complexity to it, making the evaluation of the results unnecessarily difficult.

BVH Memory

During the cache analysis the term "BVH memory" refers to the memory that is loaded from the BVH. It is required for node or leaf intersections during the traversal of a work group. The memory overhead is the the additional memory required to compute wide traversal.

The following tables shows the number of cache lines that have to be loaded for a node intersection or a leaf intersection.

Node Size	Node Cache lines
N4	2
N8	4
N12	6
N16	7

Leaf Size	Leaf Cache Lines
L4	4
L8	6
L12	8
L16	10

Both traversal algorithms perform the same intersections per work group and therefore also load the same cache lines. The only difference is that the wide traversal reorders intersection in a way that that consecutive intersections have a high chance of performing an intersection with the same node or leaf.

Wide traversal memory overhead

This subsection explain what cache loads are measured from the memory overhead caused by wide traversal. This is explanation uses work groups of the size $16 \times 16 = 256$.

One part is the memory to store all rays. The ray data is an array with 256 entries, with 10 floats for each ray, resulting in a total of 140 cache lines. For one step the algorithm has to cycles through all active rays and if they do not fit in the cache this generates 140 cache misses per step.

The `nodeStack` is the construct required to store the nodes and leaves that each ray still has to intersect. It consists of 48 arrays of 256 arrays of 4 byte integers. This would require 640 cache lines. Luckily each traversal step only requires the top element of the stack, so when all rays have the same amount of elements in the stack one cache line can contain the values 16 rays.

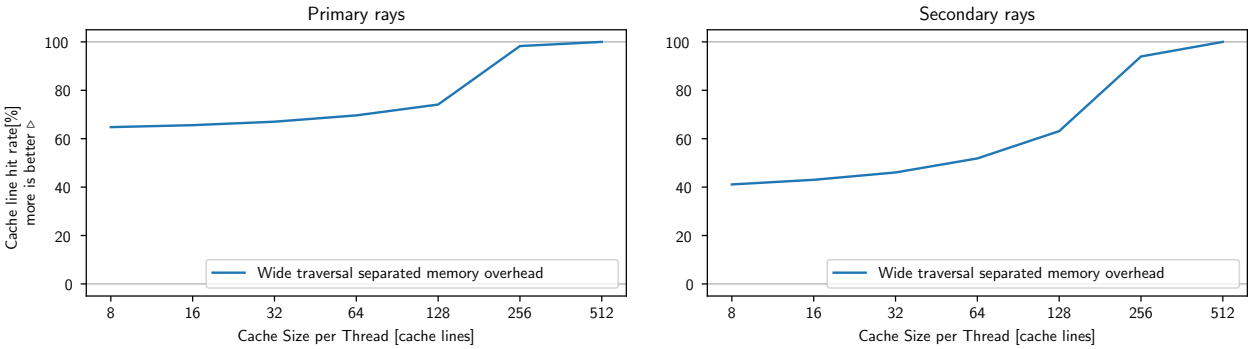


Figure 4.12: Cache hit rate on the memory overhead of wide traversal. This was simulated in an separated cache for clear results.

Figure 4.12 shows the cache behavior of the memory overhead. It is independent from the node and leaf sizes. The primary rays have higher hit rate, as expected, since secondary rays have slightly incoherent memory access for the `nodeStack`.

For single ray traversal the memory overhead of would be the node stack, therefore it is also included in the cache simulation and tracked separately, similarly to the wide traversal memory overhead. The results are not shown in the graphs since it has a relatively constant cache hit rate of about 99%.

Cache results analysis

In this section all graphs use a work group size of $16 \times 16 = 256$. Additional graphs with work groups of of $8 \times 8 = 64$ and $32 \times 32 = 1024$ can be found in the appendix. The smaller work group generally have better cache behavior for larger caches than 64 cache lines, but worse for smaller one. Single ray traversal results are the same for all work group sizes.

The Figure 4.13 demonstrates, that the approach of wide traversal to reorder the intersections to increase the cache reuse, works as intended. Wide traversal manages to achieve a 91% hit rate for BVH data of primary rays with a very small cache of only 8 cache lines (512 byte) for N4L4. This is surprisingly high since 8 cache lines can only store a total of 4 nodes or 2 leaves for this BVH configuration.

The hit rates for the secondary rays are relatively good for rays that go in random directions and therefore traverse in many different areas of the BVH. Here the cache size has a visible impact, so the traversal requires a larger pool of nodes than for the primary rays. It also happens more often that a visited node is later required in a future step. eave

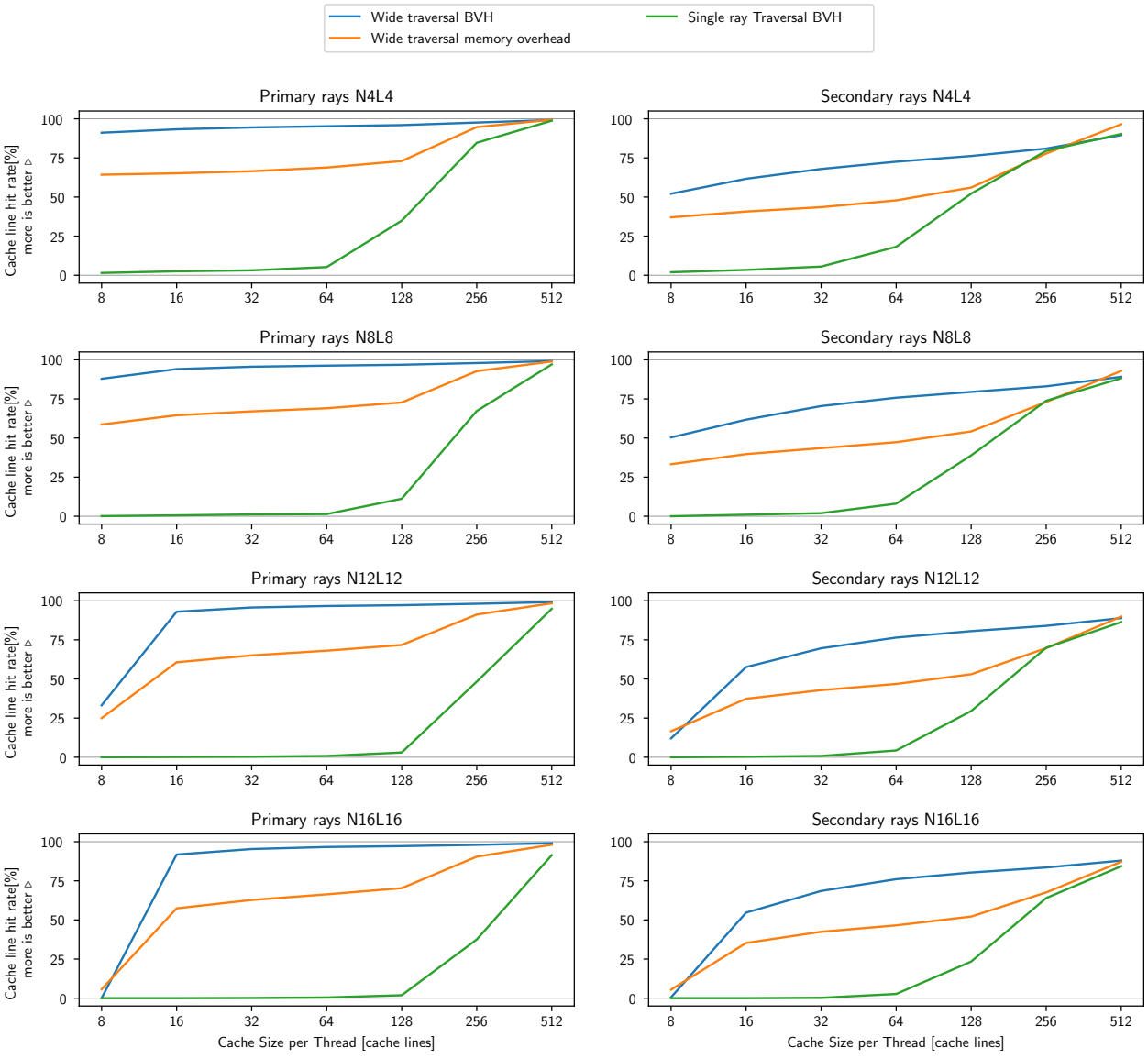


Figure 4.13: Overview of cache results for Bistro Interior. 16×16 work group

For single ray traversal the size of the cache is very important. When the cache is not large enough to keep all the nodes of one ray traversing the BVH all the nodes that could be reused by the next ray are evicted before they can be used again. Since there is no middle ground, single ray traversal has basically 0% hit rate once the cache is not large enough to store all the nodes and leaves of the traversal for the majority of the rays.

Single ray traversal has slightly better cache behavior for secondary rays than for primary rays since secondary rays have less intersections. For the Bistro Interior example each primary ray loads about 150 cache lines and each secondary ray loads 85 cache lines. Those rays that terminate early due to hitting a wall also have less than average steps. With N16L16 it requires about 280 cache lines for each primary ray and 140 cache lines for each secondary ray.

The reason why the single ray traversals performed better than wide traversal for the performance benchmark for smaller node and leaf sizes, but similar or worse on N16L16, is because the Level 1 cache of our CPU is large enough to contain all data needed for one traversal step. The cache size of our CPU would be somewhere between 256 and 512 cache lines for the graph since one cache is always shared by two threads.

Single ray traversal requires enough cache to store the entire BVH data needed to compute one ray. This means the minimum cache size for it to run well heavily depend on the properties and complexity of the scene that is rendered. For a real time environment this can mean that rendering a very complex scene that fills the cache could lead to 100% cache miss rate. This what would result in a significant performance drop.

In the Figure 4.14 the number of cache misses can be seen for the different cache sizes. For the cache analysis Bistro Interior was used, since it is the most complex scene.

The primary ray results of wide traversal are very good.

The incoherent secondary rays are basically the worst case scenario for wide traversal. At smaller cache sizes the algorithm still performs significantly better than single ray traversal and has similar results for larger caches. N4L4 is the only configuration for secondary rays where single ray traversal is better.

Single ray traversal has an increasing number of cache misses with larger node sizes, for the smaller cache sizes it is the same as number of AABB intersections in Figure 4.7 indicates.

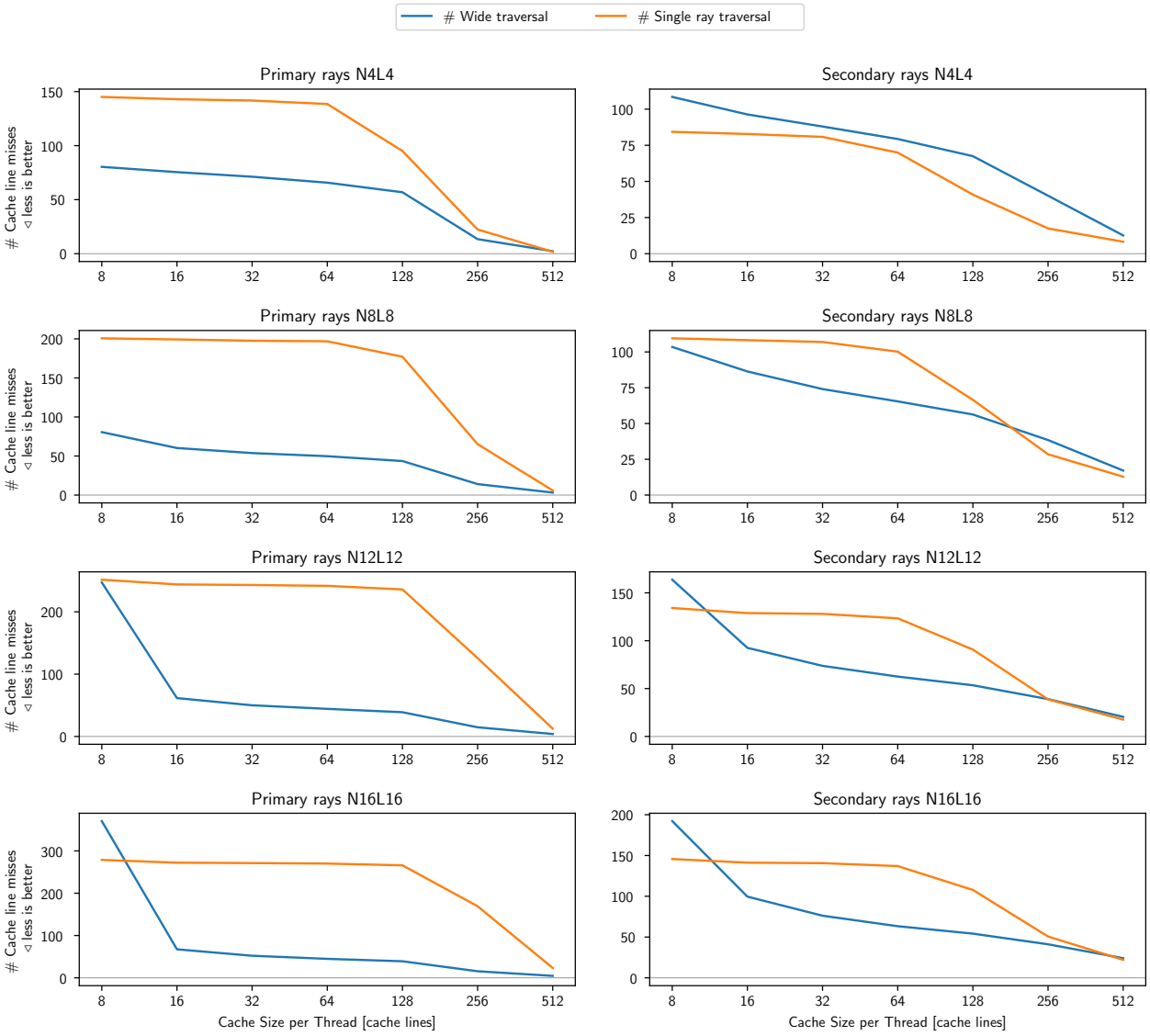


Figure 4.14: Number of Cache misses during the rendering, averaged per ray. Bistro Interior.

Wide traversals number of cache misses slightly improves for larger node and leaf sizes.

The largest part of the wide traversal cache misses comes from the memory overhead. The cache hit rate for the memory overhead is independent on the node or leaf size. This means that the number of cache misses depends on the number of intersections. Since the number of intersections decreases with node and leaf size the number of cache misses also decreases. This effect can be better seen in Figure 4.15. It shows the same data as Figure 4.14 but with the node and leaf size as X axis. An additional factor is that an increased node and leaf size makes the traversal through the BVH more coherent.

A visualization of the cache behavior for each individual ray of the work group for single ray traversal can be seen in Figure 6.18 and 6.19 in the appendix. Since the cache is cleared before each work group the first ray has a 100% miss rate. Each 16 rays a new row begins in the work group and the spacial jump slightly increases the cache misses, as it can be seen in the spikes at multiples of 16.

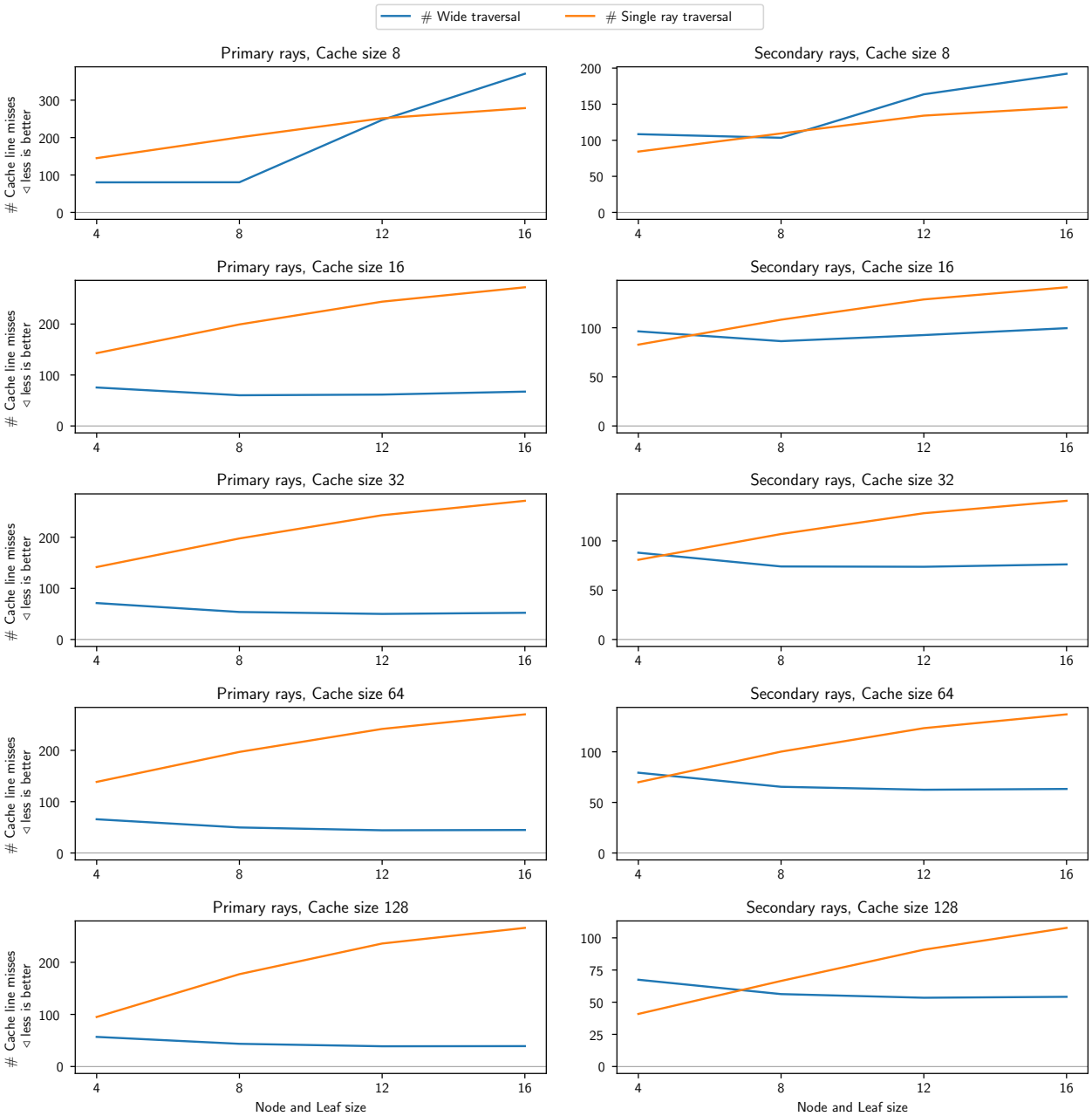


Figure 4.15: Number of Cache misses during the rendering, averaged per ray. Bistro Interior.

Chapter 5

Discussion

In this section we explain different improvements that could be implemented in the future and conclude this thesis with a summary of the results.

5.1 Future Research

GPU Implementation

A possible way to verify the intersection results would be a GPU implementation of the performance benchmark. Since the SIMD width of most GPUs is 32 this would allow us to test all interesting BVH configurations.

According to the intersection results, increasing the node and leaf size always improves the performance, until some memory bottlenecks of the GPU are reached.

Split BVH

An interesting approach to improve the intersection results is the splitting of triangles during the BVH construction [15]. It can improve the ray tracing behavior by splitting large primitives into multiple smaller parts that allow tighter bounding boxes and better grouping with surrounding geometry. This approach should improve the performance of scenes with varying sizes of triangles, like it is normally preferred for models using in games, since the cost of the rasterizer scales with the number of triangles to be rendered. For scenes that mostly have similar sized triangles, like the gallery scene, there should not be much difference.

The issue with the very large diagonal triangles of Bistro Interior showed in 2.4 could be fixed by splitting those long triangles into multiple leaves.

Reducing Incoherence

It is possible to reduce the incoherence of secondary rays by grouping them according to their start position and duration. It would be interesting to see how much grouping could improve the cache misses of wide traversal.

The current implementation of wide traversal does not preserve the original ray order. With the first leaf intersections, the order the rays are computed with can change. Grouping should still improve the cache hit rate, but it needs to be tested to see if the performance improvement outweighs the cost of sorting.

BVH size improvements

There are multiple ways to possibly reduce the memory requirement of the stored BVH. The two core issues that could be improved are half empty nodes and the fact that leaves mostly consist of unused memory.

The half empty nodes are mostly a design decision and are required for the SoA storage order. The fact that every node and the primitive have the same size also enables us to store only the first index of multiple elements. It would be possible to store the actual node size and then have nodes with different sizes, but that would most likely be slower than our approach and would require individual storage of the child indices.

The second aspect of wasted memory are the leaf nodes, since the space to save the AABB is unused. The above approach could improve this since it could store leaves without the bounds storage.

For some BVH configurations it would also be possible to store the triangle data inside the bounds array that normally stores the AABB. We tried it for N12L8 since 8 triangles fit perfectly into the memory needed for 12 AABB, but the performance decreased. We did not investigate this further.

Since leaves only store the index of the first triangle, a better approach would be not to leaves at all and instead save the triangle index at the parent node. This would also require individual storage of child indices. This significantly reduces the overall size of the BVH. It also reduces the number of cache lines loaded for each leaf size by one. This would not significantly change the results of the cache simulator.

5.2 Conclusion

To our knowledge, this is the first work to investigate all the different BVH. We hoped to find something like the optimal BVH configuration or some ratings that narrow down reasonable BVH configurations.

After the extensive evaluation it can be said that there is no optimal BVH configuration. It is always a trade of between different factors and ultimately depends on the given hardware. For CPUs a safe approach is to choose the BVH configuration that fits the largest supported SIMD width of the processor. Single ray traversal is most likely faster for smaller node sizes, since most CPUs have a sufficiently large Level 1 cache of 512 cache lines per physical core. With a SIMD width of 16 wide traversal will probably be faster and more consistent for complex scenes.

In general it can be said that N2 and L1 should be avoided, since increasing the leaf and node size by at least one significantly reduces the number of intersections.

For the fixed function hardware, wide traversal is a very promising approach, since it requires only a fraction of the cache compared to single ray traversal. A Level 1 cache of 16 to 32 cache lines is enough for good cache hit rates. This is only 1 to 2 KiB.

As long as the node and leaf size is not below 4, there seems to be no false choice for the SIMD width. It is most likely better to choose an even leaf size to utilize the fact that most artists create models with quads. Node and leaf sizes between 8 and 16 show the least cache misses.

Chapter 6

Appendix

Bvh Showcase of Different Node Sizes

The first figures visualizes the effect of different branching factors on the BVH. The first 4 Figures show the BVH of N2, N4, N8, N12, and N16. The model is Erato, downloaded from Morgan McGuire's Computer Graphics Archive [10]. All figures follow the same scheme. They show the amount of AABBs the ray hit during ray tracing for the specific BVH depth. The leftmost image shows the AABB of the root (depth 0). From left to right the depth increases by one. Black represents no collision. The grey scales are normalized to the maximum intersection count for each image.

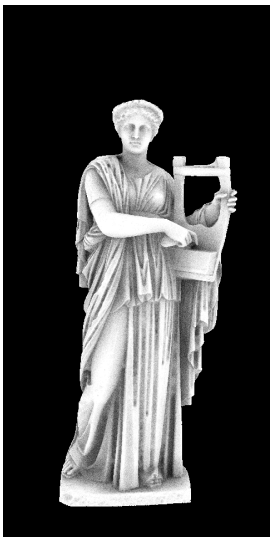


Figure 6.1: Erato render

The Figure 6.7 showcases why it is necessary to chose a new spiting axis after each split. Since every split uses the same axis BVH creation results in 16 horizontal slices for the root node. Depending on the viewing angle this can result in a drastic increase in successful node intersections.

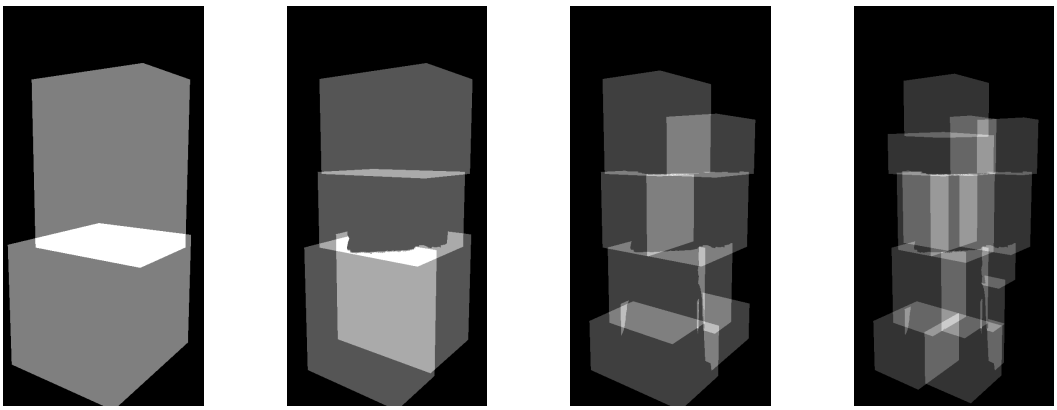


Figure 6.2: Erato BVH showcase nodesize 2

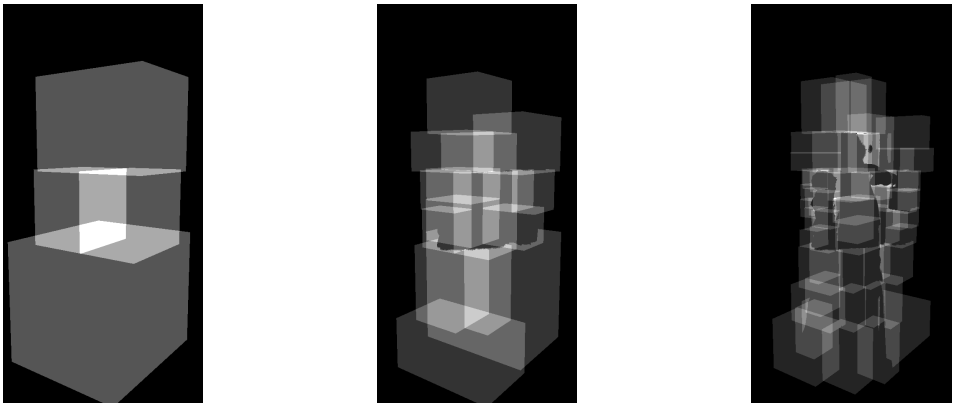


Figure 6.3: Erato BVH showcase nodesize 4

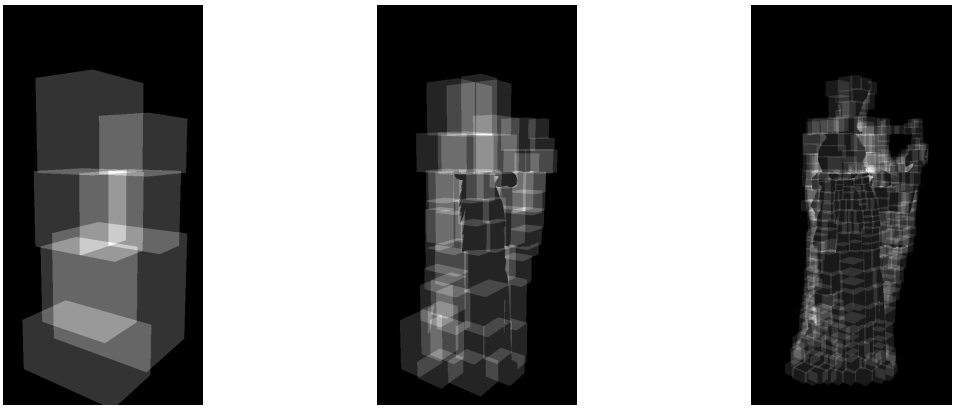


Figure 6.4: Erato BVH showcase nodesize 8

Figure 6.5: Erato BVH showcase nodesize 12

Figure 6.6: Erato BVH showcase N16

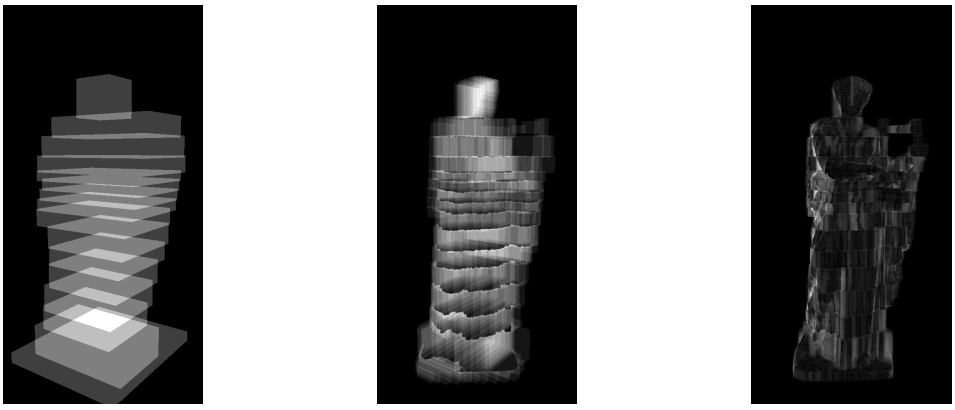


Figure 6.7: Erato BVH showcase N16. The difference to the above figures is that this BVH does not change the split axis for each split.

BVH Sizes for all Scenes

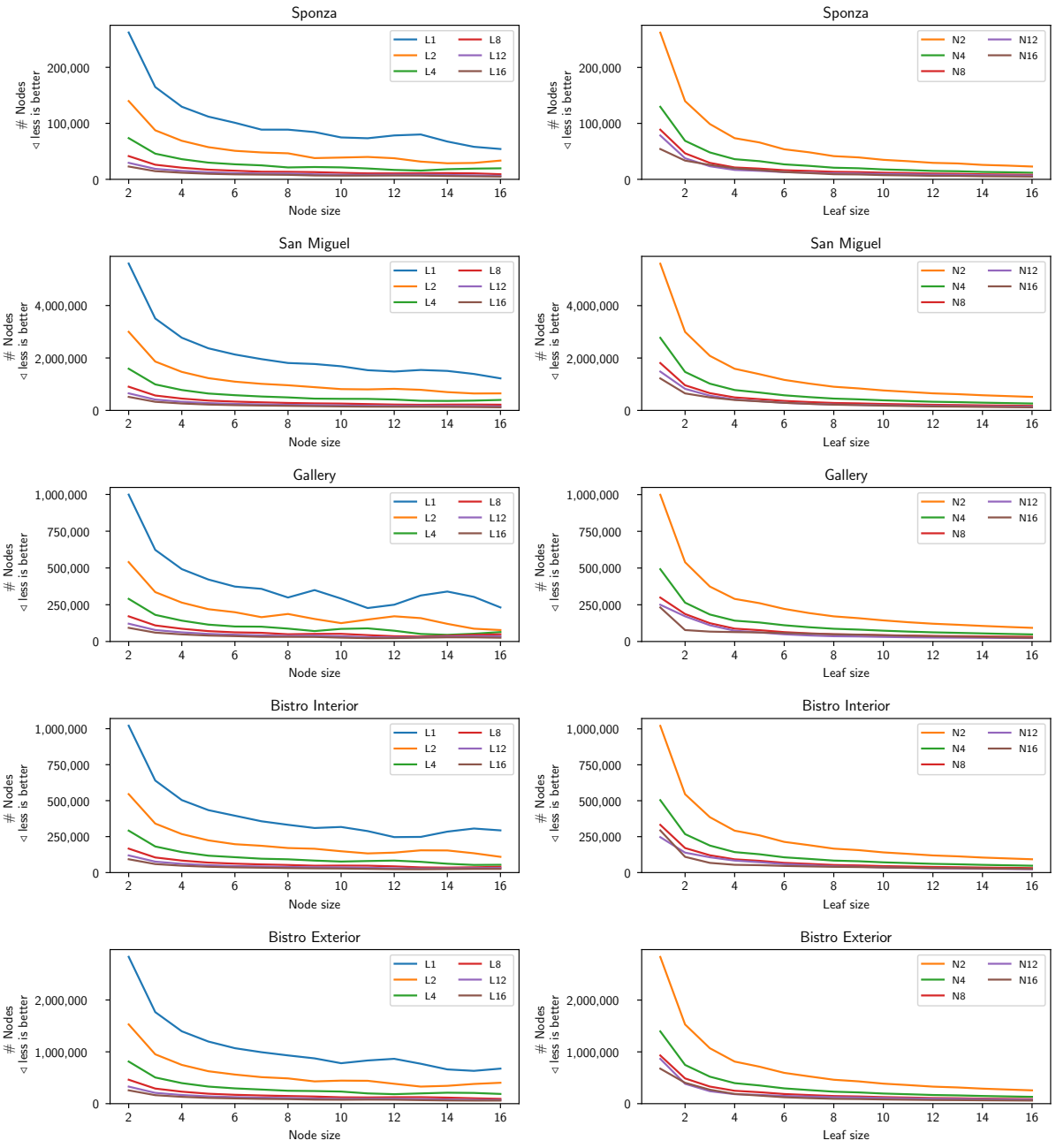


Figure 6.8: Primary rays intersection results for all scenes.

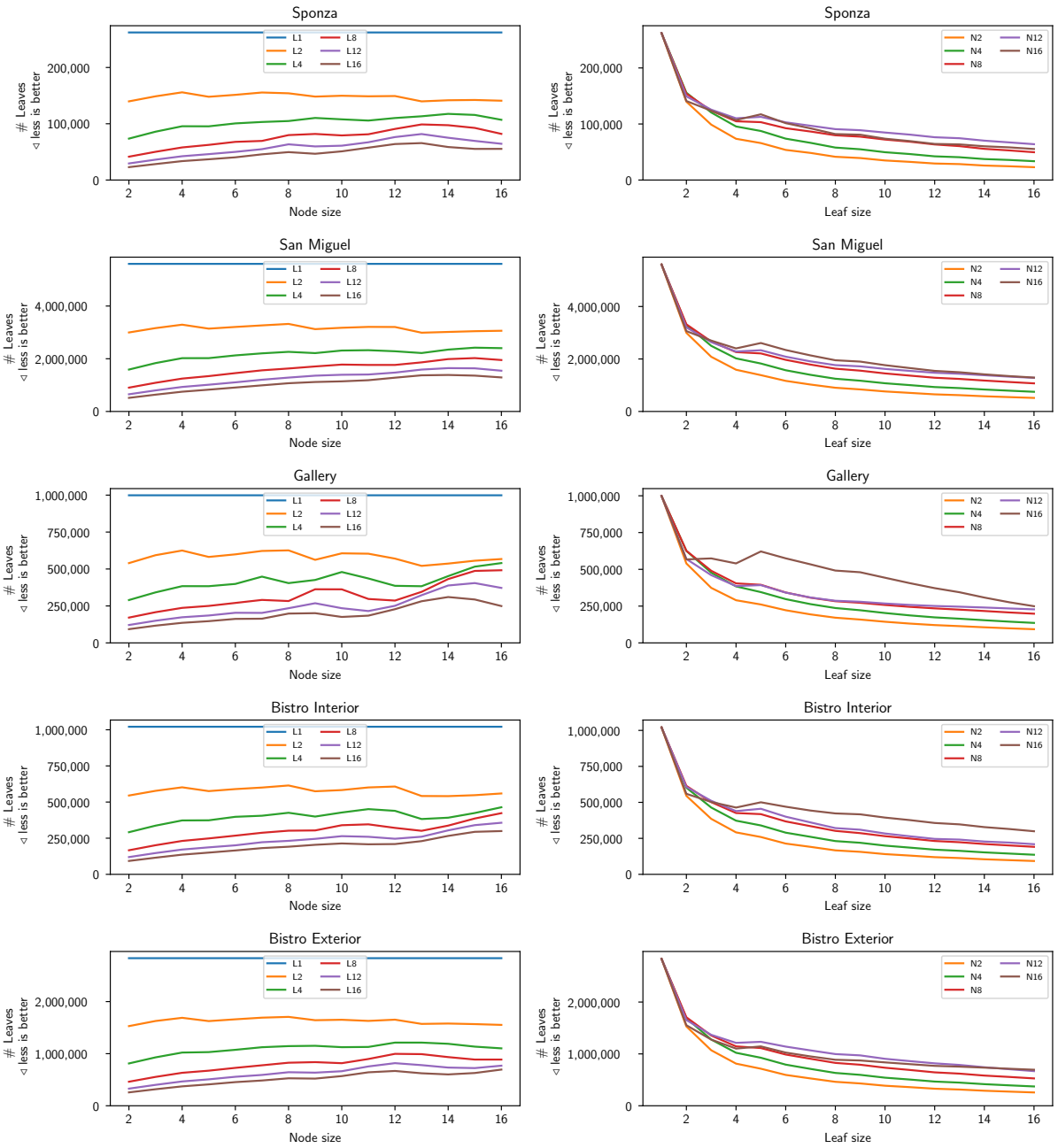


Figure 6.9: Secondary ray intersection results for all scenes.

Intersection Results for all Scenes

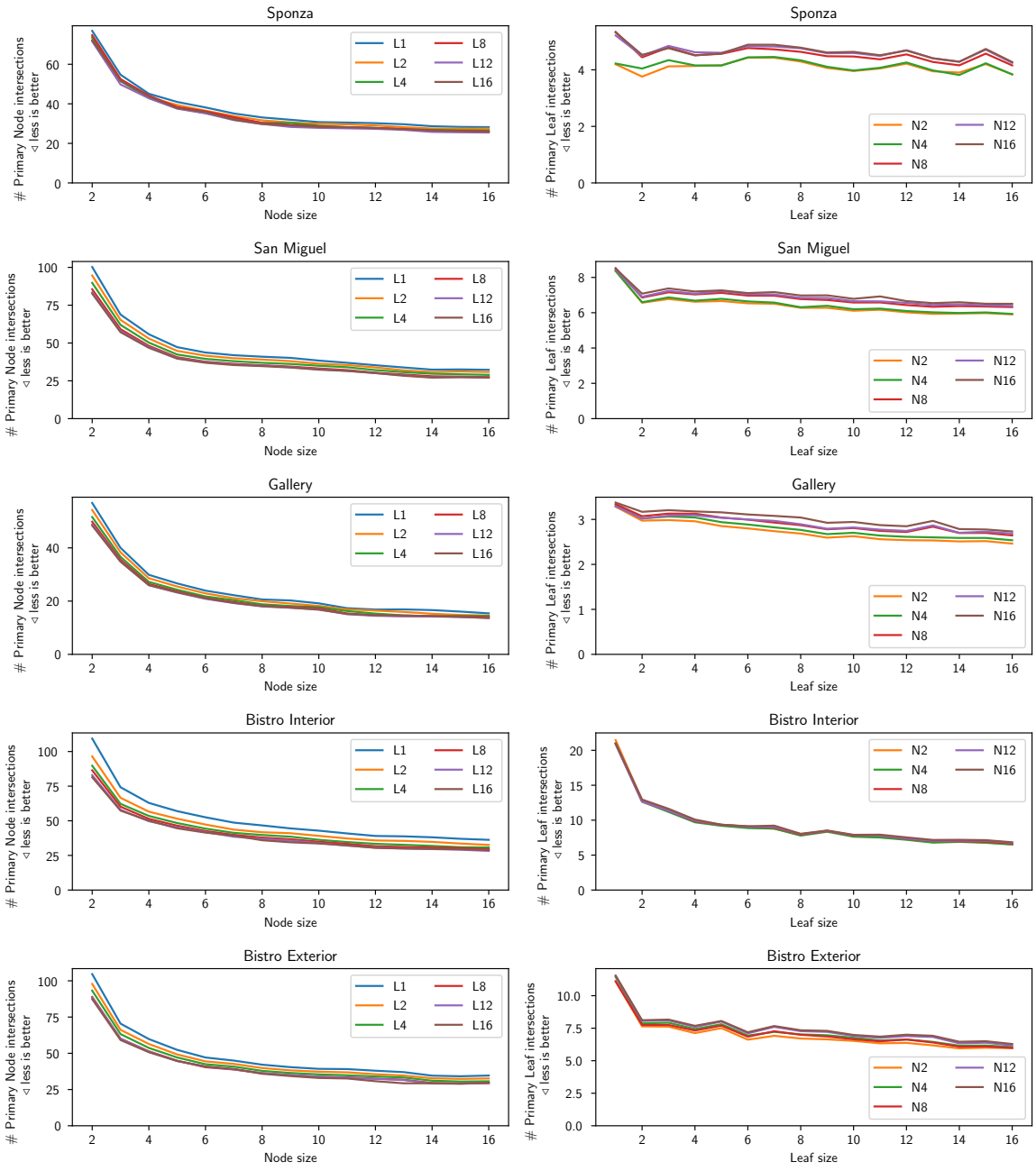


Figure 6.10: Primary rays intersection results for all scenes.

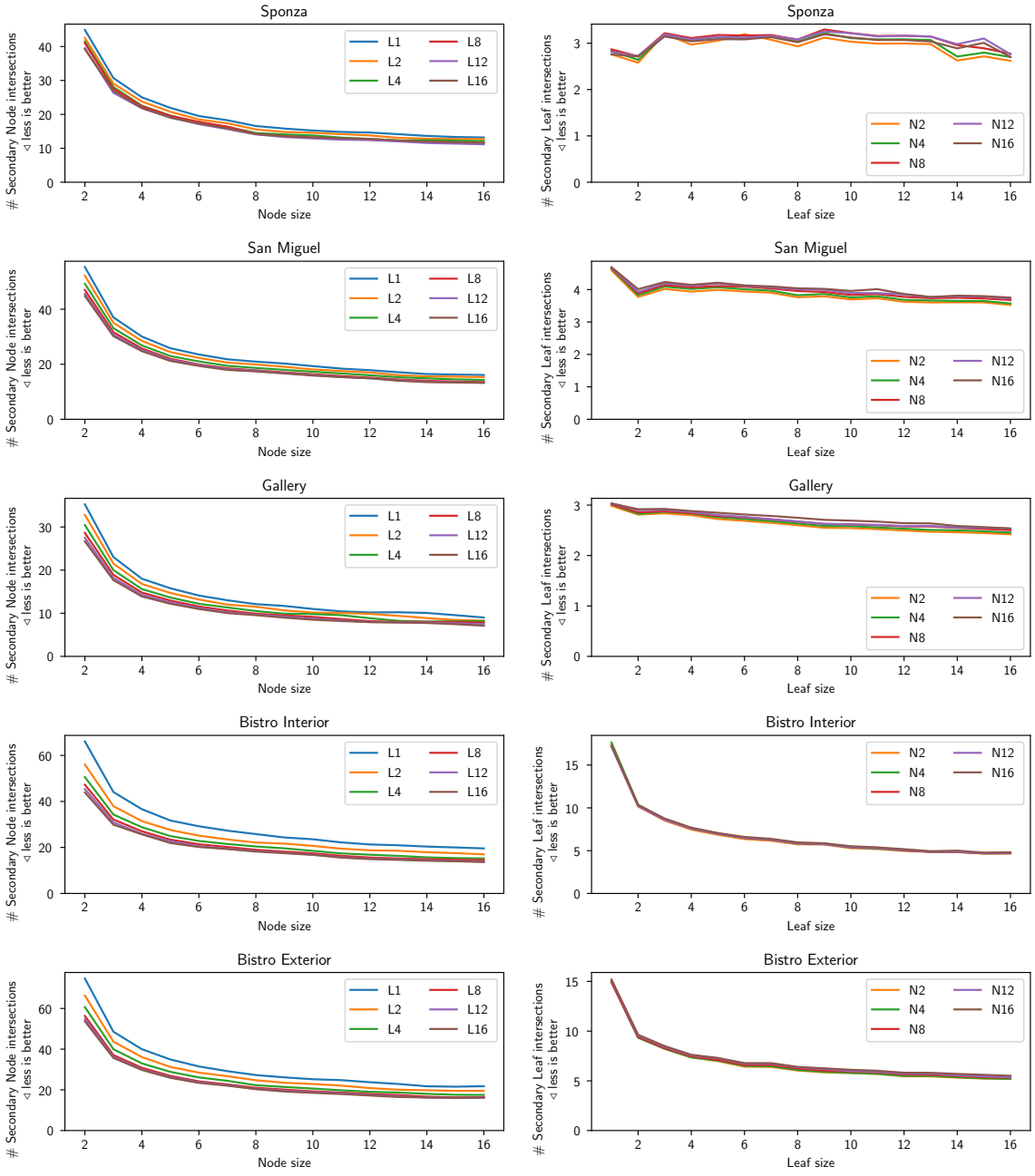


Figure 6.11: Secondary ray intersection results for all scenes.

Cache Analysis Graphs

The first group of graphs shows the results of the cache analysis for different work group sizes. Single ray traversal results are mostly independent of the work group size. Wide traversals benefits from smaller work groups for the larger cache sizes since it reduces the memory overhead. The larger work groups improve the cache behavior of the smaller cache sizes.

Figure 6.16 and 6.17 show the number of cache misses for the other work group sizes.

After that Figure 6.18 and 6.19 show the cache behavior each individual ray during single ray traversal.

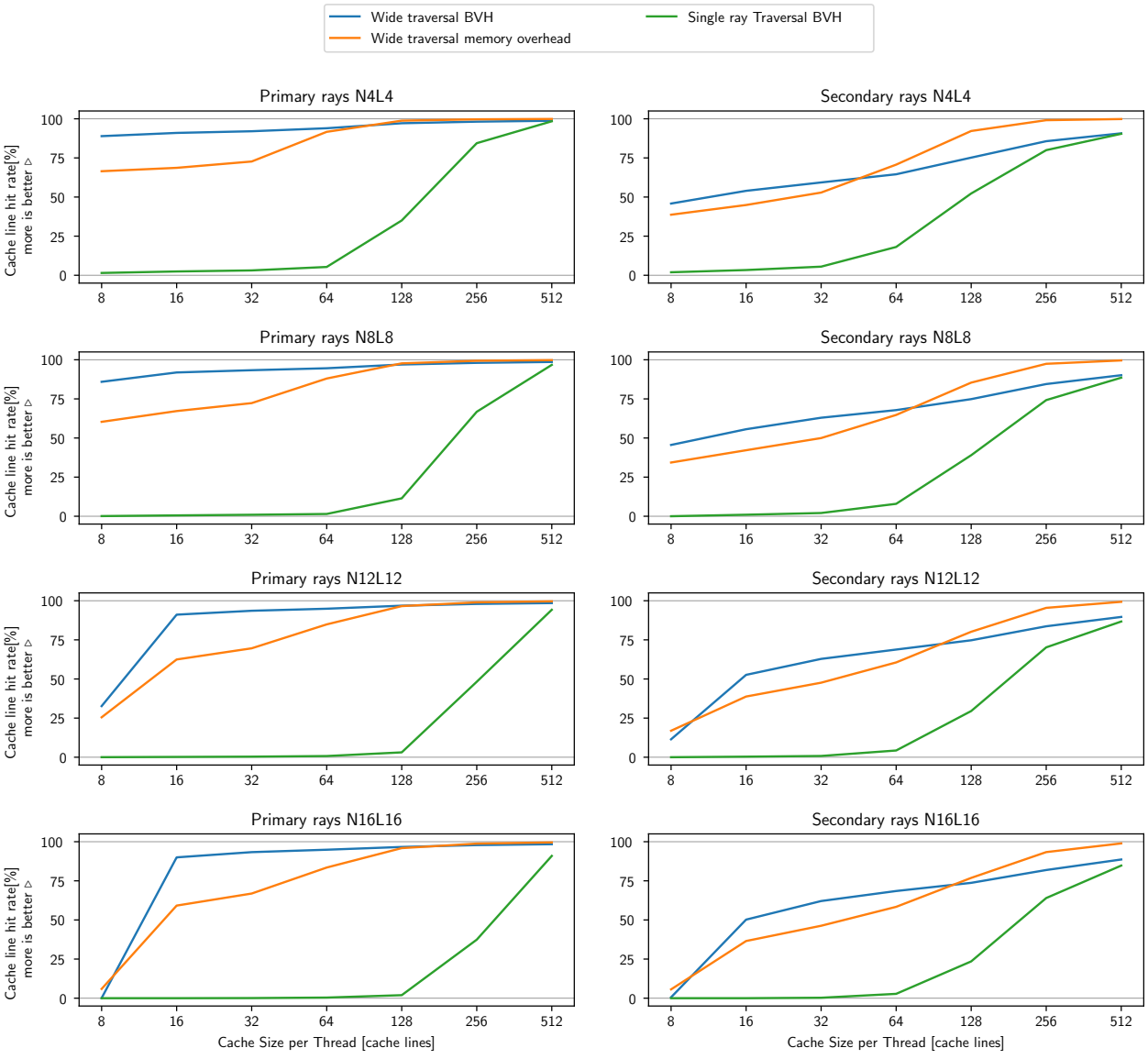


Figure 6.12: Overview of cache results for Bistro Interior. 8×8 work group

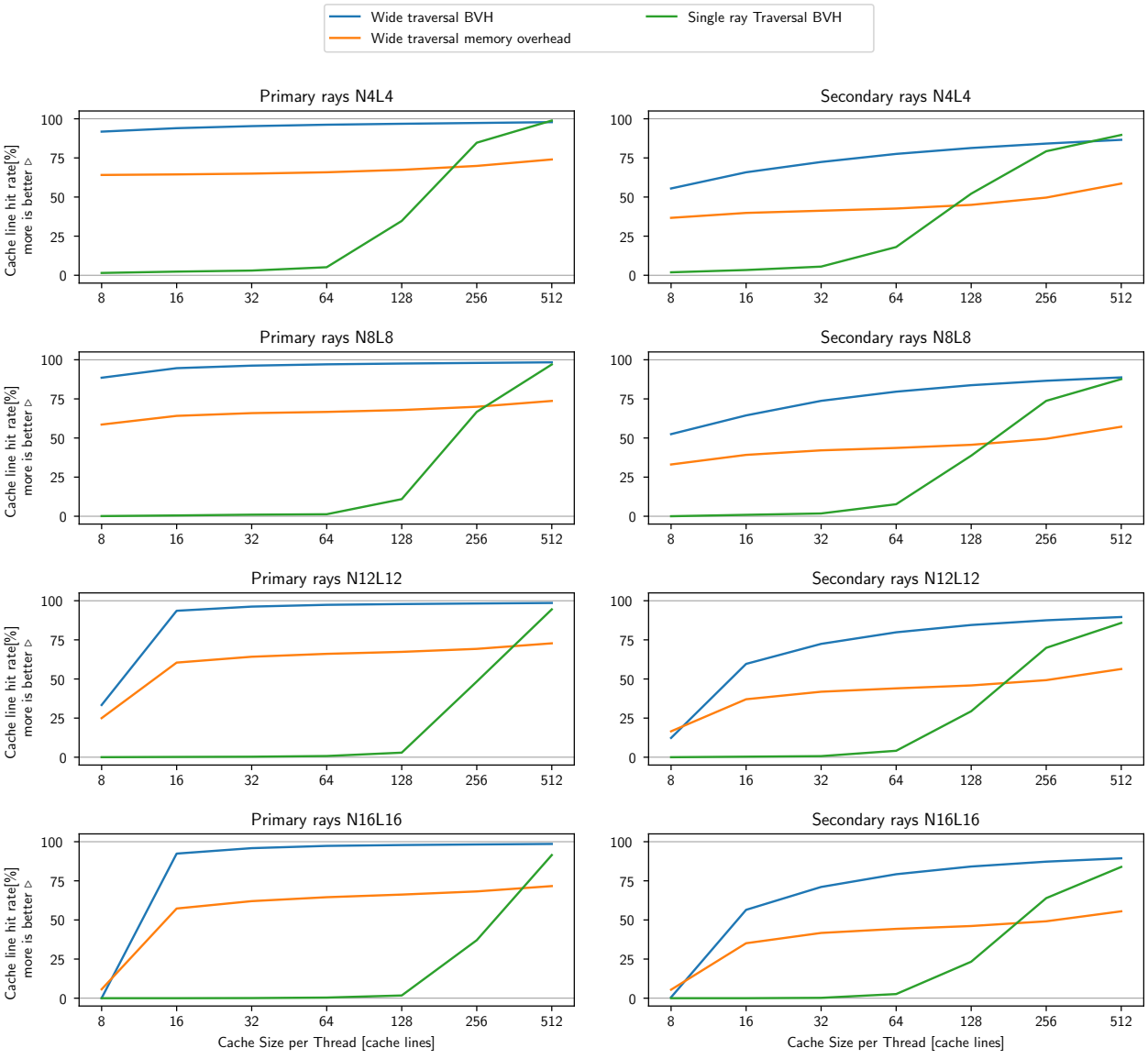


Figure 6.13: Overview of cache results for Bistro Interior. 32×32 work group

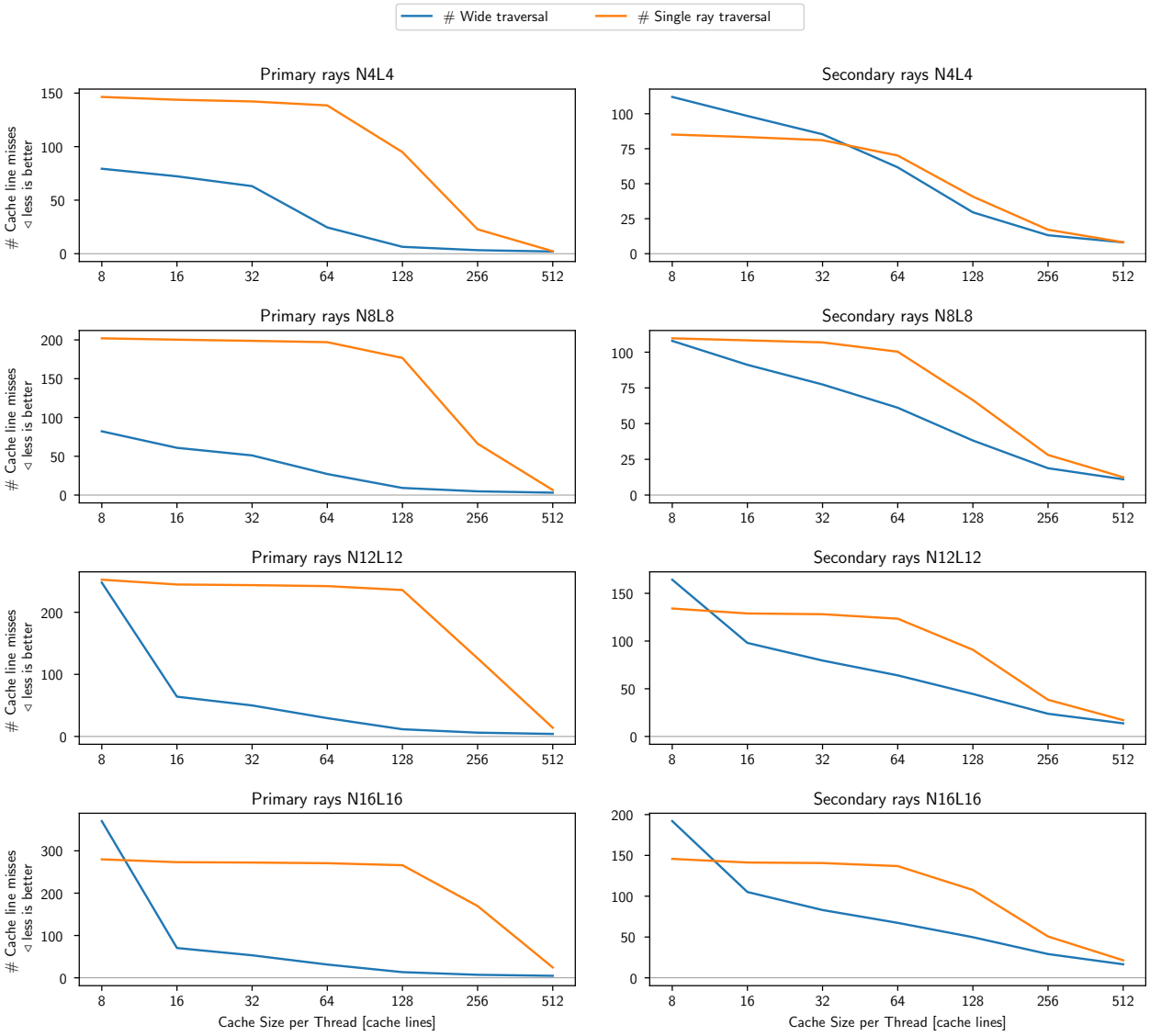


Figure 6.14: Number of Cache misses during the rendering, averaged per ray. Bistro Interior. 8×8 work group

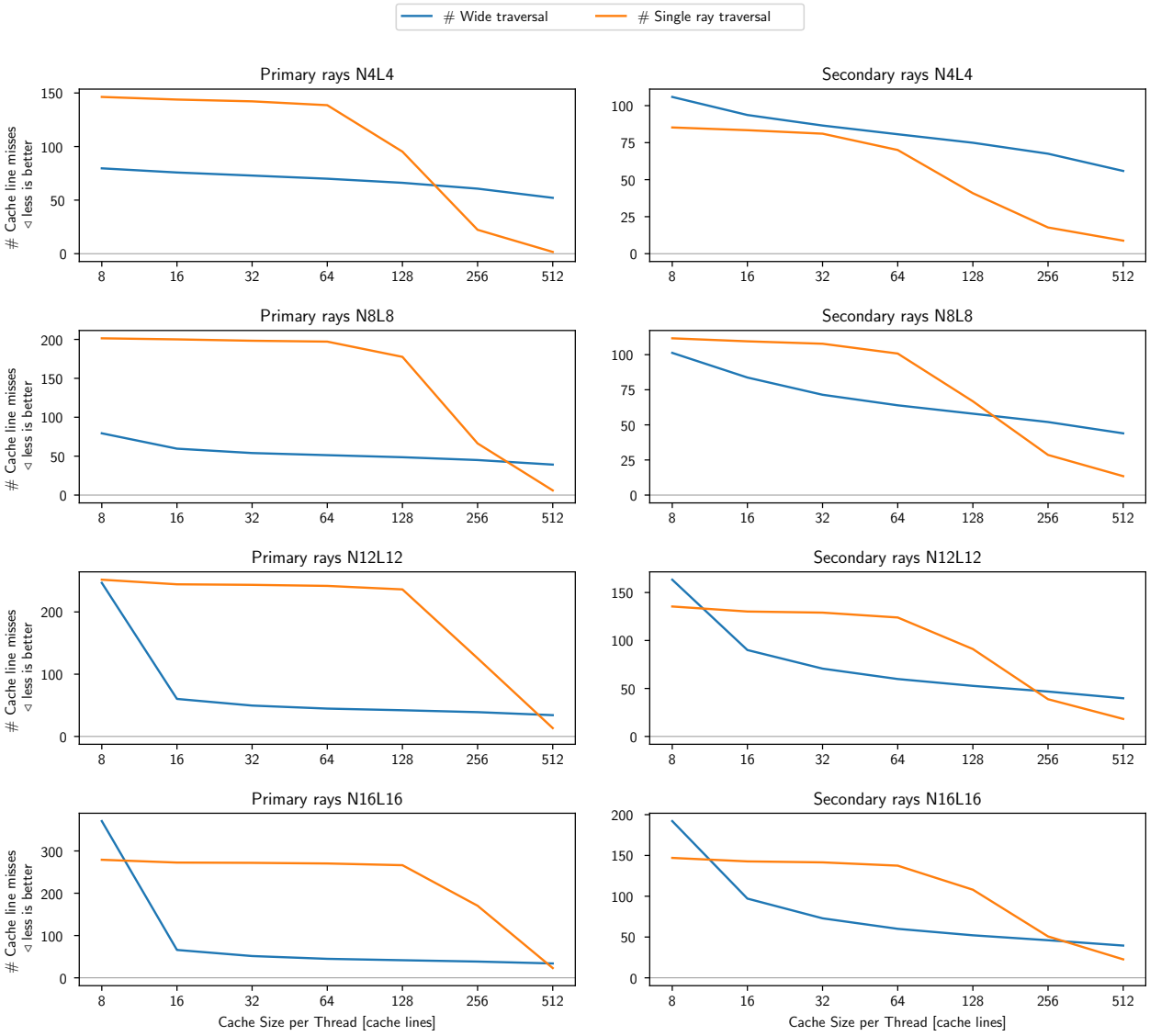


Figure 6.15: Number of Cache misses during the rendering, averaged per ray. Bistro Interior. 32×32 work group

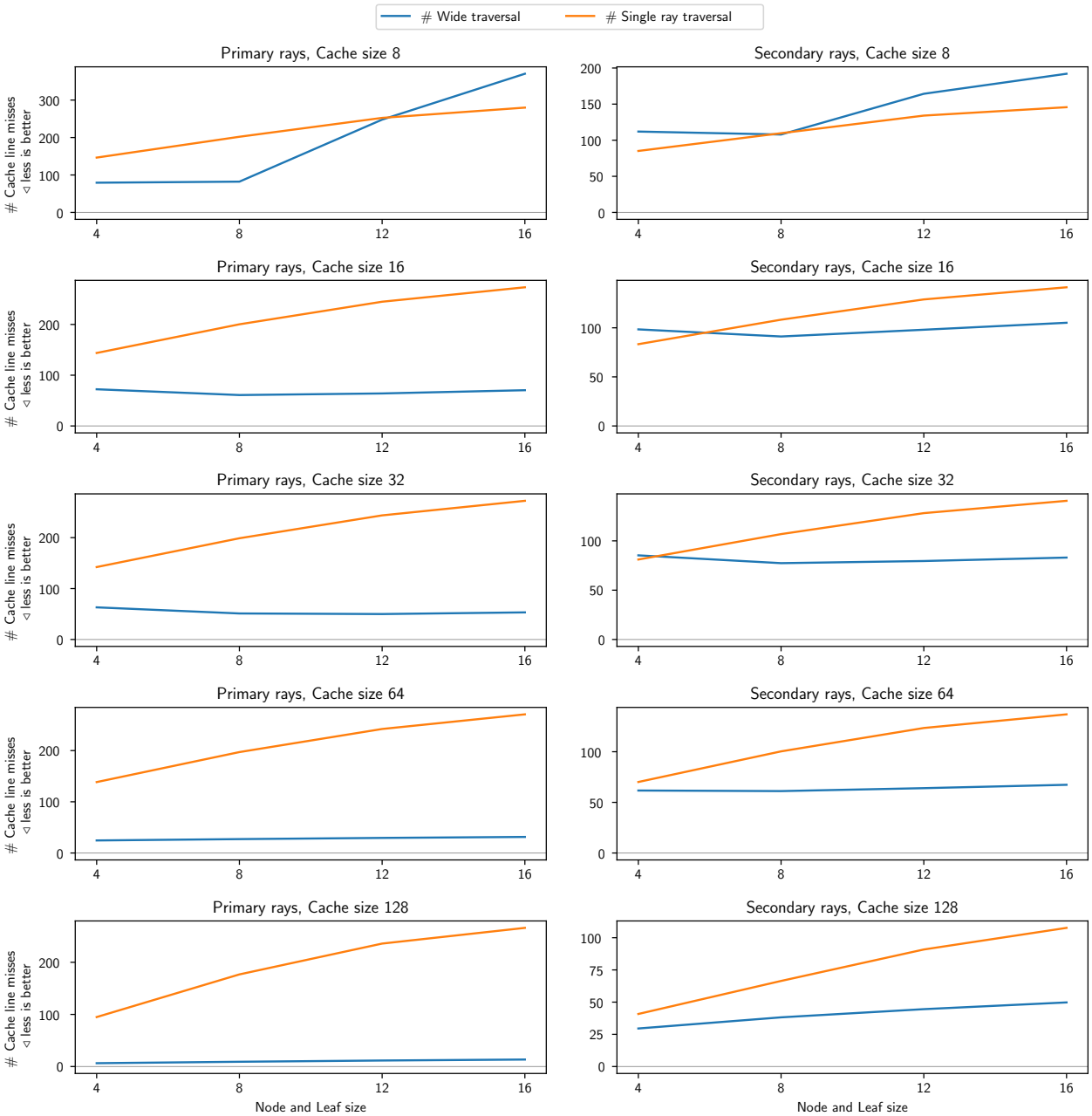


Figure 6.16: Number of Cache misses during the rendering, averaged per ray. Bistro Interior.

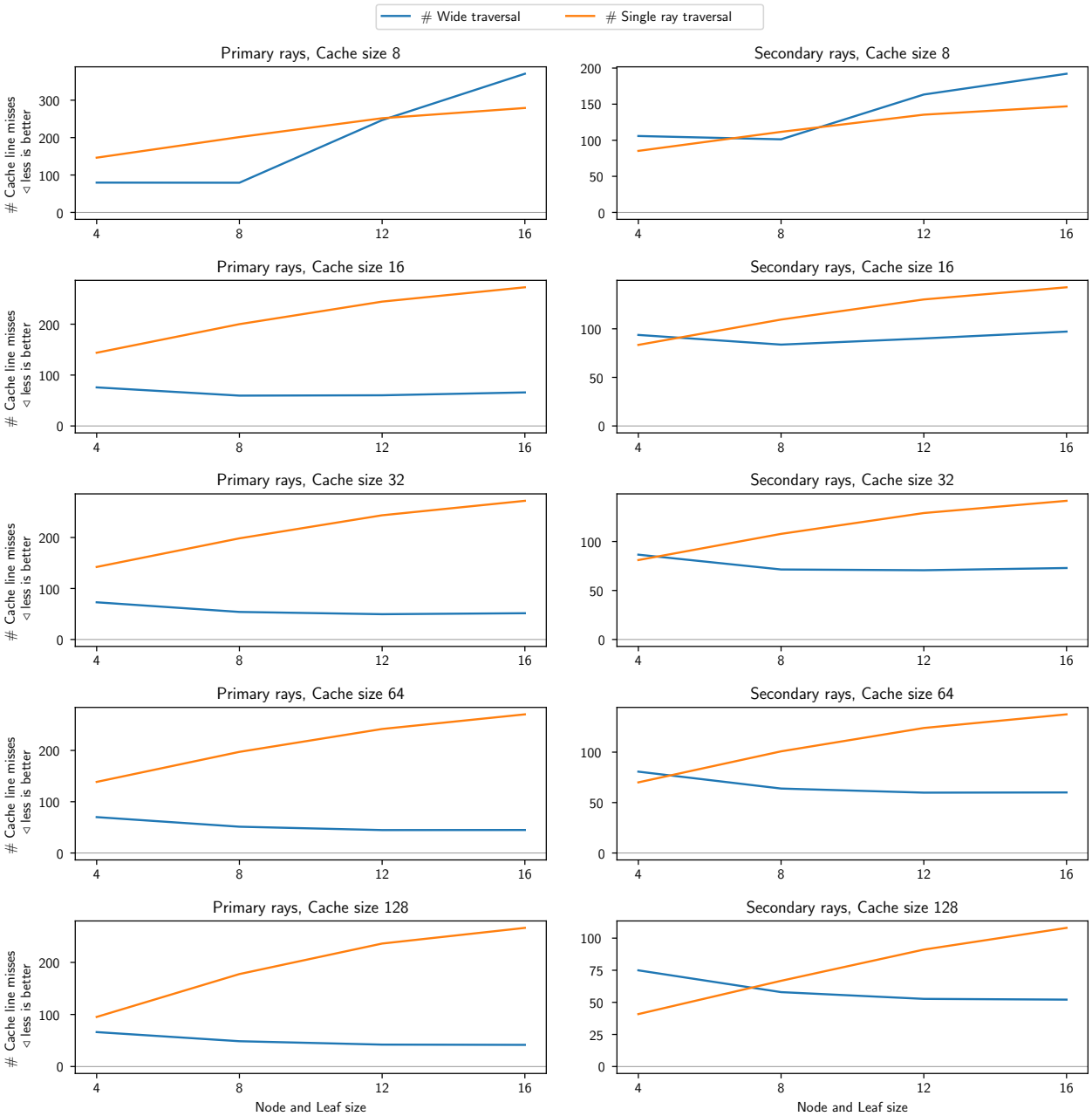


Figure 6.17: Number of Cache misses during the rendering, averaged per ray. Bistro Interior.

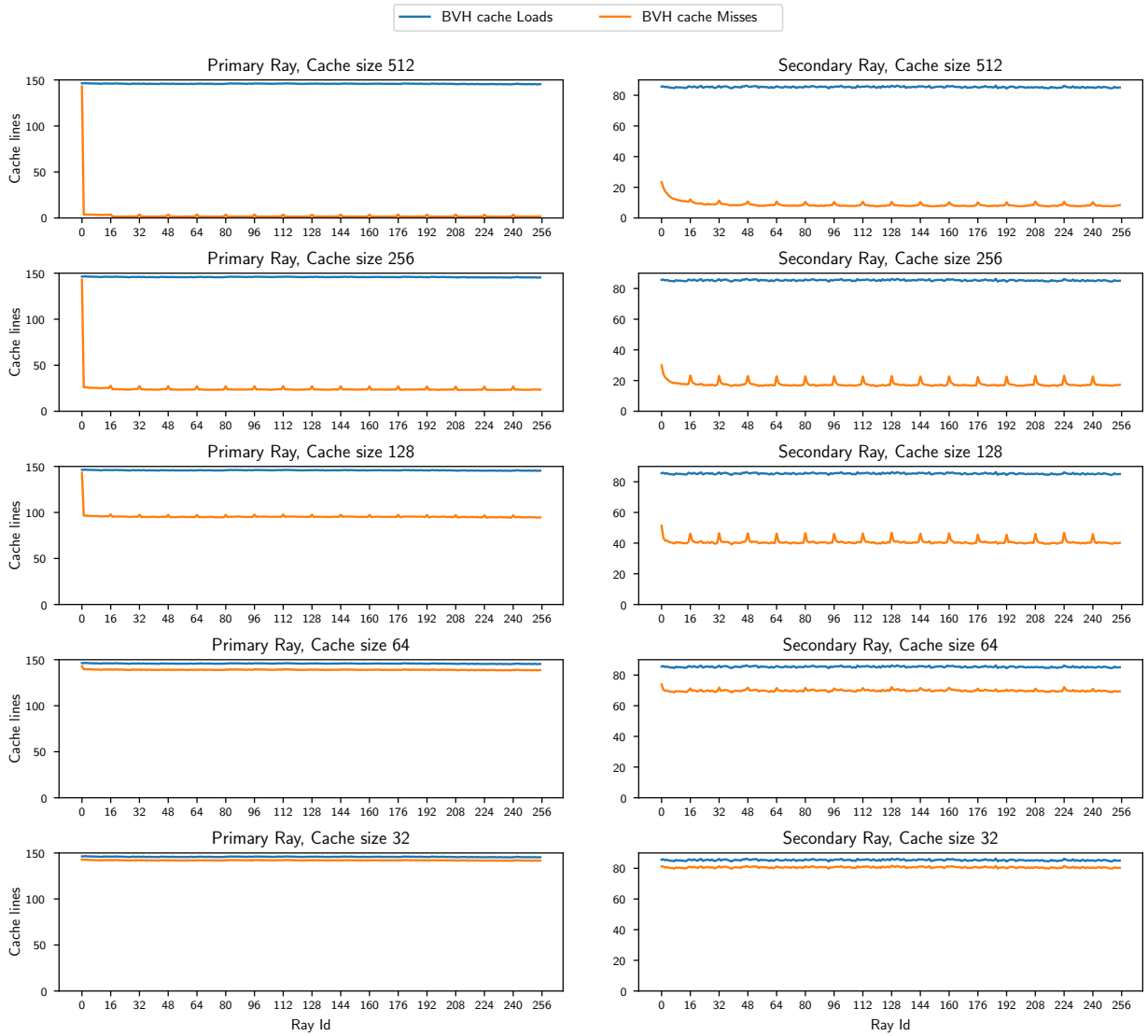


Figure 6.18: Cache loads and hits of single ray traversal for each ray of the work group. Work group size of 16×16 and a BVH of N4L4 for Bistro Interior.

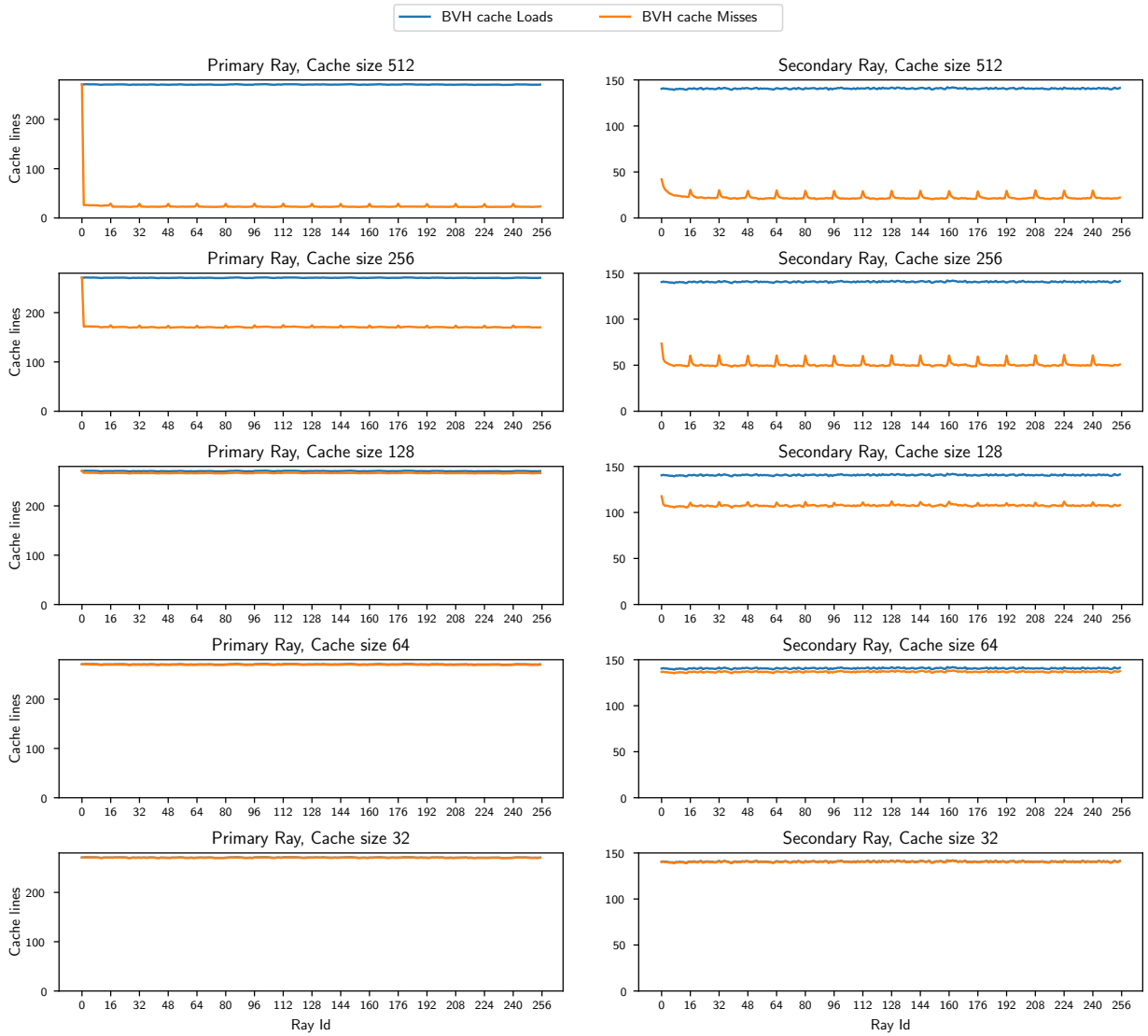


Figure 6.19: Cache loads and hits of single ray traversal for each ray of the work group. Work group size of 16×16 and a BVH of N4L4 for Bistro Interior.

Bibliography

[1] Andreas Abel. nanobench cache analyzer. https://github.com/andreas-abel/nanoBench/tree/master/tools/CacheAnalyzer.

[2] Timo Aila, Tero Karras, and Samuli Laine. On quality metrics of bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference*, pages 101–107, 2013.

[3] Intel Corporation. Intel spmd program compiler. https://ispc.github.io/.

[4] Intel Corporation. Ispc examples. https://github.com/ispc/ispc/blob/master/examples/rt/rt.cpp.

[5] Adrian Courrèges. Doom (2016) - graphics study. http://www.adriancourreges.com/blog/2016/09/09/doom-2016-graphics-study/, September 2016.

[6] G-Truc Creation. Opengl mathematics. https://github.com/andreas-abel/nanoBench/tree/master/tools/CacheAnalyzer.

[7] Gille Damien. *Study of different cache line replacement algorithms in embedded systems*. PhD thesis, Citeseer, 2007.

[8] Holger Dammertz, Johannes Hanika, and Alexander Keller. Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays. In *Computer Graphics Forum*, volume 27, pages 1225–1233. Wiley Online Library, 2008.

[9] Syoyo Fujita. Header only c++ tiny gltf library. https://github.com/syoyo/tinygltf.

[10] Morgan McGuire. Computer graphics archive. https://casual-effects.com/data, July 2017.

[11] OpenMP Architecture Review Board. OpenMP application program interface version 2.0. https://www.openmp.org/wp-content/uploads/cspec20.pdf, 2002.

[12] M. Pharr, W Jakob, and G. Humphrey. *Physically Based Rendering: From Theory To Implementation, 3rd ed.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2016.

[13] Armin Ronacher. Jinja. `https://palletsprojects.com/p/jinja/`.

[14] VV Sanzharov, AI Gorbonosov, VA Frolov, and AG Voloboy. Examination of the nvidia rtx. 2019.

[15] Martin Stich, Heiko Friedrich, and Andreas Dietrich. Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 7–13, 2009.

[16] Henri Ylitie, Tero Karras, and Samuli Laine. Efficient incoherent ray traversal on gpus through compressed wide bvhs. In *Proceedings of High Performance Graphics*, pages 1–13. 2017.

All websites have been last accessed on March 15, 2020