

Master Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Scene optimization for RTX

Bc. Tomáš Bilák

Supervisor: doc. Ing. Jiří Bittner, Ph.D.

Field of study: Open Informatics

Subfield: Computer Graphics

May 2024

I. Personal and study details

Student's name: **Bilák Tomáš** Personal ID number: **435592**
Faculty / Institute: **Faculty of Electrical Engineering**
Department / Institute: **Department of Computer Graphics and Interaction**
Study program: **Open Informatics**
Specialisation: **Computer Graphics**

II. Master's thesis details

Master's thesis title in English:

Scene optimization for RTX

Master's thesis title in Czech:

Optimalizace scény pro RTX

Guidelines:

Bibliography / sources:

[1] Meister, Daniel, and Ji í Bittner. 'Parallel locally-ordered clustering for bounding volume hierarchy construction.' IEEE transactions on visualization and computer graphics 24.3 (2017): 1345-1353.
[2] Carsten Benthin, Sven Woop, Ingo Wald, and Attila T. Áfra. 2017. Improved two-level BVHs using partial re-braiding. In Proceedings of High Performance Graphics (HPG '17). Association for Computing Machinery, New York, NY, USA, Article 7, 1–8.
[3] Wald, I., Morrical, N., Zellmann, S., Ma, L., Usher, W., Huang, T., & Pascucci, V. (2020). Using Hardware Ray Transforms to Accelerate Ray/Primitive Intersections for Long, Thin Primitive Types. Proceedings of the ACM on Computer Graphics and Interactive Techniques, 3(2), 1-16.

Name and workplace of master's thesis supervisor:

doc. Ing. Ji í Bittner, Ph.D. Department of Computer Graphics and Interaction

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **16.02.2023** Deadline for master's thesis submission: **24.05.2024**

Assignment valid until: **22.09.2024**

doc. Ing. Ji í Bittner, Ph.D.
Supervisor's signature

Head of department's signature

prof. Mgr. Petr Páta, Ph.D.
Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt

Student's signature

Acknowledgements

I want to express my special thanks to my supervisor doc. Ing. Jiří Bittner, Ph.D. for his guidance and patience throughout the entire course of this project.

I sincerely thank everyone who supported me during my everlasting studies. My loving parents in the first place, who are always supportive. My dear partner Babu, formally second, but certainly not in second place, for her love, care, and understanding during the completion of this work.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

In Prague, 24. May 2024

Abstract

Ray tracing complex scenes in real-time is possible today and with technological advancement even fairly performant. Hardware ray tracing in consumer graphic cards is one of the newest additions. This thesis researches RTX architecture in particular with aim to optimize scene for OptiX acceleration structure builder. Our method is focusing on reducing overlaps between objects by splitting or merging them based on certain rules and constraints.

Keywords: ray tracing, RTX, OptiX, scene optimization, instancing, acceleration structures

Supervisor: doc. Ing. Jiří Bittner, Ph.D.

Abstrakt

Sledovanie lúčov v komplexných scénach v reálnom čase je dnes možné a s technologickým pokrokom dokonca pomerne výkonné. Hardvérové sledovanie lúčov v spotrebiteľských grafických kartách je jedným z najnovších prírastkov. Táto práca sa zaoberá najmä architektúrou RTX s cieľom optimalizovať scénu pre tvorbu akceleračnej štruktúry knižnicou OptiX. Naša metóda sa zameriava na zníženie prekrývania medzi objektmi ich rozdelením alebo zlúčením na základe určitých pravidiel a obmedzení.

Kľúčové slová: sledovanie lúčov, RTX, OptiX, optimalizácia scény, inšancovanie, akceleračné štruktúry

Preklad názvu: Optimalizácia scény pre RTX

Contents

1 Introduction	1	5 Results	37
1.1 Goals	2	5.1 Configurations tests	38
2 Analysis	5	Experiment 1: Parameters comparison for BVH method	39
2.1 Ray tracing	5	Experiment 2: Parameters comparison for Overlap method	39
2.2 Acceleration data structures	7	Experiment 3: Performance comparison of rendering animated geometry	41
2.3 Cost function	10	5.2 Scene optimization performance	43
2.4 Instancing geometry	11	Experiment 4: Performance comparison of implemented methods on chosen scene setups	43
2.5 RTX architecture	12	6 Conclusion	47
2.6 OptiX	13	A Bibliography	49
2.7 OWL - OptiX 7 wrapper library	15		
3 Related work	19		
3.1 Scene layout optimization	19		
3.2 Using instancing as hardware OBB test	20		
3.3 Partial rebraiding of two-level BVH	22		
4 Implementation	25		
4.1 Scene optimization	25		
4.2 Visual tools	27		
4.3 GUI	30		
4.4 Automation scripts	32		
4.5 Third-party libraries	35		

Figures

1.1 Dürer's door	1	5.4 SoM thresholds tracing performance	41
2.1 Ray tracing diagram	6	5.5 10^5 instances traversal performance	43
2.2 Hard vs. soft shadows	7	5.6 Instanced scene comparison	43
2.3 Example of a BVH	9	5.7 Traversal comparison of all methods on Fireplace scene	44
2.4 Scene organization	11	5.8 Traversal comparison of all methods on Conference scene	44
2.5 Overlapping of BLASes	11	5.9 Traversal comparison of all methods on Sibenik scene	45
2.6 Pascal vs. Turing BVH search . .	12	5.10 Traversal comparison of all methods on Sponza scene	45
2.7 Graph of OptiX 7 programs relationship	13	5.11 Traversal comparison of all methods on Powerplant scene	46
2.8 Shallow IAS vs. multi-level IAS	14	5.12 Traversal comparison of all methods on San Miguel scene	46
3.1 Single BLAS	20		
3.2 BLAS per object	21		
3.3 OBB test with masquerading technique	21		
3.4 Illustration of re-braiding method	22		
4.1 Transparent model view	28		
4.2 Heatmap model view	29		
4.3 AABB model view	30		
4.4 GUI tabs	31		
5.1 Tracing performance of BVH's .	39		
5.2 Scene fragmentation comparison	40		
5.3 Skip threshold comparison on all scenes	40		

Tables

2.1 Build flags properties	15
4.1 Scene file parameters	33
4.2 Environment file parameters . . .	34
5.1 Geometric complexity of the tested scenes	37
5.2 Build performance of BVH's . . .	39
5.3 Skip thresholds build statistics .	42

Chapter 1

Introduction

Creating a visualization of the scene using ray tracing is known from the 16th century. At the time, ray tracing was mechanical with the tool called Dürer's door. Ray was cast with a thread hanging on the hook through the wooden frame of Dürer's door to a point in the scene. This technique brought dotted outlines of an object in the manner of hours [Hof90].

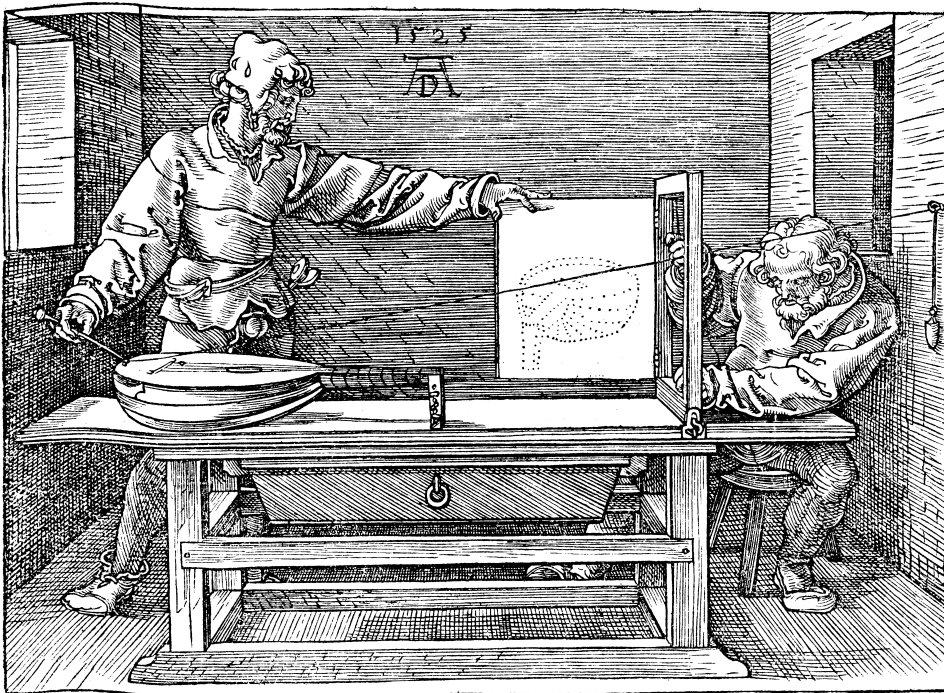


Figure 1.1: Illustration of Dürer's door by Dürer himself [Dür25].

Later, at the turn of the sixties and seventies of the 20th century, computer scientists created the first computer programs to render the surface of simple objects of the scene using ray tracing [App68]. In 1979 Turner Whitted formulated the first recursive ray tracer, which bounced rays between objects of the scene and made visualization of reflection and refraction possible. Working with the hardware limitations of the time simple scene with one glass sphere and one metal sphere on a checkered plane took 74 minutes to render. Dimensions of the rendered image were 480 by 640 pixels with Warnock-type sampling. Further on, according to Whitted [Whi80], time spent by the program was divided into three categories: 13% Overhead, 75% Intersection, and 12% Shading. For complex scenes, the report states that 95% of the time takes the intersection part.

Other problems surfaced: fast intersection tests and the organization of the scene. Various intersection problems are described by Hanrahan [Han89]. Intersection tests compute whether the ray hits a primitive in the scene. Ray-triangle is probably the most common intersection test pair. Fast computation and low memory requirement of this problem were proposed by Möller and Trumbore [MT97], which might be especially useful in today's hardware-implemented intersection tests. For the second problem mentioned, many different acceleration structures exist for various purposes. Even after a few decades of research, it is a very active topic in computer graphics and is the primary focus of this thesis.

1.1 Goals

The primary objectives of this master thesis revolve around two pivotal ideas. The first idea is based on prior experimental testing, which showed that the quality of RTX Application Programming Interface's (API) acceleration structure depends on scene organization. The basis of this critical idea is described in Section 3.1. The second idea is to exploit instancing transformations to boost hardware ray-bounding box intersection tests. More details are described in Section 3.2.

The common objective of both ideas is ultimately to reduce the time of finding the closest intersection between the ray and the triangle. While the first idea aims to do that by preparing the scene before it is handled by RTX API to increase the quality of the resulting acceleration structure, the second idea intends to reduce search space by using appropriate transformation to minimize the area of the object's bounding volume.

Before the mentioned ideas are further explored, it is essential to set out all goals. We have identified these critical goals:

- Research related work and RTX architecture
- Develop prototype application
- Explore and test analyzed ideas
- Compare the results of the original scene with the optimized one

Chapter 2

Analysis

This chapter describes the fundamentals of the explored domain. In the opening section, the basics of ray tracing are explained. Later defines acceleration data structures and explains cost function in the two following sections. The further section illustrates the concept of instancing using two-level acceleration structures. The last three sections are dedicated to RTX from the bottom to the most abstract level. The first of three sections zooms in on the hardware architecture. OptiX ray tracing API is outlined in the second section, and the final section of this chapter is about OWL, the OptiX 7 wrapper library.

2.1 Ray tracing

Ray is a half line; it is described by starting point R_O , also called origin, and its direction vector \vec{R}_D . Direction is usually normalized ($|\vec{R}_D| = 1$). The parametric equation can express any point on the ray:

$$P = R_O + t * \vec{R}_D, t \geq 0$$

The ray tracing fundamentally shoots rays from the camera through the projection plane into the scene and looks for the closest hit. The naive approach to finding the closest hit tests each primitive for an intersection. That induces testing for an intersection between each primitive and each

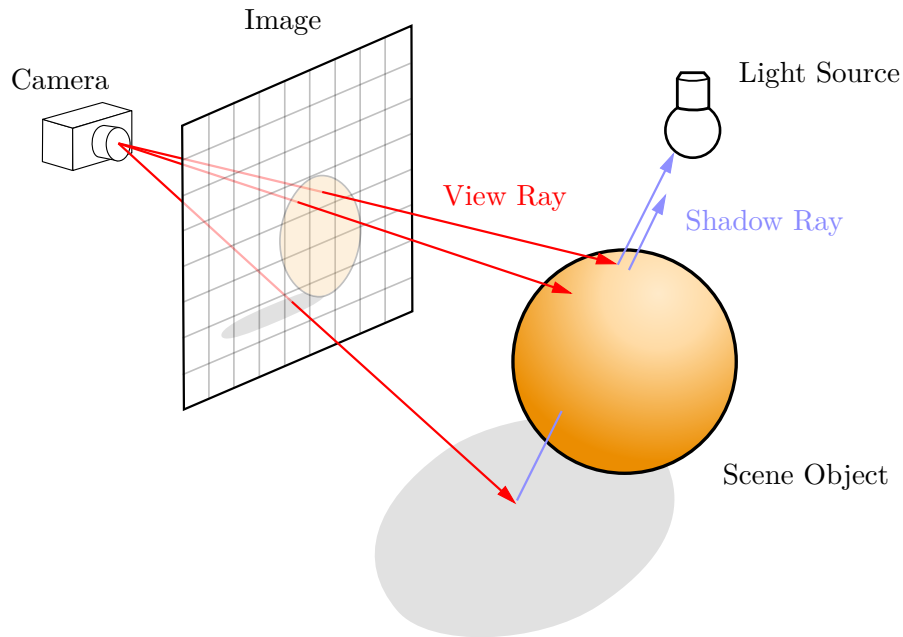


Figure 2.1: Ray tracing diagram. Image courtesy of Henrik [Hen08].

ray. For a full image of w by h pixels, at least $N_R = w * h$ rays have to be shot. The total complexity of this method for a scene with N_O objects is $\mathcal{O}(N_R * N_O)$, and the intersection test is computed for every ray-primitive pair. If we consider that we only solved hits for primary rays and usually secondary rays are needed at the minimum, depending on the shading model, the naive algorithm is obviously infeasible for any performance-oriented use.

Secondary rays have a crucial role in simulating complex lighting interactions and optical effects to achieve realistic image synthesis. They are used mainly for simulating the shading of reflective and refractive materials and global illumination.

Shadow rays compute the contribution of light sources from the scene, and they can be terminated early if anything is hit before reaching the light source. Sampling each light source with multiple shadow rays is a technique commonly applied with area lights. Each sample directs shadow ray at a different point of the area light's surface. If all shadow rays reach a light source without being interrupted by other objects, the surface is fully lit. Partial occlusion of shadow rays creates soft shadows and the region where all surface points have only a portion of the light source visible is called the penumbra. Furthermore, if all shadow rays intersect scene objects before hitting the light source, that light is totally excluded, and the region of such surface points is called the umbra. The difference between hard shadows and soft shadows is depicted in Figure 2.2.

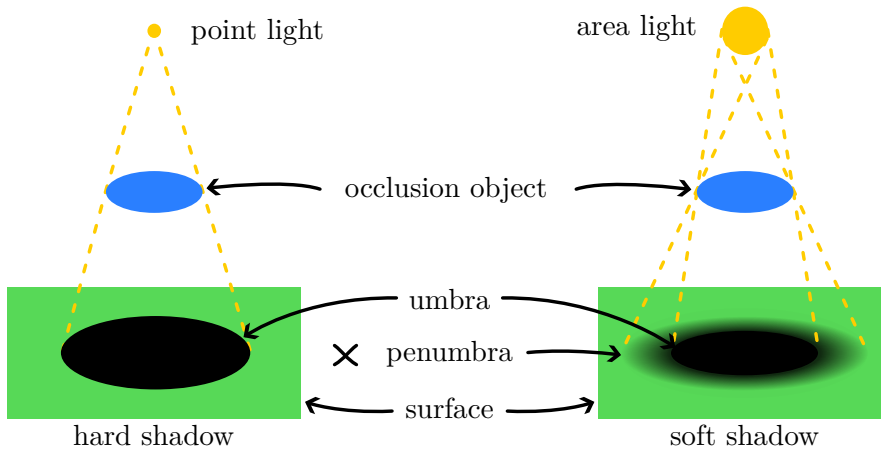


Figure 2.2: Hard shadows with umbra only from point light on the left, soft shadows from area light on the right.

2.2 Acceleration data structures

Grouping the scene’s geometry into subsets helps prune the search space when traversing the scene with a ray. For fast trace, a structure with good quality is needed, but dynamic scenes need refitting or rebuilding of the structure every few frames. Arvo and Kirk [AK89] name a multitude of ray tracing acceleration techniques in their survey, but in relation to acceleration structures, they organized solutions into two categories: *Bounding volumes and hierarchies* and *3D spatial subdivisions*. The second-mentioned can be further broken down into *Uniform spatial subdivision* and *Nonuniform spatial subdivision*. Meister et al. [Mei+21] state that bounding volume hierarchy (BVH) has been the dominant acceleration data structure used for ray tracing in the last decade. As the reasons for BVH’s popularity, the authors identify four key attributes:

- Predictable memory footprint
- Robust and efficient query
- Scalable construction
- Dynamic geometry

Memory complexity of BVH can be calculated in advance, for example, binary BVH would consist of $2n - 1$ nodes maximally, where n is the number

of scene primitives. Finding intersections without acceleration structure means linearly testing every primitive in the set, while time complexity using acceleration structures decreases to logarithmic on average.

BVH construction can be classified based on its approach to organizing and partitioning the spatial hierarchy. Broadly speaking, there are three main categories:

- top-down construction
- bottom-up construction
- incremental construction

Top-down construction initially holds all primitives at the root node and recursively splits every node into two subnodes until one of the termination criteria is met. It may be desirable to terminate the algorithm based on a number of primitives in the node, maximum tree depth, maximum memory quota, or a combination of these criteria. An example of such a top-down algorithm is shown with pseudocode in Listing 2.1.

```
1      root contains all scene primitives
2      push root onto the stack
3      while stack is not empty do
4          pop node from the top of the stack
5          if termination criteria for node are met then
6              make node leaf
7          else
8              split primitives in node into children
9              assign children as child nodes of node
10             foreach child in node do
11                 push child onto the stack
```

Listing 2.1: The basic top-down construction algorithm. Algorithm courtesy of Meister et al. [Mei+21]

Bottom-up construction, on the other hand, starts with all primitives as leaves and incrementally groups pairs (or more) of nodes together until one root node is formed. Both methods require all primitives before construction. Incremental construction doesn't have this requirement, which is beneficial if primitives are streamed on demand, but the quality of such trees is expected to be suboptimal.

```

1   push root onto the stack
2   while stack is not empty do
3       pop node from the top of the stack
4       if ray intersects node then
5           if node is not leaf then
6               foreach child in node do
7                   push child onto the stack
8           else
9               foreach prim in node do
10                  test whether ray intersects prim

```

Listing 2.2: The basic stack-based traversal algorithm. Algorithm courtesy of Meister et al. [Mei+21]

Traversal of BVH usually starts at the root of the hierarchy and progresses downward through nodes. Ray is tested for intersection with the node's bounding volumes and continues inside if they intersect while skipping the entire node otherwise. When the ray reaches a leaf node and intersects primitive, the distance from the origin is noted. Any following intersection tests may be terminated prematurely if the distance to the node's bounding volume is greater than the nearest hit noted so far. The traversal ends when there are no nodes left to test, either because nodes were pruned or already tested. In some cases, early termination may occur after any hit is found, for example, when light source occlusion is tested with shadow rays. An example of a stack-based traversal algorithm is demonstrated in Listing 2.2.

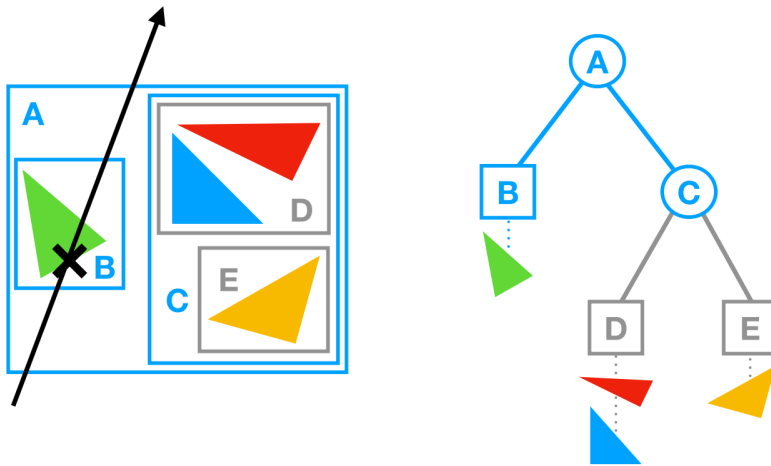


Figure 2.3: An example of a BVH built over four primitives. BVH nodes contain bounding volumes (axis-aligned bounding boxes), and primitives are referenced in leaves. While traversing the BVH to find the ray/scene intersection, we cull the entire subtree of node C since the ray does not intersect the associated bounding box. Image courtesy of Meister et al. [Mei+21].

Spatial subdivision methods are also used as acceleration data structures, namely uniform grids, octrees, and kd-trees. Kd-trees are known for forming good-quality trees, but their build times are long and cannot be refitted if the geometry is dynamic.

2.3 Cost function

The cost function is a metric used to evaluate the effectiveness of a particular split or partitioning strategy during the construction process. The goal is to find an optimal spatial hierarchy that minimizes the expected cost of traversing the hierarchy and intersecting rays with the contained geometry. The most widely used cost function in BVH construction is the Surface Area Heuristic (SAH). Goldsmith and Salmon [GS87] formulated this heuristic, where the key idea is in evaluating the cost of the hierarchy as a ratio between the sum of surface areas of inner nodes and leaf nodes to the surface area of the root node. The equation for hierarchy cost, sometimes referred to as quality, is:

$$C = \frac{1}{S_{root}} * ((\sum_{l=1}^{leaves} S_l * C_i * N_l) + (\sum_{n=1}^{nodes} S_n * C_t))$$

where S_{root} is the surface area of the root node, $leaves$ is the number of leaves in the BVH, N_l is the number of primitives in the leaf, $nodes$ is the number of inner nodes in the BVH, C_i is constant for the cost of intersection test with primitive, and C_t is constant for the cost of intersection test with bounding volume.

The SAH is typically used in a recursive fashion during a top-down BVH construction process. The algorithm evaluates different split candidates and selects the split that minimizes the overall SAH cost. This process is applied recursively until termination criteria are met, and the hierarchy is constructed. Finding the optimal BVH is, according to Karras and Aila [KA13], an NP-hard problem.

Other cost functions were proposed to overcome the shortcomings of the SAH, but mostly they either address specific scenarios or are not as robust as the SAH.

2.4 Instancing geometry

Instancing geometry means creating the bottom level acceleration structure (BLAS) for one geometry group and then using the same group instance in the top level acceleration structure (TLAS) multiple times. Instancing geometry helps minimize memory consumption and overhead time needed for copying smaller buffers between CPU and GPU. Yet each instance node can still have separate data describing the instance's material, scale, rotation, and translation.

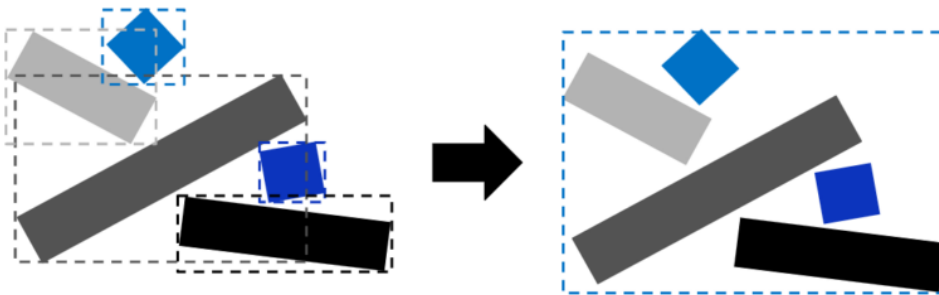


Figure 2.4: Independent BLAS instances with overlapping bounding boxes (left) and one merged BLAS instance (right). Image courtesy of Sjöholm [Sjö22].

Instancing reduces memory usage, but sometimes for the sake of faster trace. Bad mesh organization may lead to ineffective traversing of the scene. It is recommended to cluster multiple instances into a single geometry group should their bounding boxes overlap heavily, as seen in Figure 2.4. In Figure 2.5, on the other hand, splitting into smaller separate groups is best practice if there is a lot of empty space in the geometry group [Sjö22].

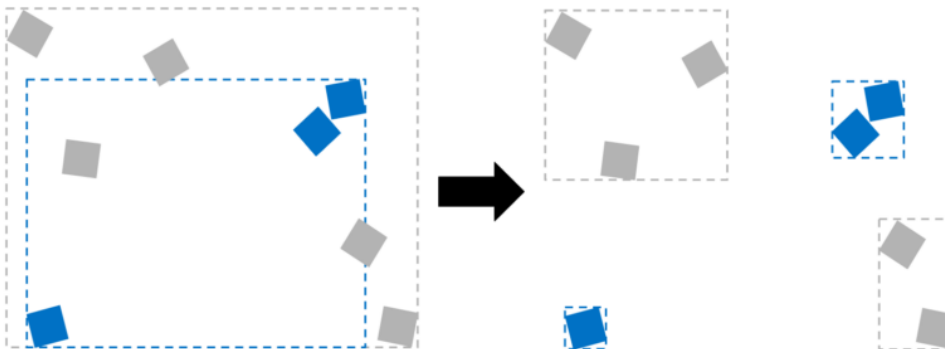


Figure 2.5: Geometries in two BLASes with overlapping bounding boxes with a lot of empty space (left). After geometries are split into four independent BLASes, the bounding boxes no longer overlap (right). Image courtesy of Sjöholm [Sjö22].

2.5 RTX architecture

New RTX architecture with the code name Turing designed by NVIDIA was described as “the biggest architectural leap forward in over a decade” [Kil+18]. The major innovation of the Turing GPU is the new architecture of the streaming multiprocessor, which contains eight Tensor Cores and one RT Core. Tensor Cores are focused on matrix computations essential for deep learning operations, such as deep learning supersampling. RT Cores are specified for quick ray-box and ray-triangle intersection tests. Figure 2.6 compares older Pascal architecture using software emulation and newer Turing architecture using RT Core’s hardware acceleration search in BVH [Kil+18].

Software Emulation for BVH Search

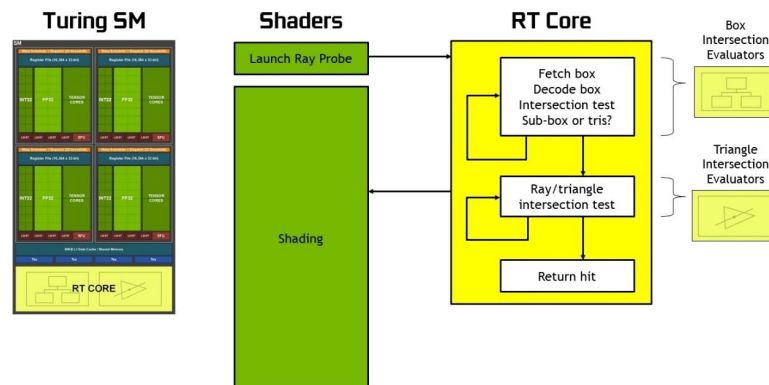
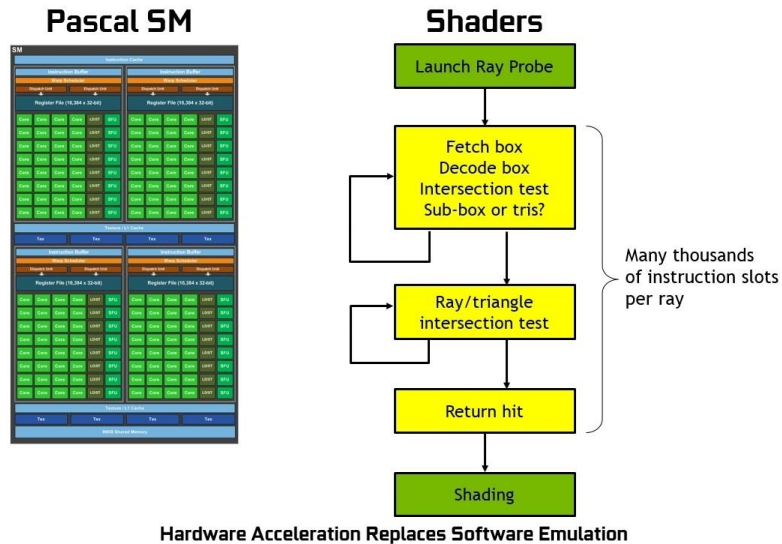


Figure 2.6: Ray tracing pre-Turing (top), Turing ray tracing with RT cores (bottom). Image courtesy of NVIDIA [Kil+18].

2.6 OptiX

NVIDIA OptiX is a general-purpose, highly programmable ray tracing API. In contrast with previous versions, OptiX 7.0+ offers developers a new low-level CUDA-centric API with direct control of memory, compilation, and launches [NVI22b]. When referring to OptiX in this report, further on OptiX 7.6 is explicitly meant. Particular shading operations are computed in executable code on the GPU called a program, also known as a shader in DXR and Vulkan. Traversal of the scene is one of eight interconnected programs. The relationship of programs is visualized in Figure 2.7, where gray represents user programs and green fixed-function, hardware-accelerated operations.

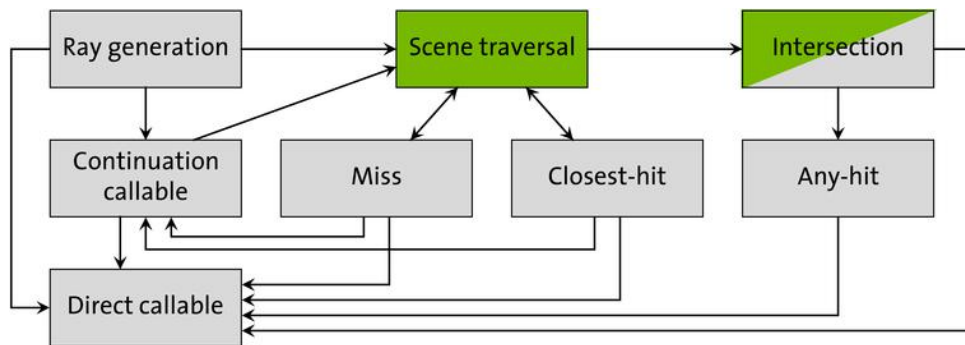


Figure 2.7: Relationship of NVIDIA OptiX 7 programs. Green represents fixed functions; gray represents user programs. Image courtesy of NVIDIA [NVI22a].

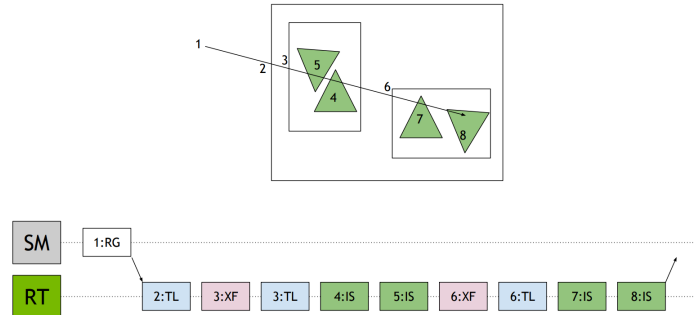
Geometric data is connected to programs via the Shader Binding Table (SBT). The SBT consists of records. Each record has two regions, header and data. The record header serves OptiX to describe programs available for specific primitives. The record data is used by programs to retrieve primitive or program-specific parameter values.

Ray tracing queries are initiated with `optixTrace` call. The ray payload may be passed to other programs through `optixTrace`, but the payload is limited to thirty-two 32-bit integer values. Each one can be read from and written to by getter/setter functions (`optixGetPayload_x` and `optixSetPayload_x`, $x \in [0, 31]$). If the payload is not sufficiently large, a pointer to stack-based variables or application-managed global memory can be encoded in it.

Acceleration structures in OptiX are built on the device. Their implementation and data layout is GPU architecture-specific but based on the BVH model. Two basic types are provided, geometry acceleration structure (GAS) and instance acceleration structure (IAS). GAS is built over primitives and is the same as BLAS in DXR or VulkanRT. IAS is similar to TLAS in DXR and

VulkanRT with the advantage of supporting multi-level instancing. Keeping IAS shallow is recommended because traversing multiple layers of IAS brings round trips between streaming multiprocessors and RT Cores [Har19]. Figure 2.8 shows conceptual execution models of shallow IAS, where every program runs on RT Core, and multi-level IAS, where transforms have to be computed back on streaming multiprocessor.

RTX traversal: 1 level instancing (2 lvl scene)



RTX traversal: 2 level instancing (3 lvl scene)

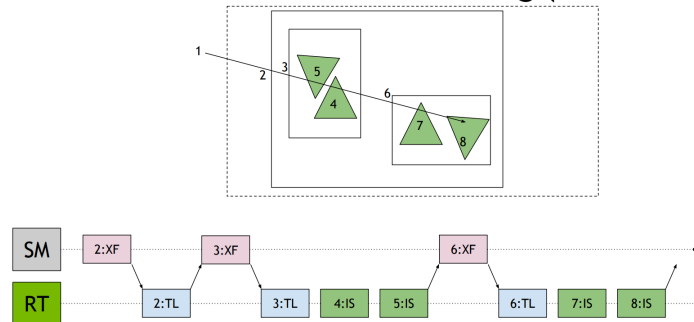


Figure 2.8: Comparison of program conceptual execution models between shallow (2-level scene, top) versus deeper multi-level IAS (3-level scene, bottom). Image courtesy of Hart [Har19].

Building acceleration structures with OptiX 7 doesn't allow for extensive control over the choice of BVH algorithm construction, but it may be affected by build flags. Two mutually exclusive flags are *PREFER_FAST_TRACE* and *PREFER_FAST_BUILD*. Allowing refitting with *ALLOW_UPDATE* is useful for dynamic geometry. Table 2.1 summarizes possible build flag options and states what properties should acceleration structure hold. *ALLOW_COMPACTION* build flag is most efficient with static geometry and might be reasonable with dynamic geometry, which doesn't update often and has a long lifetime. Compaction requires another build pass, and spending additional build time on dynamic geometry every few frames will bring no benefit, but the opposite [Dun19].

	FT	FB	AU	Properties	Example
1.	✗	✓	✗	Fastest possible build. Slower trace than 3. and 4.	Fully dynamic geometry like particles, destruction, changing prim counts, or moving wildly (explosions, etc.), where a per-frame rebuild is required.
2.	✗	✓	✓	Slightly slower build than 1., but allows very fast update.	Lower LOD dynamic objects, unlikely to be hit by too many rays but still need to be refitted per frame to be correct.
3.	✓	✗	✗	Fastest possible trace. Slower build than 1. and 2.	Default choice for static level geometry.
4.	✓	✗	✓	Fastest trace against updateable AS. Updates are slightly slower than 2. Trace a bit slower than 3.	Hero character, high-LOD dynamic objects that are expected to be hit by a significant number of rays.

Table 2.1: Combination of flags with their properties and examples. FT - Fast trace, FB - Fast build, AU - Allow update [Dun19].

2.7 OWL - OptiX 7 wrapper library

OptiX 7 wrapper library (OWL) is an abstraction layer on top of OptiX 7 created by Ingo Wald. [Wal20a; Wal20b] It is a productivity-oriented library and aims to make it easier to write OptiX programs. OWL provides higher-level abstractions than OptiX with CUDA for operations such as creating device buffers, uploading data, building shader programs and pipelines, building acceleration structures, etc. Although there are some features currently not supported by OWL most of the key concepts are supported, such as:

- Buffers realized via CUDA allocated memory
- Abstraction for geometries and geometry types
- Creation of groups and acceleration structures
- Instancing and multi-level instancing through Instance Groups
- Abstraction/Automatic creation of SBT, Programs, etc.
- Multi-device rendering

- Parametrization of device-side Geometries, Launch Params, etc. via Variables
- Support for launch parameters, asynchronous launches, and CUDA interop

Passing information from the host program to the device counterpart is performed via the OWL variables concept. Variables are used to specify data for Geometries and their Closest-Hit, Any-Hit, Intersect, and Bounds programs, RayGen programs, Miss programs, and Launch Parameters. Setting up variables consists of four steps:

1. Declaring the device-side C++/CUDA class or struct that the program(s) operate on the device.
2. Creating an `OWLRayGen`, `OWLMissProg`, `OWLGeoType`, etc. on the host that describes this device-side struct, and what Variables/members this class has.
3. Assigning values to these objects' variables on the host side.
4. OWL takes care of getting these values to the device(s), putting them into the SBT or OptiX Launch Parameters, etc.

The initial two steps are shown in Listings 2.3, 2.4, where a simple `struct` with triangle mesh and material data is declared for the device data in the former example and declaring of the set of variables using an array of `OWLVarDecl` is declared in the latter one. After declaring an array of `OWLVarDecl`, it is used for the creation of `OWLGeoType`. Launch Params are created in the same fashion but with their respective objects and functions. `OWLGeoType` has `OWLParams` counterpart and is created via call to `owlParamsCreate` function with same input layout as `owlGeoTypeCreate`.

```

1  struct TriMesh {
2      float3  *vertices;
3      float3  *normals;
4      int3    *indices;
5      struct {
6          float3          diffuseColor;
7          cudaTextureObject_t texture;
8      } material;
9  };

```

Listing 2.3: Example of simple struct declaration for device data

```

1   OWLVarDecl triMeshVars[] = {
2       { "vertices", OWL_BUFPTR,
3         OWL_OFFSETOF(TriMesh,vertices) },
4       { "normals", OWL_BUFPTR,
5         OWL_OFFSETOF(TriMesh,normals) },
6       { "indices", OWL_BUFPTR,
7         OWL_OFFSETOF(TriMesh,indices) },
8       { "diffuseColor", OWL_FLOAT3,
9         OWL_OFFSETOF(TriMesh,material.diffuseColor) },
10      { "texture", OWL_TEXTURE,
11        OWL_OFFSETOF(TriMesh,material.texture) },
12      { nullptr /* sentinel to mark end of list */ }
13  };
14
15  OWLGeomType triMeshGT = owlGeomTypeCreate(context,
16      OWL_GEOM_TRIANGLES,
17      sizeof(TriMeshVars),
18      triMeshVars,
19      -1);

```

Listing 2.4: Example of declaring *OWLVarDecl* and creating of *OWLGeomType* for the triangle mesh struct

Most of the types, like `OWL_FLOAT`, `OWL_INT3`, etc., are copied to the device, but for objects as buffers and textures, additional translation is needed. `OWL_BUFPTR` member is formed as `OWLBuffer` on the host, and OWL will assign the correct address for data on the device side. Likewise, `OWL_TEXTURES` is set to `OWLTexture` and is written to `cudaTextureObject_t` on the device side. It is important to notice that OWL does not type-check if underlying data corresponds to declared types, so undefined behavior may occur if the same-sized types are used. However, setting values with totally incompatible types will result in error.

The third step is to assign values to variables. OWL provides helper functions with naming convention `owl<Object>Set<Type>`. Objects which have member variables (e.g. an `OWLParams`, an `OWLRayGen`, or an `OWLGeom`) may have those variables set using said helper functions. Assigning `OWL_FLOAT3` to `OWLRayGen` would be accomplished with function call `owlRayGenSet3f(...)`. Frequently changed data like camera position and direction should be set using Launch Params as changing and copying them is quick and inexpensive. On the contrary, Geometries, Miss programs, RayGen programs, etc., are placed in the SBT, which means that any changes would be visible only after rebuilding the SBT. For the scenes with a large number of Geometries, rebuilding the SBT ever so often would incur significant performance tax.

The last step, after successfully arranging variables and building the SBT, is accessing variables on the device side of the code. Gathering objects' data from SBT might be achieved with direct `optixGetSbtDataPointer()` or with owl wrapper calls as shown in the Listing 2.5. Launch Params reside in global `__constant__` memory `optixLaunchParams`.

```
1   OPTIX_CLOSEST_HIT_PROGRAM(TriMesh_ClosestHit)
2   {
3       const TriMesh &mesh = owl::getProgramData<TriMesh>();
4       int3 indices = mesh.indices[optixGetPrimitiveIndex()];
5       ...
6   }
7
8   OPTIX_RAYGEN_PROGRAM()
9   {
10      ...
11      const Camera &camera = optixLaunchParams.camera;
12      ...
13  }
```

Listing 2.5: Example of accessing SBT data and Launch Params in the device programs



Chapter 3

Related work

This thesis focuses on effectively organizing the scene, leveraging the maximum potential of RTX architecture with OptiX API. In this chapter, three main ideas are analyzed.



3.1 Scene layout optimization

The first article investigates the possibilities of automatic optimization of scene structure prior to building acceleration structure using modern graphics APIs. Káčerik and Bittner [KB23] focus on the performance differences between rendering times of scenes built as a hierarchy provided by the scene author with one BLAS per instance and a scene as one compact BLAS. Different layouts of said approaches can be seen in Figures 3.1, 3.2.

Merging all scene geometry and its information into a single BLAS brings benefits to acceleration structure builder. Knowing the positions and relations of all objects in the scene has the potential to improve the resulting AS. While it may be extremely efficient, especially in scenes with clusters of different objects overlapping or in small localities, it also makes elements like instancing, refitting, or object-specific shading essentially unusable.

According to the research done by authors on eight static scenes with six different configurations (three builder flag configurations with two BLAS

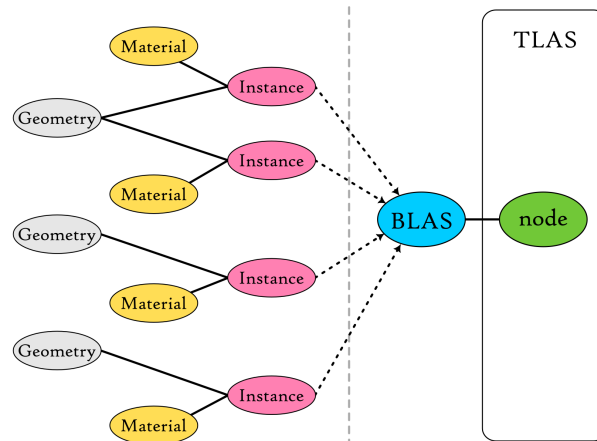


Figure 3.1: Scene graph structure in one BLAS. Image courtesy of Káčerik and Bittner [KB23].

methods), creating a single BLAS was mostly a superior approach in terms of generating rays per second. On average, a single BLAS setup was shooting 1.3 times more rays per second than a BLAS per object setup. Regarding build times, the latter method was faster, which authors expect to be achieved by a concurrent build of BLASes and $\mathcal{O}(n * \log(n))$ complexity of the acceleration structure construction algorithm.

From the testing of various scenes and setups, one specific scene stood out. In the Fireplace scene, BLAS per object strategy with the *PREFER_FAST_TRACE* build flag reached better results in both ray tracing speed and build time. This finding hints that there might be such an arrangement of the scene objects, that is at the same level with a single BLAS strategy in performance yet has the ability to use features that are not supported by the other approach. Two directions for advancement are discussed:

- Minimization of nodes' overlaps
- Finding suitable rotation of BLAS to reduce overlaps

3.2 Using instancing as hardware OBB test

The second analyzed article uses OptiX for hardware-accelerated visualization of long, thin primitive types. That is an exceptional case of the scene because it might lead to many overlapping and poorly aligned objects, which most likely

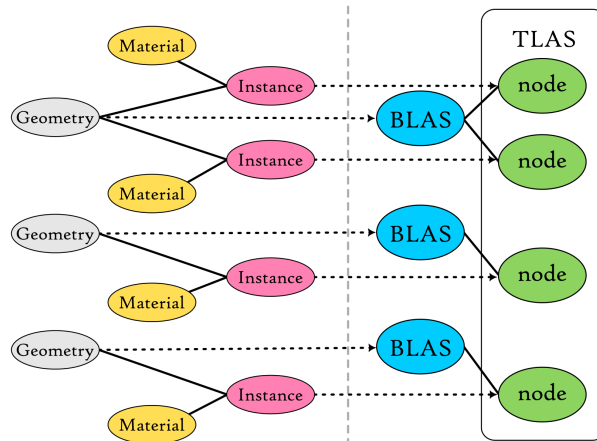


Figure 3.2: Scene graph structure in BLAS per object. Image courtesy of Káčerik and Bittner [KB23].

leads to an acceleration structure with significantly suboptimal performance. The idea formulated by Wald et al. [Wal+20] uses instancing to perform the transformation to effectively get an oriented bounding box (OBB) without switching context from RT Core to streaming multiprocessor as is illustrated in Figure 3.3.

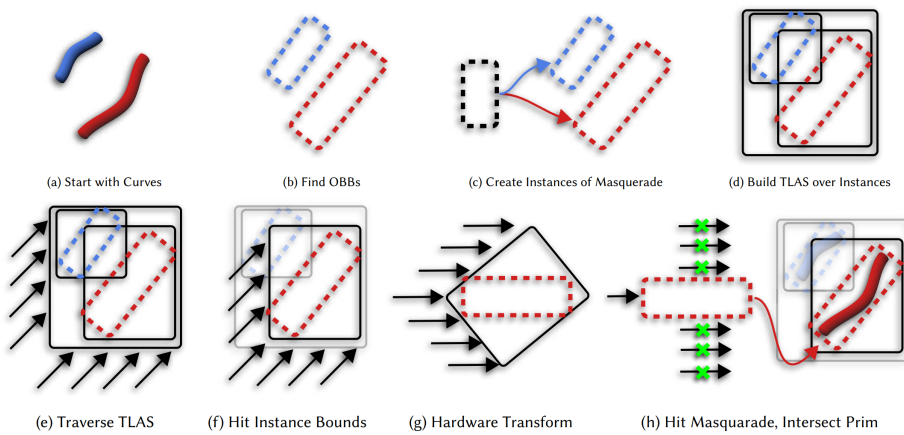


Figure 3.3: Illustration of masquerading technique using instance transforms as OBB test. Image courtesy of Wald et al. [Wal+20].

Speed up is gained by potentially reducing the number of rays that actually hit the bounding box of encapsulated primitive. After the ray hits the bounding box context switch is required since the primitive inside is not a triangle, and thus intersection test is not hardware-accelerated. Another downside of this approach is that instancing is used for instantiating the OBB rejection test and can not be used as intended. That might be extremely

beneficial for bounding boxes of thin and long primitives but might not improve performance in most cases.

3.3 Partial rebraiding of two-level BVH

Two-level BVHs are often built in a way that separates levels with top-level BVH consisting of logical objects and not geometric primitives. Consequently, two-level BVHs reduce build times at the cost of increased traversal times compared to flat BVHs. Benthin et al. [Ben+17] claim that performance-wise traversal is typically worse by a factor of 1.5–2x in comparison with flat BVH due to overlaps of top nodes.

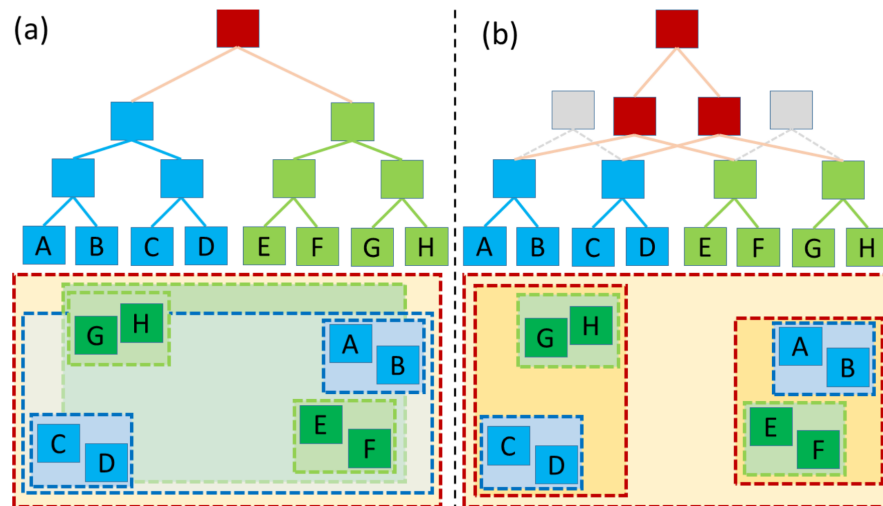


Figure 3.4: Illustration of re-braiding method: (a) two objects (green and blue), each with their own BVH; with their topologies (top), and their spatial extents (bottom). As the objects spatially overlap and the top-level BVH (brown) has to treat them as monolithic entities a significant BVH node overlap in the spatial domain occurs, leading to low traversal performance. (b) Re-braiding method allows the top-level BVH to look into the object BVHs, and to “open up” object nodes where appropriate. This allows the top-level BVH to create new top-level nodes (red) that address individual subtrees in the object BVHs, resulting in improved BVH quality. Image courtesy of Benthin et al. [Ben+17].

Authors [Ben+17] propose a way to tackle the ineffective traversal at the top nodes by incorporating new method after classic two-level BVH build. They describe overview of the BVH build in three steps:

1. start with object BVHs in the same way a traditional two-level BVH would;

2. find a suitable “cut” through each object’s BVH such that the resulting set of BVH subtrees has low(er) overlap;
3. build a top-level BVH over those resulting subtrees.

Finding the optimal cut is a difficult and time-consuming task. Commonly, various heuristics are used to significantly reduce time complexity in spite of getting sub-optimal results. Benthin et al. [Ben+17] use BVH node build references (BRefs), which is their structure used in the re-braiding method. Listing 3.1 shows BRef’s structure contents, where `ref` is a reference/pointer to a BVH node inside an object BVH (initialized with the root BVH node), `bounds` contains world-space bounding information including instance transformation and `objectID` is the ID of object/instance the BVH node belongs to. The variable `numPrims` is the estimated number of primitives in the subtree, which the builder uses as a coefficient in the SAH cost estimation of the bounding box and the binning step.

```

1     struct BRef {
2         BVHNodeReference ref;
3         AABB bounds;
4         unsigned int objectID;
5         unsigned int numPrims;
6     };

```

Listing 3.1: BRef structure used in the re-braiding method [Ben+17]

The article states that the construction algorithm used for top-level BVH merges with the second step in a recursive top-down fashion. At the beginning, a list of BRefs is formed, one per each object. This initial list forms one whole segment, which is then recursively partitioned in these four steps:

1. Open BRefs in the current segment according to some opening heuristic;
2. Replace opened BRefs with BRefs to their children;
3. Apply a SAH-based binning on the current segment and split it into sub-segments;
4. Apply these steps to left and right sub-segments until termination criteria are met.

To sum up, the main advantage of this method over classical two-level BVH is the ability to open up and intertwine bottom-level BVHs, which, according to

the study [Ben+17], significantly reduces big overlaps of upper nodes. While RTX API provides BVH builder as a black box, the re-braiding method and its approach to finding cuts between GASes (bottom-level) and IAS (top-level) in OptiX will be further explored.



Chapter 4

Implementation

In this chapter, implementation details of the experimental application are described. The first section breaks down algorithms for scene optimization. The subsequent section provides more information about visual tools used during experimentation. The following section explains the usage of UI elements and their connection to rendering and scene organization algorithms. The next section presents scripts used for the automation of tasks related primarily to the testing. Finally, the last section states which third-party libraries were used and how they were incorporated.



4.1 Scene optimization

Every scene has its specific object organization based on how the author created or processed it. Some scenes contain many objects for the convenience of the artist's manipulation and creation; others may be optimized by the exporter and grouped by materials or based on groups in the modeling tool. Such arrangements might be efficient for human readability but may hinder the building of acceleration structures.

Four approaches to scene optimization for OptiX GAS builder are explored in this thesis. The first two use the initial object layout from the OBJ Wavefront file. The `tinyobjloader` library [Fuj+24] provides OBJ file loading and stores vertices, normals, and texcoord information in `attrib_t` structure. Face indices are stored in `shape_t` and material information in `material_t`.

Loaded data are then processed into `TriangleMesh` structures. Every shape is divided into these meshes by unique material. The first method uses all meshes as build inputs for one `owlGroupBuildAccel` call, while the second approach calls builder for each object. The former is later addressed as 'Single GAS', while the latter is 'Multi GAS' or 'GAS per object'.

The third approach is processing the scene further by applying binned SAH BVH builder on each object. The builder has a constraint on the maximum number of available nodes, which is proportional to the number of triangles in the scene. At the beginning of the algorithm node for each object is pushed in the queue, and then while cycle starts and pops one node at a time. The node is split; if the total node limit is not reached, the current node has more than eight triangles, and the area of its bounding box is bigger than the threshold value. Splitting is done on the axis with the greatest extent of the node's bounding box and uses sixteen bins for the triangle centroids. After the binning process, the best-splitting plane is chosen, and the mesh is split accordingly. Implementation is based on Bikker's article [Bik23] on real-time ray tracing.

```

1 while True
2     for i = 0 to object.size()
3         for j = i + 1 to object.size()
4             if overlaps(O_i,O_j)
5                 and intersection_area(O_i,O_j) > skipThreshold
6                     push (overlap, i, j) in priority_queue
7                     total_overlap += overlap
8 overlap_ratio = total_overlap / model_area
9 if overlap_ratio < termination_criterium return
10
11 while priority_queue is not empty
12     pop (overlap, i, j) from priority_queue
13     if i or j in dirty_set continue
14     pair_overlap_ratio = overlap / bounds(O_i, O_j)
15     if pair_overlap_ratio < split_and_merge_threshold
16         if pair_overlap_ratio < split_threshold
17             split(O_i)
18             split(O_j)
19         else
20             split_and_merge_overlap(O_i, O_j)
21     else
22         merge(O_i, O_j)
23     append i, j to dirty_set
24 clear dirty_set

```

Listing 4.1: Overview of the overlap reduction method in pseudocode

The last method explicitly addresses the problem of object bounds overlapping. We are calling it the 'Overlap reduction' method. An overview of the method is in the Listing 4.1. The algorithm runs while there are overlaps to split or merge, or until termination criteria are met. Overlap of each object pair is pushed to the priority queue if the overlap is greater than a specified threshold. After populating of the priority queue with tasks another cycle starts. In every loop pair with biggest overlap is popped from the queue and gets processed, if both objects were not modified yet. Three branches follow depending on the ratio between overlap and unified bounds of objects: splitting of both objects, splitting of both objects with merge on the overlap, and merge of both objects. After modification, object id is noted and that object can not be modified until new overlap is found.

For object splitting, spatial median split in the dimension of greatest extent is used. Firstly, the centroids of the triangles are calculated, and then triangles are arranged in two subsets accordingly.

Split and merge branch does the splitting phase with the overlap in the process. Triangle centroids are split into two sets; one contains triangles inside overlap, and the other contains the ones outside. If one of the sets is empty, object is split with spatial median split method. Merge phase follows, which is same as in merge branch.

Merging two objects is straightforward as long as they have the same material. If the material differs between objects, one material is chosen. Materials with textures need texcoord information. If one object does not have texcoords, texcoord vector is padded with zeroes.

■ 4.2 Visual tools

In the experimental phase, where the influence of the parameters has been investigated to obtain an overview of their threshold values and trends, visual tools played an important role. We have implemented several different views of the scene for various purposes. Part of the views are similar to rendering options in Nsight Compute's Acceleration Structure Viewer, but can be used directly in the running app and changed on demand. In the early stages of this thesis, some of the Nsight Compute's features were not available or not fully functional with OptiX 7.6. Our views, which are described further in this section in greater detail, are:

- Transparent model view
- Heatmap view
- AABB preview

The first visualization from the list above is the transparent model view. Figure 4.1 is an example of a view over the Sibenik scene with random colored material set to every leaf node of the BVH build. This view is implemented as special ray type, which has its own ray generation program and dedicated closest hit and miss programs. The core of the ray generation program is a loop, which terminates if a miss program was called or our defined maximum of 1024 hits was reached. Each step of the loop calls `optixTrace` function. After tracing, color from the payload is accumulated, the last hit position is noted for use in the next call of `optixTrace` in the consequent step of the loop, and finally boolean if anything was hit is set.

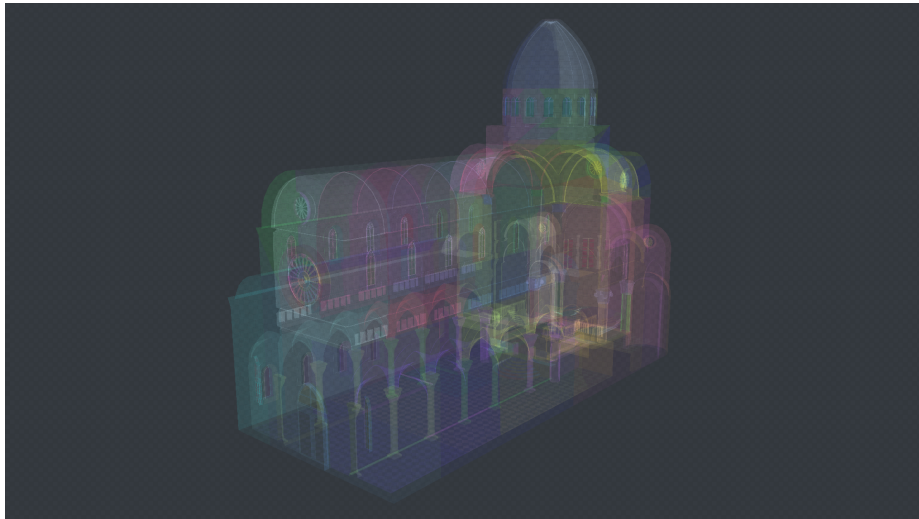


Figure 4.1: Example of transparent model view with BVH build configuration on Sibenik scene.

After the loop terminates, accumulated color is divided by our math function according to the number of hits in the loop. The function is defined as $(N_{hits}/1024) + 1/32$, where N_{hits} is the number of triangles hit in the scene by the ray.

The closest hit program is very simple and minimal. Firstly, it collects data from the shader binding table for the primitive that is hit. Afterward, the color is set in the payload structure, the surface position of the intersection is calculated and stored in the payload, and the hit flag is set.

The miss program sets the background to the checkered pattern, as is usual in many graphical applications where transparency is used. Tiling is based on the launch index, which is fetched by `optixGetLaunchIndex` function.

The next visual tool is based on the timing of `optixTrace` function. In the ray generation programs, before first tracing, a timestamp is marked by the `clock` function. After the tracing is finished, another timestamp is marked, and the time delta is calculated. This value is scaled by a user-defined arbitrary scaling factor, which can be adjusted at the runtime. The scaled value is finally passed to the color mapping function, resulting in the heatmap of the scene, as can be seen in Figure 4.2.

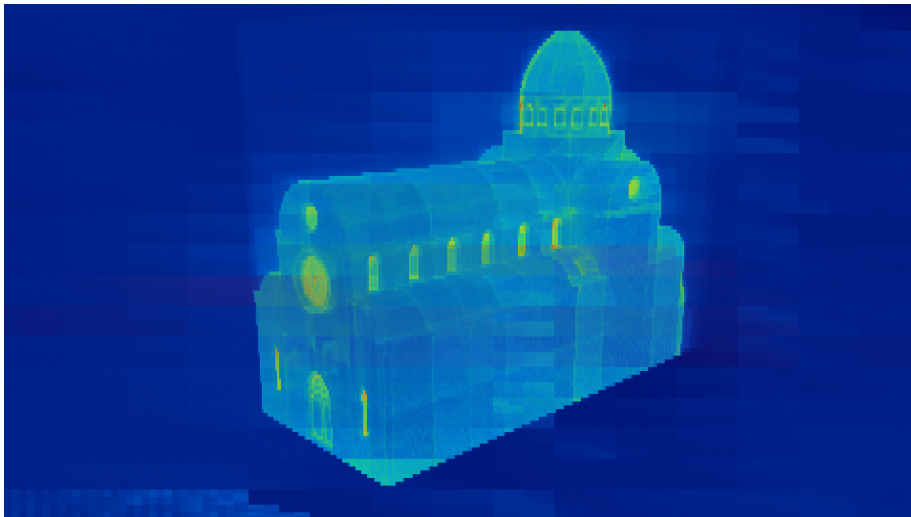


Figure 4.2: Example of heatmap model view with BVH build configuration on Sibenik scene. Render is accumulated over a few seconds.

The third and final implemented visual tool is the AABB model view, which is based on creating bounding boxes for each object or node of the BVH tree. This set of bounding boxes is used to construct single GAS, and OptiX visibility masks are utilized to differentiate between scene, AABB and instances set. Users can set masks at runtime to choose which groups are visible. Each bounding box is made of 12 right-angled triangles, two for each side of the box. There is no dedicated ray generation program for this view. In the closest hit program for bounding box objects, the distance from the intersection point to the catheti is calculated, and if it is greater than the threshold value in both cases, traversal continues. Figure 4.3 shows example of the view with visibility mask set only to the AABB objects group.

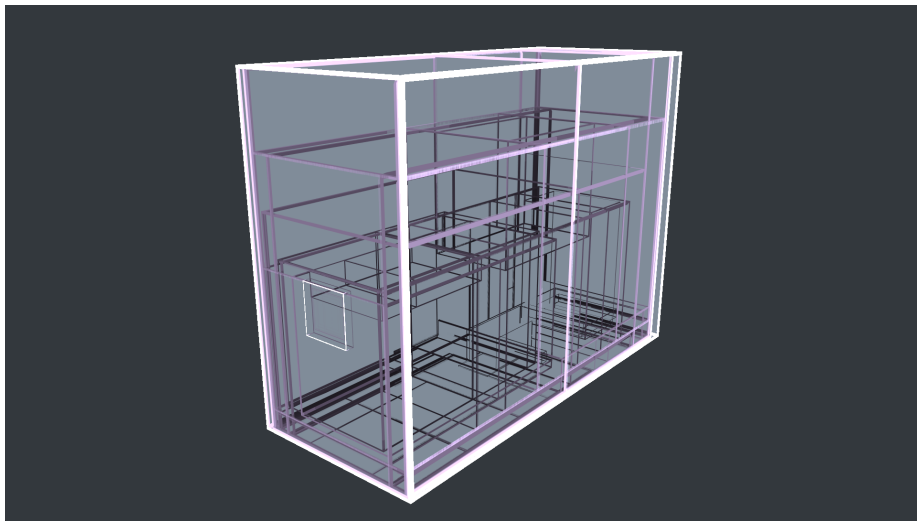


Figure 4.3: Example of AABB model view with BVH build configuration on Sibenik scene.

4.3 GUI

The graphical user interface implementation uses the Dear ImGui library and is divided into three parts. The first part consists of the performance report information in a tab on the left side of the application window. Standard C++ chrono library calls are employed to measure the time performance of individual tasks, such as AS build, scene update, render, and draw times. The number of traced rays is counted and reported in the performance tab between renderings. The other two parts are in two tabs on the right side of the application window and serve as an interactive interface.

The refresh rate of the reporting tab can currently be set only from the source code, and its default value is 0.1s. Every tenth of a second, new values of these eight stats are presented:

FPS Number of frames render divided by seconds since last update

Update t Measured time of most recent animation matrices calculation

UpdateAS t Measured time of most recent IAS build/refit

Render t Measured time of most recent tracing scene

Display t Measured time of most recent OpenGL draw

Pri MRays/s Primary rays divided by seconds since last update and 10^6

Sha MRays/s Shadow rays divided by seconds since last update and 10^6

Sec MRays/s Secondary rays divided by seconds since last update and 10^6

The interactive part of the GUI on the right side is split into two logical parts. The upper tab is tied to the scene overlap reduction method. The current number of meshes, overlap ratio, and current SAH (if BVH is used) are reported at the top of the tab. Two sliders follow, with the first affecting the reduction method's threshold between splitting and merging branches. The second slider sets the threshold for overlap skipping. Button *Reduce overlap* calls the reduction method with a termination threshold set to *infinity*, which effectively means that the reduction method will make a single loop, allowing the user to apply reduction with different parameters in each step.



Figure 4.4: Detailed view of implemented GUI tabs. Reporting tab on the left, Overlap info and adjustments in the middle, and rendering options on the right.

The bottom tab of the interactive part is connected to rendering options. The first ImGui ListBox comprises two ray generation programs, which are radiance and transparent view programs. The second ListBox allows to choose one of the three rendering modes: render, heatmap with the jet scheme, and heatmap with the cividis scheme. The heatmap view has an adjustable heat scale, which may be altered by the slider. Next, pixel samples may be increased or decreased by arrow buttons. The same applies to adjusting the number of light samples. Tracing of secondary rays can be toggled with a

checkbox. Camera animation around the point of interest set in the scene configuration file can be toggled with another checkbox. Lastly, a group of three checkboxes manages the visibility mask of tracing for the scene model, AABB of objects, and instances.

4.4 Automation scripts

At the heart of the automation are two types of configuration files for the application. One configuration file describes the scene, while the other regulates application environment. Both file types are laid out one variable per line in key-value pairs separated by '='. Implementation of both underlying structures is same and uses variables mapped to enum, which is employed in parsing of configuration files. Extensions 'scn' and 'env' are used for the scene and environment files, respectively.

Scene structure provides means to set models, camera settings, lighting information and more. All parameters that can be used are listed in Table 4.1. Environment structure holds information about scene optimization, build configuration and visualization settings. List of all parameters is in Table 4.2. Additionally, the environment file contains a list of N positions (3x float) for each instance and M lines for an animation description (7x float) of each animated instance.

```
1  {
2      "actions": [
3          {
4              "type": "run",
5              "scenes": ["san_miguel.scn",
6                          "san_miguel_1000.scn",
7                          "san_miguel_10000.scn",
8                          "san_miguel_100000.scn"],
9              "envs": ["singleGAS.env",
10                       "multiGAS.env",
11                       "bvh_1e-3.env",
12                       "reduce_greedy.env"]
13          }
14      ]
15  }
```

Listing 4.2: Example of JSON configuration file for automation scripts

Parameter Key	Description	Value
SCENE_NAME	Name of scene file used in the log's filename	string
MODEL_FILE_NAME	.obj file name for the scene model	string
INSTANCE_FILE_NAME	.obj file name for the instances model	string
CAMERA_POSITION	Initial position of the camera	3x float
CAMERA_POI	Initial camera point of interest	3x float
CAMERA_ROTATION_SPEED	Frequency of camera revolutions around point of interest	float
ANIMATE_CAMERA	Boolean for camera animation	bool (0/1)
INSTANCE_COUNT	Number of instances in the scene	int
ANIM_INSTANCE_COUNT	Number of animated instances in the scene	int
LIGHT_ORIGIN	Area light's position of lower corner	3x float
LIGHT_SIZE	Length of area light's sides	float
LIGHT_POWER	Area light's emission power	float

Table 4.1: All scene file parameters with description and value types.

All automation scripts and helper functions are written in Python. Scripts serve three purposes: scene generation, automated testing, and presentation of results. There is a separate module for each purpose so that they may be used separately, but there is also a config module that functionally glues them together.

The config module must be provided with a 'json' description file to run. The `actions` property at the root of the JSON file holds a list of all actions to be executed. Each action must have `type` property set to call the desired module. Three types are currently valid: `generate`, `run`, and `plot`. Example of a JSON file that runs San Miguel scene in four configurations and four environments can be seen in Listing 4.2

Generate action type operates on the scene files and, with the parameters provided, generates location and animation data for the instances. Parameters are coupled in a dictionary accessible with key `params`. The dictionary must contain `model_mask` key with a list of indexes to scene models, `n_all` and `n_anim_all` lists with numbers of location and animation data, respectively, to be generated. Base velocity and velocity deviation must be specified by `v` and `v_dev` if any animation data are expected. The same applies to base distance and distance deviation.

Parameter Key	Description	Value
BUILD_FLAGS	Optix build flag value	int
SCENE_REBUILD_INTEREVAL	Number of frames until IAS rebuild	int
ENVIRONMENT_NAME	Name of environment file used in log's filename	string
WIDTH	Width of application window	int
HEIGHT	Height of application window	int
HEATSCALE	Value of heatscale range	float
PIXEL_SAMPLES	Number of samples per pixel	int
SECONDARY_RAYS	Boolean for secondary rays	bool (0/1)
LIGHT_SAMPLES	Number of samples per light	int
SOM_THRESHOLD	Threshold for split or merge branches in overlap reduction	float
SKIP_OVERLAP_THRESHOLD	Threshold for overlap's size to skip	float
OVERLAP_THRESHOLD	Termination threshold for overlap reduction	float
MIN_SIZE	Minimal number of triangles in the mesh before splitting	int
SINGLE_GAS	Boolean for GAS build algorithm	bool (0/1)
USE_BVH	Boolean for scene optimization	bool (0/1)
NODES_TO_TRIS	Ratio of BVH's nodes to scene triangles	float

Table 4.2: All environment file parameters with description and value types.

Run action type needs two lists on input: one containing scene files and the other environment files. Application is then executed $N_{scenes} * N_{envs}$ times with each distinct pair. Between each run, the script waits for the decrease of GPU temperature below 60°C. Nvidia_smi module is used to read GPU temperature data. After the application is executed, the log filename is added to the list of logs for further usage.

The plot action type may employ the list of log files from previous actions and/or use log files from scenes and environments provided as two lists in the same way as for the run type. The `plot_type` specifies how are data expected to be presented, whether with a line graph or a bar chart. Plots are created with the Mathplotlib library, and `plot_metadata` carries a dictionary with all necessary annotation information, such as title, axis labels, and output filename. The most important part of metadata is `value_data` because it specifies which data from the log file are processed in the graph.

■ 4.5 Third-party libraries

Experimental application written in C++ uses six third-party libraries altogether, namely:

OptiX 7.6 Backbone of application's rendering domain

OWL Efficient abstraction layer of OptiX library

GLFW3 Support for window handling and input capture

TinyObjLoader Loading of models in OBJ file format

stb_image Provides reading of textures and writing of images

Dear ImGui Graphical user interface library

Automation scripts implemented in Python use these 3rd party modules:

NumPy Array data handling and math operations

NVidia_SMI GPU temperature reading

Matplotlib Graphs and charts generation

Chapter 5

Results

In the beginning of this chapter chosen scenes are presented. The description of test environment and configurations follows. Each experiment is introduced and evaluations described and supplemented with graph or table.





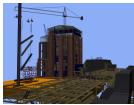

Scene						
	Fireplace	Conference	Sibenik	Sponza	Powerplant	San Miguel
Objects	24	41	15	393	56	2203
Triangles	580,486	331,179	75,283	289,405	12,759,225	9,978,412
Overlap	7.272	17.447	10.124	12.878	8.944	222.422

Table 5.1: Geometric complexity of the tested scenes. *Objects* are counted as groups/objects in the .obj file, each split into separate objects per material. *Overlap* is the ratio calculated as the sum of the surface areas of the overlapping volume of each pair of objects to the surface area of the scene’s bounding volume.

Altogether, six scenes were chosen to test the performance of our implementation. A list of scenes with their complexity information is presented in Table 5.1. The view of the scene was always chosen with two conditions in mind. The first condition was to choose a spot overlooking most of the scene. The second requirement was to contain some object at a relatively close distance.

As previously established, the overlap between the object’s bounding boxes directly affects performance. The relation between overlap ratio and rendering

performance is one of the main focuses of the tests. We measure the overlap ratio as a single value by computing surface areas of overlapping volumes of each object pair. The sum is then divided by the surface area of the scene’s bounding volume. A formula can express it:

$$Overlap = \frac{\sum_{i=1}^{objects-1} \sum_{j=i+1}^{objects-1} Surface(Overlap(V_i, V_j))}{Surface(V_{scene})},$$

where $Overlap()$ is a function calculating the overlapping volume of two bounding volumes and $Surface()$ is a function returning the surface area of volume. Overlap ratios for selected scenes prior to any optimization are in Table 5.1.

Every test is run with a config script in a batch of scenes and environments. The batch run executes a timed application build, which automatically stops after 10 seconds. GPU temperature threshold to start the application is set to a maximum of 60°C. The following experiments were all conducted on a laptop with these hardware specifications:

CPU Intel Core i7-11800H @ 2.3GHz

GPU NVIDIA GeForce RTX 3070 Laptop GPU @ 1110 MHz:

- 8 GB GDDR6 memory @ 1750 MHz
- Ampere architecture
- TGP 115 W
- 5120 cores, 40 SM units, 40 RT cores (1 per SM)

RAM 2x 16GB Samsung DDR4 @ 3200 MHz

SSD Samsung PM9A1 M.2 1TB

■ 5.1 Configurations tests

Four setups are used in the tests in the context of geometry instancing. One setup uses only a static scene model with no additional instances; the other three gradually add a number of animated instances. The first setup has 1,000 instances; the second uses 10,000 instances, and the last 100,000 instances.

Experiment 1: Parameters comparison for BVH method

Results of the BVH method depend on the number of nodes provided to the builder. This experiment tests four values for the ratio of nodes to triangles and compares the results. The tested values are: 10^{-1} , 10^{-2} , 10^{-3} , and 10^{-4} . All scene variants are tested. Most significant difference was measured on the San Miguel scene and from the results in Figure 5.1 ratio of 10^{-3} is the best in terms of tracing performance. Table 5.2 shows build information for the San Miguel scene.

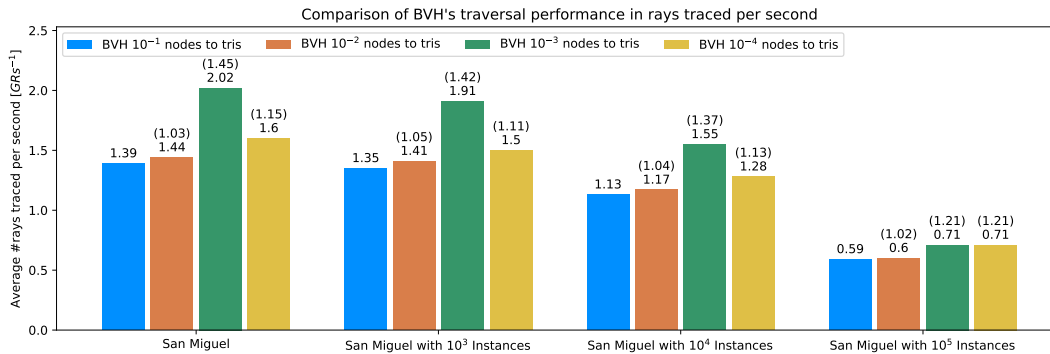


Figure 5.1: Comparison of BVH’s traversal performance on San Miguel scene. All types of rays are counted. Higher value is better.

We have also tested building GASes per object with BVH splits, but the tracing results were almost identical to building GASes per object without pre-processing.

Build method	BVH 10 ⁻¹	BVH 10 ⁻²	BVH 10 ⁻³	BVH 10 ⁻⁴
Optimization t[s]	10.657	10.575	6.536	2.963
GAS Build t[ms]	12558.8	11387.4	2164.86	1305.87
GAS Size [MB]	817.753	803.459	713.371	706.997
IAS Build t[ms]	32.126	31.139	19.941	18.056
IAS Size [MB]	16.875	14.966	1.789	0.910

Table 5.2: Comparison of BVH’s build performance and AS sizes on static San Miguel scene. The best results are in bold.

Figure 5.2 depicts the difference between the original scene and BVH’s fragmentation. Every unique mesh is shaded with a random diffuse color.



Figure 5.2: Fragmentation of the San Miguel scene is shown in random diffuse color per mesh. Left original scene, center BVH 10^{-4} , right BVH 10^{-1}

■ Experiment 2: Parameters comparison for Overlap method

Overlap method has multiple constraints that control how are scene objects processed. Two parameters are exposed and their effects are measured in this experiment. Skip threshold controls the size of overlap that is acceptable for split or merge. SoM threshold chooses which action to take.

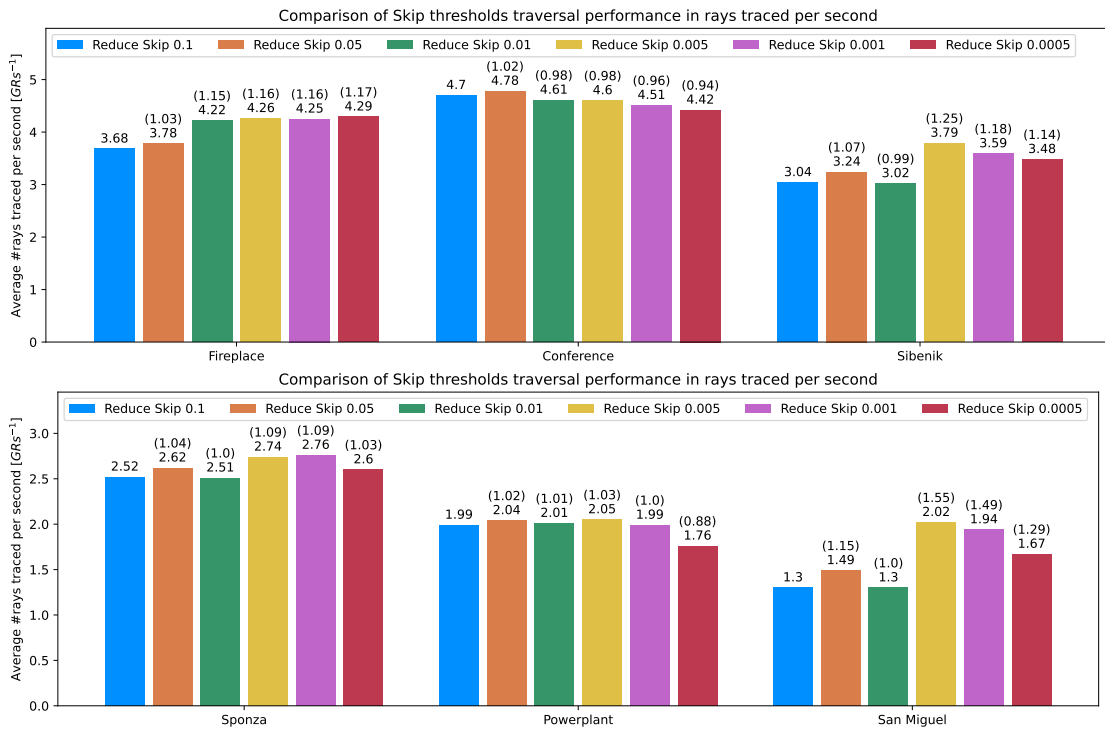


Figure 5.3: Tracing performance of skip threshold values. Fireplace, Conference, and Sibenik in the top chart. Sponza, Powerplant, and San Miguel in the bottom chart. All types of rays are counted. Higher value is better.

Figure 5.3 presents results of skip threshold set to 0.1 (blue), 0.05 (orange), 0.01 (green), 0.005 (yellow), 0.001 (pink), and 0.0005 (red). Tracing results

are in favor of 0.005 with the best results in the more complex Powerplant and San Miguel scenes. In Table 5.3 are statistics of build with chosen skip threshold. Lower overlap is achieved for all scenes except for Sponza and Powerplant.

Seven values of SoM threshold were tested ranging from 0.2 to 0.8. Highest achieved tracing performance was with value set to 0.5, as can be seen in Figure 5.4. Sponza scene was outlier again with same performance for every value.

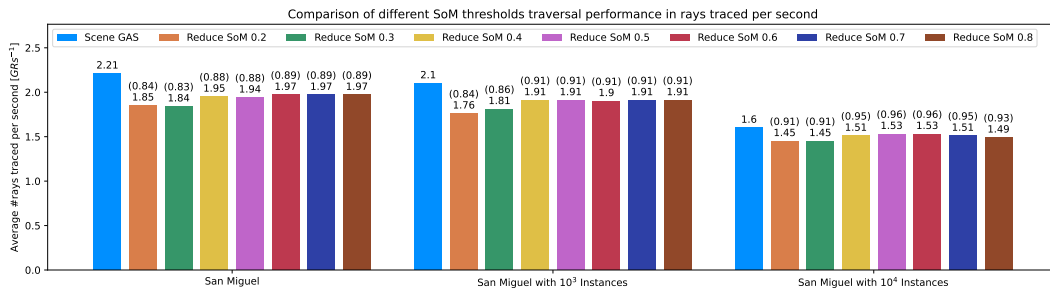


Figure 5.4: Comparison of traversal performance on San Miguel scene with different SoM thresholds. All types of rays are counted. Higher value is better.

Experiment 3: Performance comparison of rendering animated geometry

Every dynamic scene test uses uniformly random distributed instances in the scene model bounds. Density of the instances in the scene increases with their count, as they are scaled proportionally to the scene model, not the number of instances. In this experiment we measure tracing performance of animated geometry only and then compare the results with the selected scene model with instances.

Figure 5.5 shows that every scene performs roughly the same as the scene with instances only. It suggests that instances prevail in the IAS complexity. If we look at the render with so many instances, almost exclusively, instances are visible. Figure 5.6 depicts a comparison of the instanced Sibenik scenes. We went further and scaled the instanced model down. With the reduction of instances clustering, the performance of the test with instances only increased, but accordingly so in the scenes with instances. Scenes with 10⁵ instances will not be tested in further experiments, setting our limit to 10⁴ instances in the subsequent tests.

Skip threshold		0.1	0.05	0.01	0.005	0.001	0.0005
Fireplace	Optimization t[ms]	89.2	103.197	476.653	1204.66	1786.17	1817.28
	Overlap	6.69451	5.355	2.545	1.897	1.835	1.854
	Split/Merge	67/0	107/4	348/90	1015/571	2603/1897	3307/2481
	GAS Build t[s]	31.525	40.129	76.278	119.247	167.443	192.591
	GAS Size [MB]	10.214	10.27	10.613	10.973	11.517	11.764
	IAS Build t[s]	20.797	20.816	20.872	20.784	20.863	20.942
	IAS Size [MB]	0.029	0.039	0.085	0.139	0.216	0.251
Conference	Optimization t[ms]	50.667	56.715	109.647	140.115	218.107	286.401
	Overlap	10.509	10.09	9.211	9.358	8.753	8.753
	Split/Merge	48/0	64/0	198/0	403/0	1339/0	1914/0
	GAS Build t[s]	33.101	37.991	74.465	127.789	292.474	402.736
	GAS Size [MB]	23.411	23.465	23.606	24.093	26.03	27.28
	IAS Build t[s]	20.714	20.7965	20.754	21.385	17.922	21.225
	IAS Size [MB]	0.027	0.033	0.072	0.132	0.407	0.575
Sibenik	Optimization t[ms]	93.74	116.368	172.822	210.518	687.249	5078.14
	Overlap	21.756	18.104	8.545	7.097	5.115	5.112
	Split/Merge	102/0	208/0	866/0	1362/0	3860/172	7585/1503
	GAS Build t[s]	40.669	66.707	197.883	291.531	662.414	1055.8
	GAS Size [MB]	5.445	5.537	6.897	8.058	13.369	18.888
	IAS Build t[s]	20.931	21.133	20.339	21.029	21.425	18.873
	IAS Size [MB]	0.036	0.067	0.26	0.406	1.088	1.791
Sponza	Optimization t[ms]	14.611	26.529	48.004	59.833	287.464	558.708
	Overlap	54.644	52.284	54.492	55.429	58.169	59.395
	Split/Merge	58/0	108/0	407/0	683/0	2921/0	3930/0
	GAS Build t[s]	130.316	140.413	202.548	264.644	659.853	856.791
	GAS Size [MB]	19.21	19.303	19.857	20.523	25.247	27.068
	IAS Build t[s]	21.223	19.86	20.828	20.921	19.46	20.01
	IAS Size [MB]	0.134	0.148	0.237	0.318	0.974	1.27
Powerplant	Optimization t[ms]	460.489	1614.45	4811.33	8272.43	29961	33136.8
	Overlap	8.509	11.017	20.902	19.477	10.756	9.513
	Split/Merge	10/0	32/0	233/0	791/0	5524/0	8780/0
	GAS Build t[s]	203.532	228.349	333.791	480.082	1688.01	2773.05
	GAS Size [MB]	892.695	892.784	893.55	895.272	901.87	906.412
	IAS Build t[s]	17.373	20.693	17.85	17.87	19.086	20.31
	IAS Size [MB]	0.021	0.027	0.086	0.25	1.639	2.595
San Miguel	Optimization t[ms]	716.276	1288.56	3313.51	3713.25	6325.15	76054.9
	Overlap	237.112	110.966	28.497	25.332	23.672	18.42
	Split/Merge	90/0	224/0	948/0	1608/0	6800/124	27592/5907
	GAS Build t[s]	1138.8	1072.8	1326.7	1481.1	2611.13	6857.55
	GAS Size [MB]	705.15	705.606	707.654	708.129	717.435	751.162
	IAS Build t[s]	18.245	18.392	18.827	19.394	20.602	24.733
	IAS Size [MB]	0.675	0.714	0.926	1.12	2.607	7.012

Table 5.3: Statistics of chosen skip threshold values.

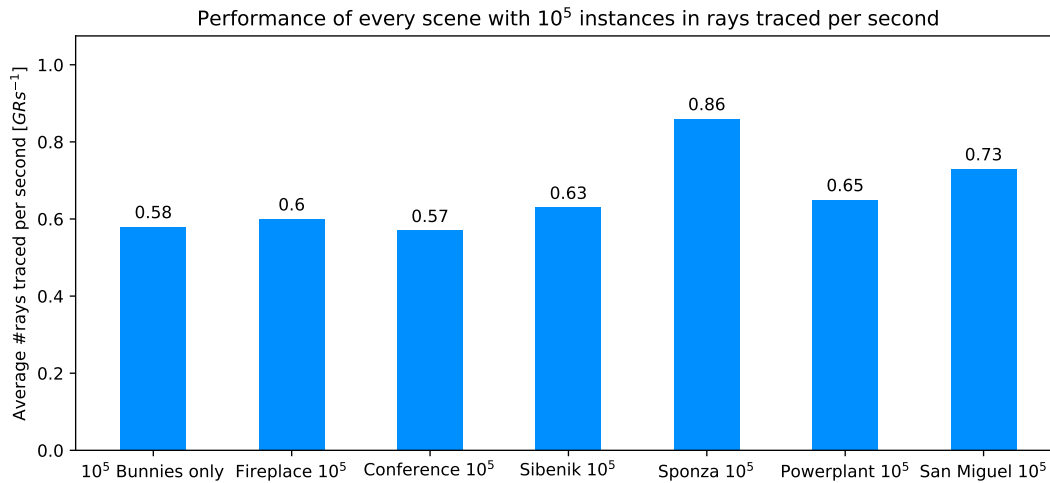


Figure 5.5: Traversal performance of every scene with 10^5 instances compared with instances only. All types of rays are counted. Higher value is better.

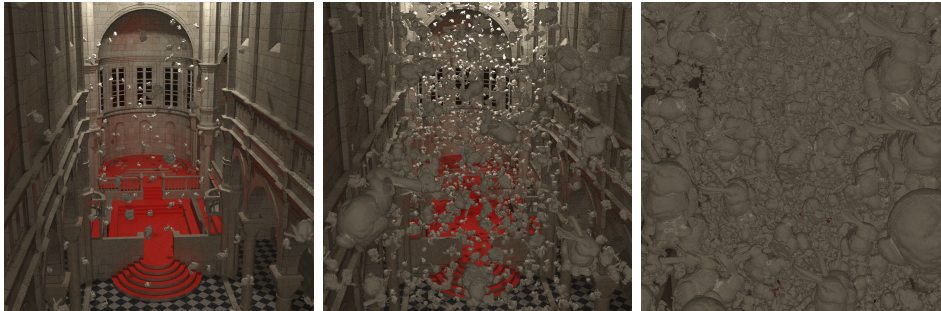


Figure 5.6: Sibenik scene with 10^3 instances of Stanford bunny model on the left, 10^4 instances in the middle, and 10^5 instances on the right

5.2 Scene optimization performance

Experiment 4: Performance comparison of implemented methods on chosen scene setups

In the final experiment we are comparing all implemented methods on selected scenes. Four methods are tested: Single GAS per scene, GAS per object, BVH with 10^{-3} , and Overlap reduction with Skip threshold set to 0.005 and SoM threshold set to 0.5. Scenes are tested with static geometry only, with 10^3 animated instances and 10^4 animated instances. In static Sponza scene was our method best among the tested methods. That was unexpected because the overlap ratio achieved with our method in the Sponza scene was always higher than the initial overlap. The best performance has a 58.1689 overlap ratio. That is approximately 4.5 times higher than the initial ratio of

12.8781. BVH method scored overlap ratio of 16.1623.

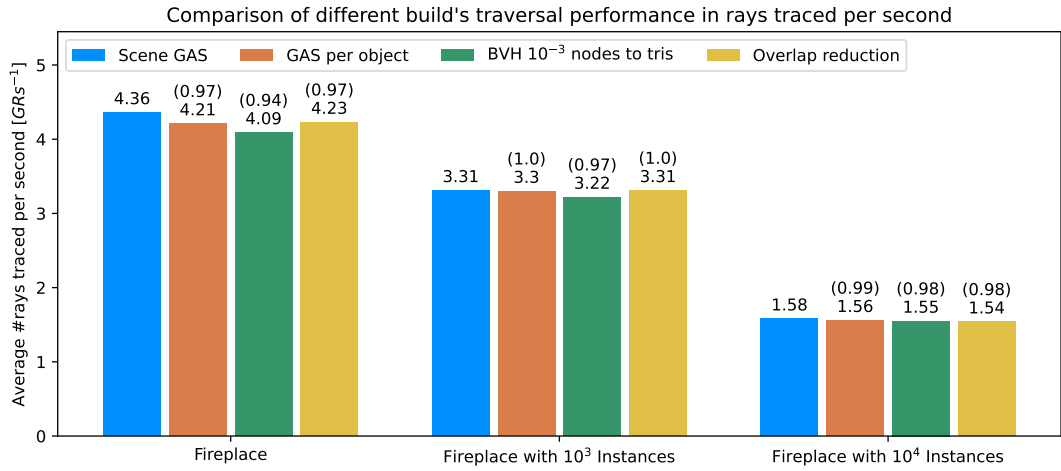


Figure 5.7: Traversal performance of all methods on Fireplace scene. All types of rays are counted. Higher value is better.

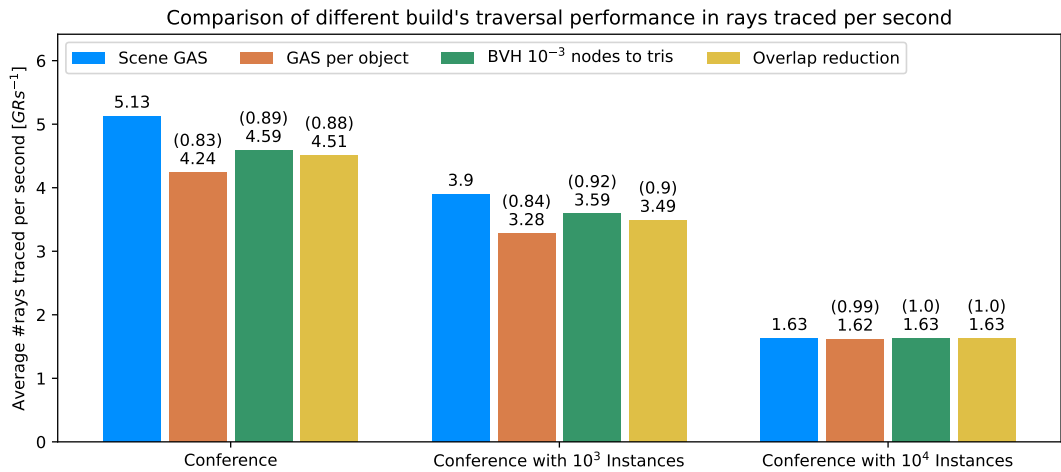


Figure 5.8: Traversal performance of all methods on Conference scene. All types of rays are counted. Higher value is better.

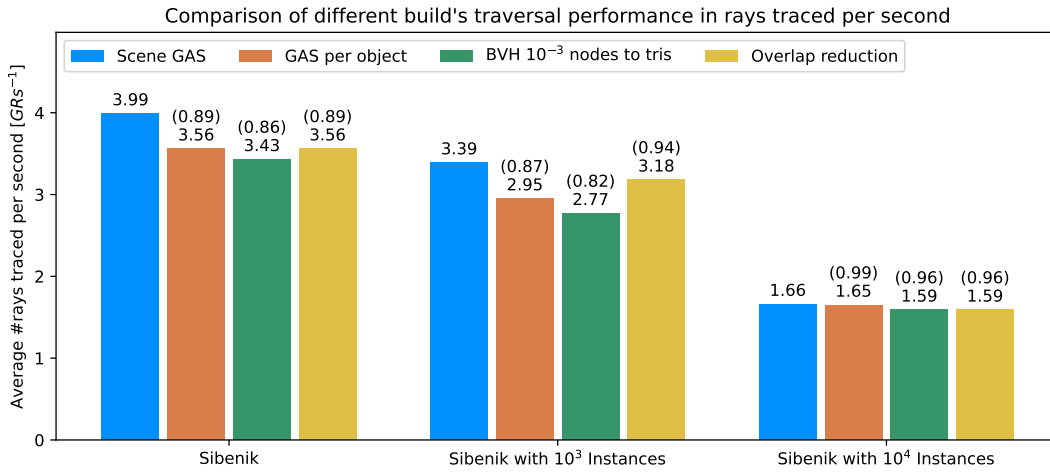


Figure 5.9: Traversal performance of all methods on Sibenik scene. All types of rays are counted. Higher value is better.

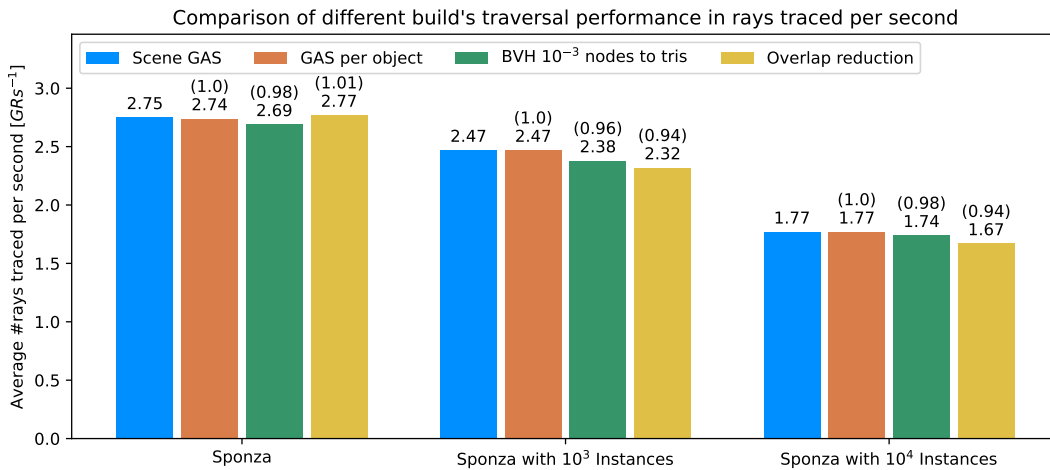


Figure 5.10: Traversal performance of all methods on Sponza scene. All types of rays are counted. Higher value is better.

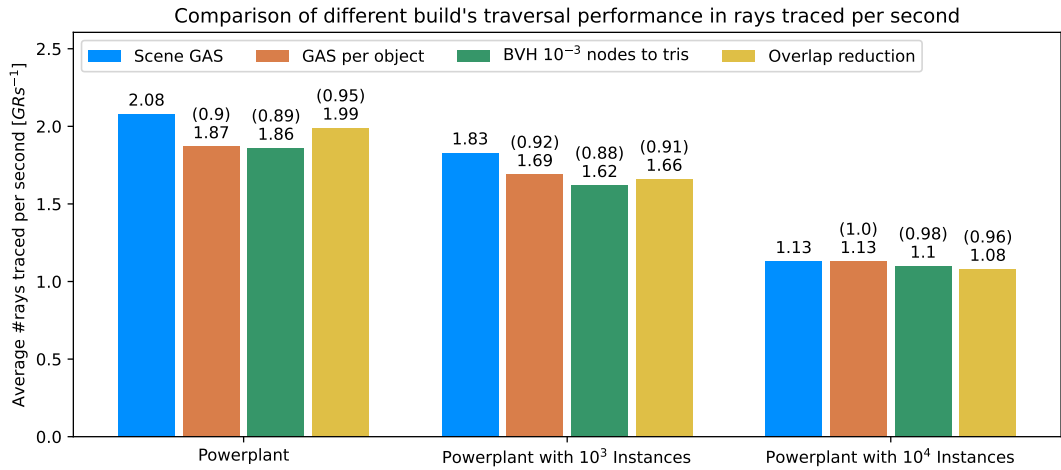


Figure 5.11: Traversal performance of all methods on Powerplant scene. All types of rays are counted. Higher value is better.

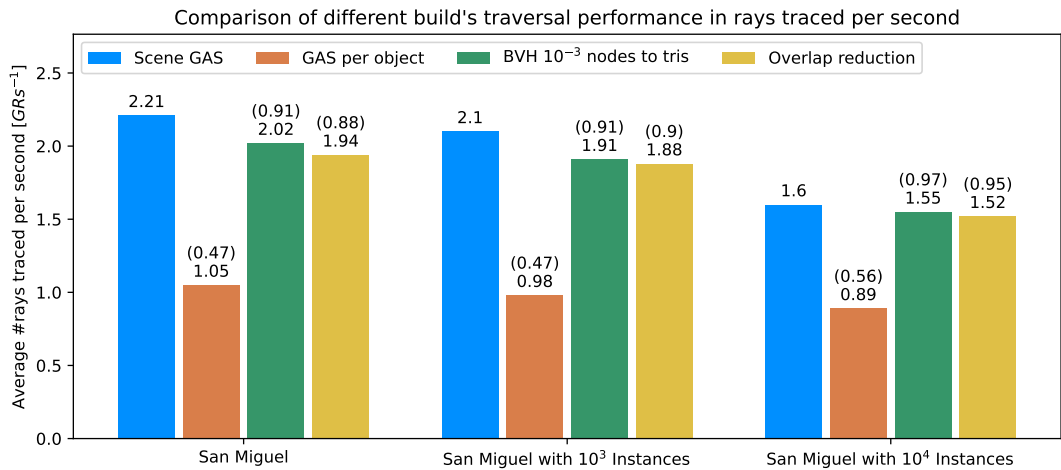


Figure 5.12: Traversal performance of all methods on San Miguel scene. All types of rays are counted. Higher value is better.



Chapter 6

Conclusion

We have yet to come up with the solution we hoped for, that is, pre-processing of the scene so that we would achieve better performance than building a single GAS for the entire scene. We have found valuable insights on the scene optimization. Creating structures that perform similarly to a single GAS per scene but could be used dynamically is a partial success. It must be noted that the reduction method still has flaws because of merging materials. That could be further explored. We could create mesh groups instead of merging meshes straightaway. Such postponed merging could also mean increasing the performance of the algorithm. Mesh groups could further be used as a single group for OptiX builder as a group of build inputs, or merged before calling OptiX API.

Appendix A

Bibliography

- [App68] APPEL, Arthur. Some Techniques for Shading Machine Renderings of Solids. In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. Atlantic City, New Jersey: Association for Computing Machinery, 1968, pp. 37–45. AFIPS '68 (Spring). ISBN 9781450378970. Available from DOI: [10.1145/1468075.1468082](https://doi.org/10.1145/1468075.1468082).
- [AK89] ARVO, James; KIRK, David. An introduction to ray tracing. In: Oxford, England: Morgan Kaufmann, 1989, chap. 6, pp. 201–262. The Morgan Kaufmann Series in Computer Graphics.
- [Ben+17] BENTHIN, Carsten; WOOP, Sven; WALD, Ingo; ÁFRA, Attila T. Improved two-level BVHs using partial re-braiding. In: *Proceedings of High Performance Graphics*. Los Angeles, California: Association for Computing Machinery, 2017. HPG '17. ISBN 9781450351010. Available from DOI: [10.1145/3105762.3105776](https://doi.org/10.1145/3105762.3105776).
- [Bik23] BIKKER, Jacco. *jbikker/bvh_article* [https://github.com/jbikker/bvh_article]. Github, 2023.
- [Dun19] DUNN, Alex. *Tips and Tricks: Ray Tracing Best Practices* [<https://developer.nvidia.com/blog/rtx-best-practices/>]. 2019. [Accessed 19-Jan-2023].
- [Dür25] DÜRER, Albrecht. *Underweysung der Messung, mit dem Zirckel und Richtscheyt, in Linien, Ebenen unnd gantzen corporen*. Nüremberg: Hieronymus Andreae, 1525.
- [Fuj+24] FUJITA, Syoyo et al. *tinyobjloader/tinyobjloader* [<https://github.com/tinyobjloader/tinyobjloader>]. Github, 2024.

- [GS87] GOLDSMITH, Jeffrey; SALMON, John. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications*. 1987, vol. 7, no. 5, pp. 14–20. Available from DOI: [10.1109/MCG.1987.276983](https://doi.org/10.1109/MCG.1987.276983).
- [Han89] HANRAHAN, Pat. An introduction to ray tracing. In: Oxford, England: Morgan Kaufmann, 1989, chap. 3, pp. 79–120. The Morgan Kaufmann Series in Computer Graphics.
- [Har19] HART, David. *OptiX Performance Tools and Tricks* [<https://on-demand.gputechconf.com/siggraph/2019/pdf/sig915-optix-performance-tools-tricks.pdf>]. 2019. [Accessed 19-Jan-2023].
- [Hen08] HENRIK. *Ray trace diagram* [https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg]. 2008. [Accessed 19-Jan-2023].
- [Hof90] HOFMANN, G. R. Who invented ray tracing? *The Visual Computer*. 1990, vol. 6, pp. 120–124.
- [KB23] KÁČERIK, Martin; BITTNER, Jiří. On Importance of Scene Structure for Hardware-Accelerated Ray Tracing. *Computer Science Research Notes*. 2023, vol. 3301, pp. 361–367. ISSN 2464-4617. Available from DOI: [10.24132/CSRN.3301.60](https://doi.org/10.24132/CSRN.3301.60).
- [KA13] KARRAS, Tero; AILA, Timo. Fast Parallel Construction of High-Quality Bounding Volume Hierarchies. In: *Proceedings of the 5th High-Performance Graphics Conference*. Anaheim, California: Association for Computing Machinery, 2013, pp. 89–99. HPG '13. ISBN 9781450321358. Available from DOI: [10.1145/2492045.2492055](https://doi.org/10.1145/2492045.2492055).
- [Kil+18] KILGARIFF, Emmett; MORETON, Henry; STAM, Nick; BELL, Brandon. *NVIDIA Turing Architecture In-Depth* [<https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth>]. 2018. [Accessed 19-Jan-2023].
- [Mei+21] MEISTER, Daniel et al. A Survey on Bounding Volume Hierarchies for Ray Tracing. *Computer Graphics Forum*. 2021, vol. 40, no. 2, pp. 683–712. Available from DOI: [10.1111/cgf.142662](https://doi.org/10.1111/cgf.142662).
- [MT97] MÖLLER, Tomas; TRUMBORE, Ben. Fast, Minimum Storage Ray-Triangle Intersection. *Journal of Graphics Tools*. 1997, vol. 2, no. 1, pp. 21–28. Available from DOI: [10.1080/10867651.1997.10487468](https://doi.org/10.1080/10867651.1997.10487468).
- [NVI22a] NVIDIA CORPORATION. *NVIDIA OptiX 7.6 - Programming Guide* [<https://raytracing-docs.nvidia.com/optix7/guide/index.html>]. 2022. [Accessed 19-Jan-2023].

- [NVI22b] NVIDIA CORPORATION. *NVIDIA OptiX Ray Tracing SDK Release Notes* [https://raytracing-docs.nvidia.com/optix7/release_notes/OptiX_Release_Notes_7.6_01.pdf]. 2022. [Accessed 19-Jan-2023].
- [Sjo22] SJOHOLM, Juha. *Best Practices for Using NVIDIA RTX Ray Tracing (Updated)* [<https://developer.nvidia.com/blog/best-practices-for-using-nvidia-rtx-ray-tracing-updated/>]. 2022. [Accessed 19-Jan-2023].
- [Wal20a] WALD, Ingo. *OWL - The Optix 7 Wrapper Library* [<https://owl-project.github.io/>]. 2020. [Accessed 19-Jan-2023].
- [Wal20b] WALD, Ingo. *OWL: A Node Graph "Wrapper" Library for OptiX 7* [<https://github.com/owl-project/owl/wiki>]. 2020. [Accessed 31-Dec-2023].
- [Wal+20] WALD, Ingo et al. Using Hardware Ray Transforms to Accelerate Ray/Primitive Intersections for Long, Thin Primitive Types. *Proc. ACM Comput. Graph. Interact. Tech.* 2020, vol. 3, no. 2, Art. No. 17, 1–16. Available from DOI: [10.1145/3406179](https://doi.org/10.1145/3406179).
- [Whi80] WHITTED, Turner. An Improved Illumination Model for Shaded Display. *Commun. ACM.* 1980, vol. 23, no. 6, pp. 343–349. ISSN 0001-0782. Available from DOI: [10.1145/358876.358882](https://doi.org/10.1145/358876.358882).