Evaluating BVH splitting strategies

Trung Tuan Le

A THESIS
In
Computer Graphics and Game Technology

Presented to the Faculties of the University of Pennsylvania in Partial
Fulfillment of the Requirements for the Degree of Master of Science in Engineering

2017

____________________________
Patrick Cozzi
Supervisor of Thesis

____________________________
Dr. Stephen Lane
Thesis Committee Chair
Abstract

Iterating through thousands of objects in a scene is expensive. By introducing a hierarchical spatial data structure, the time of traversing objects can be greatly reduced. Such a hierarchical spatial structure can be a grid, binary space partition tree, kd-tree, octree, or bounding volume hierarchy (BVH). One of the most commonly used spatial data structures in real-time ray tracing is the spatial split bounding volume hierarchy (SBVH). This thesis surveys the tradeoffs of using a SBVH in the context of ray tracing. The optimal case for using SBVH is with scenes that have a mixture between low poly and high poly meshes, or meshes that are not tightly fitted with axis-aligned bounding boxes. When the meshes have triangles with similar sizes, it is best to use the object split with surface area heuristics. When the scenes have uniformly distributed objects, equal counts partitioning is the most efficient choice. To achieve the real-time requirement, this thesis implements a hybrid rasterizer-raytracer in Vulkan.
Acknowledgement

I want to dedicate my acknowledgement to my advisor, Patrick Cozzi, to provide me with the support and freedom to explore my research, and Chris Hebert, from NVIDIA, for providing invaluable industry knowledge.

I also want say my thanks to Dr. Stephan Lane and Dr. Norman Badler for their support in my master’s program.
Outline

The thesis is structured into four parts.

**Part I** reviews the concept of ray tracing, and report on the current state of this rendering technique in modern applications. This thesis uses a custom hybrid rendering technique that combines rasterization and GPU ray tracing to achieve better visual fidelity while still maintaining speed for real-time applications.

**Part II**, the main part of this thesis, explores the use of spatial acceleration structures in the context of ray tracing. This part evaluates the different splitting strategies for partitioning a BVH.

**Part III** describes the architecture of the C++ Vulkan engine used for hybrid rendering. The implementation for the engine was built from the ground up except for third-party libraries for loading meshes and textures.

**Part IV** concludes the thesis with results and analysis.

**Appendix A** describes the glTF format for the scenes used for the evaluation.
# Table of Contents

Abstract .............................................................................................................................. 2

Acknowledgement ........................................................................................................... 2

Outline ............................................................................................................................... 3

Table of Contents ............................................................................................................. 4

**PART I** ......................................................................................................................... 6

1. Ray tracing overview .................................................................................................... 6
2. Hybrid rendering .......................................................................................................... 10

**PART II** ....................................................................................................................... 14

3. Acceleration structure ............................................................................................... 14
   3.1 Binary space partitioning (BSPs) ............................................................................. 15
   3.2 Kd-trees ................................................................................................................ 15
   3.3 Octrees .................................................................................................................. 19
   3.4 Bounding volume hierarchy (BVHs) ..................................................................... 20
      3.4.1 The ideal BVH ................................................................................................. 20
      3.4.2 Bounding volumes ........................................................................................ 22
   3.5 Splitting BVH ........................................................................................................ 24
      3.5.1 Early Split Clipping ........................................................................................ 24
      3.5.2 Edge volume heuristics ................................................................................ 26
   3.6 Spatial splits in bounding volume hierarchy (SBVHs) ........................................... 27
   3.7 BVH stack-less traversal ....................................................................................... 34
   3.8 Linear bounding volume hierarchy (LBVHs) ........................................................ 37
3.9 Bonsai BVH .................................................................................................................. 38

PART III .................................................................................................................................. 39

4. Vulkan .................................................................................................................................. 39

5. Implementation ..................................................................................................................... 48

PART IV .................................................................................................................................... 52

6. Evaluating spatial splits in BVH ......................................................................................... 52

6. 1 Analysis ............................................................................................................................. 63

7. Conclusions .......................................................................................................................... 68

Appendix A: glTF ...................................................................................................................... 70

References ................................................................................................................................ 72

Credits ...................................................................................................................................... 75
PART I

1. Ray tracing overview

While rasterization virtually dominates in real-time graphics, ray tracing excels at providing high quality image. This is because ray tracing can access the full scene information to solve for accurate shading of lights reflected or refracted off a surface, global illumination, shadows, and participating media to achieve realism.

The concept of ray tracing starts with ray casting, which begins by casting samples of rays from the eye through each pixel of the frame buffer into the three-dimensional scene. A ray can be defined mathematically with an origin and a direction. The problem is then reduced to finding the analytical solutions where this ray intersects with various types of geometry: triangles, spheres, cubes, NURBS, etc. An object can be composed of different geometries (commonly a set of triangles) with an associated material. The ray-geometry intersection then gathers the material information, computes lighting radiance based on the surface materials, normals, and tangents, and then returns the final color for a given pixel by averaging all the ray samples. To compute shadows, additional shadow feeler rays are cast from the surface intersection toward all the light sources to check for visibility.

Whitted extended ray casting into recursive ray tracing to account for reflection and refraction by generating secondary rays that bounce off the first intersection
of the surface based on Snell's law [Whitted80]. True reflection and refraction is not easily achievable through rasterization, but is a natural result of ray tracing.

Ray tracing is expensive for real-time application because it is image centric; the outer loop iterates through all the pixels, casting a ray through each pixel, and then the inner loop iterates through all the objects. The number of rays needed to compute for visibility per frame is:

\[
\text{Total number of rays} = \text{samples/pixel} \times \#\text{pixels} \times \#\text{rays/sample}
\]

\[
\text{Rays/sample} = \text{primary} + \# \text{of secondary rays}
\]

The key to improving ray tracing involves the following optimizations:

1. Fast ray-primitive intersection. Many rendering engines use triangles as the base primitive so no branching is required for checking primitive types. Fast ray-triangle intersection can be optimized by using the fast, minimum storage ray/triangle intersection by Möller and Trumbore [Möller97].

2. Reduce the number of rays needed for ray tracing. This could be done by reducing the number of rays per sample, or number of samples per pixel. Vertex tracing and adaptive sampling targets this. Amanatides also introduced cone tracing by changing the ray model from a line to a cone shape and perform cone-primitive intersection instead [Amanatides84]. This approach effectively covers tracing multiple pixels at the same time and hence reduce the total number of rays needed.

3. Fast scene traversal via acceleration structures. In the later sections, we explore the common ones. Another interesting optimization is to voxelize the scene space as a preprocess, then cache only the lights affecting the corresponding voxels. Then when testing for shadows, the feeler rays only need to be computed for the relevant lights.
4. Cache coherency by batching rays that have similar traveling patterns together. This is typical with rays that intersect the same materials, or share a similar origin and direction.

This thesis dives into the third optimization, focusing on evaluating bounding volume hierarchies as an acceleration structure for GPU ray tracing.

Related work

Ullmann introduced vertex tracing [Ullmann01]. The idea is to reduce the number of primary ray samples by only tracing rays toward vertices of visible triangles, computing the colors using recursive ray tracing, then interpolating the values in between with graphics hardware. Even though vertex tracing greatly reduces the number of rays to compute, it also results in a reduced image quality that could be unacceptable in today’s standard for real-time applications.

Instead of tracing several samples per pixel, we can trace at a subsample rate, and adaptively sample based on some heuristics, such as contrast and depth between neighboring pixels. The missing samples can be interpolated from their neighbors. This approach is described in details by Grassner[Grassner89].

Since ray samples from each pixel can be computed independently from each other, the task can be executed in parallel. The tasks can be divided by tiling the screen space into equal chunks, as each thread can carry out tracing independently.
Parallel ray tracing is efficient on the CPU, but for GPU ray tracing, another approach to divide workload more efficiently is wave front ray tracing [Laine13] describe in Figure 2:

![Wave front ray tracing process](image)

Figure 2. Wave front ray tracing processes rays in parallel for each level in the ray tracing hierarchy. From left to right: all primary rays are processed, then secondary rays, then the tertiary ray.

The GPU execution characteristics differ considerably from the CPU, based on the following factors: SMIT, and high-bandwidth, high-latency memory system. The GPU can process a large amount of data in parallel, but at the expense of a relatively long delay of memory access. Wave front ray tracing allows the GPU to minimize branch divergence and the kernel cache size to hide the memory latency.
2. Hybrid rendering

![Image of a scene rendered with hybrid rendering]

**Figure 3:** A scene rendered with hybrid rendering. The Cesium truck has texture mapping. The spheres in front of the truck alternate between reflective and refractive surfaces. The shadows are created from shadow feelers.

The hybrid rendering used in this thesis is inspired by PowerVR's rendering technique [Einig17]. Hybrid rendering works by first rasterizing images with deferred shading into the g-buffer. The g-buffer contains the information necessary to trace secondary rays for effects, therefore, combining the advantage of both techniques.
Deferred rendering has gained major popularity in real-time rendering. Some of its advantages are that it reduces the rendering algorithm complexity from $O(\text{numLights} \times \text{numObjects})$ to $O(\text{numLights} + \text{numObjects})$ by rendering a scene in two passes: it first renders the scene geometry into a g-buffer and then uses that g-buffer to calculate the scene lighting in a second pass. It is also easier to maintain since the Lighting stage is entirely disconnected from the Geometry stage. Deferred rendering is a rasterization technique, and it still has some limitations:

1. There is no direct support for refractive and reflective materials because the g-buffer can only retain information for one layer.
2. Shadows are typically computed using shadow mapping. This can introduce inaccurate shadows due to aliasing from low resolution shadow maps. Cascaded shadow mapping can mitigate these issues, but requires more memory for each cascaded level.

To combat the above limitation of deferred rendering, I implemented hybrid rendering. The basic concept is to first use rasterization through a deferred renderer to capture all objects in our scene and then apply a full-screen ray tracing pass by tracing rays initialized from the g-buffer information. In addition to the screen space data from the g-buffer, the ray tracing kernel also contains a BVH of the entire scene to be used for secondary ray tracing.
Figure 4: Image from Practical techniques for ray-tracing in games [Voica14]

There are only 3 layers needed in the g-buffer: position, normal, and material ID. In hybrid rendering, the first ray-triangle intersection has been precomputed by the g-buffer pass. For comparison, with an 800x800 resolution image, with a 1000 triangles scene without an acceleration structure, assuming we cast one ray per pixel, the first bounce generates:

\[ 800 \times 800 \times 1000 \text{ ray-triangle intersections} = 640,000,000 \text{ ray-triangle intersections} \]

Deferred shading eliminates this first bounce cost and transfers it onto the GPU rasterization pipeline.
Figure 4: Hybrid rendering combines deferred rendering with GPU ray tracing to enable reflection, refraction, and shadows.
PART II

3. Acceleration structure

Kay and Kajiya a hierarchical approach to reducing the time spent traversing complex scenes by employing a spatial data structure to reduce the number of intersection tests [Kay&Kajiya86] introduces. There are two main approaches: spatial subdivision (such as binary space partitioning, kd-trees, octrees, and regular grids) and bounding volume hierarchies. In this thesis, a combined structure is used, called a spatial bounding volume hierarchy [Stich09].
3.1 Binary space partitioning (BSPs)

In 1980, *Binary space partitioning (BSP)* was introduced by Fuchs, Kedem, and Naylor in the papers “Predetermining Visibility Priority in 3D scenes” and “On Visible Surface Generation by A Priori Tree Structures”. BSP recursively subdivides a space into convex sets using splitting planes [Fuchs80]. The BSP data structure was popularized in the Quake and Doom engines.

![Figure 5. A 2D BSP subdivision](image)

3.2 Kd-trees

Kd-trees are a special case of BSP trees, where the splitting planes are simply restricted to be perpendicular to one of the coordinate axes [Bentley75]. Kd-trees are highly efficient for fast tree traversal for ray tracing.
Kd-trees are constructed top down, like BSP trees, where each splitting plane divides the objects into two children nodes. There are infinite choices for picking the most ideal splitting plane. MacDonald and Booth [MacDo89] came up with surface area heuristics (SAH) for kdtrees. Ray tracing speed is constrained by the cost of ray-triangle intersection, so the more we can skip irrelevant triangles, the better. SAH uses the surface area of primitive's bounding box to compute to probability a ray will intersect that bounding box, and decide whether to split a BVH node at that point, or generate a leaf node. The cost of splitting a node along a plane is shown in Equation 1.

\[
C_{\text{total}} = C_{\text{traversal}} + C_{\text{intersection}} \frac{(N_A \cdot SA_A + N_B \cdot SA_B)}{SA}
\]

**Equation 1.** Cost of partitioning

Where:
• $C_{total}$ is the total cost of splitting
• $C_{traversal}$: cost of traversal.
• $C_{intersection}$: cost of ray-triangle intersection. Typically, 80 to 1 compare to the traversal cost. Their ratio is only what matters.
• $N_A$: number of triangles in child node A
• $N_B$: number of triangles in child node B
• $SA_A$: Surface area of child node A
• $SA_B$: Surface area of child node B
• $SA$: Surface area of the current node

Surface areas:
• Sphere: Area = $4\pi r^2$
• Cube: Area = $2 * (\text{width} \times \text{height} + \text{width} \times \text{depth} + \text{height} \times \text{depth})$

### 3.2.1 Binned SAH

Since exhaustively searching for the most optimal splitting plane is impractical, a common strategy is to use binning along the maximum extent of the bounding box to speed up the search. The following figure illustrates how binning works:
Figure 7. **Binned SAH. The primitive references are assigned to separate bins based on their centroid locations.**

The primitives are sorted into corresponding bins, also called buckets, based on where their centroids lie. A primitive reference is a pointer or an index to that primitive in the list of all primitives. Then for each splitting plane interleaving the bins, the splitting cost of left and right sides are computed by growing the bounding boxes of the primitive references within, and applying the equation.

Then, for each bucket, the cost of splitting plane is compared with the cost of ray-triangle intersections for all primitives in that node. If the cost of doing the intersection test is cheaper, this node is turned into a leaf node instead; otherwise, the node is split in half along the minimal cost bucket's centroid. As we will see later, SAH isn't just restricted to kd-trees, it can also be adapted for BVH construction.
3.3 Octrees

Octree is another form of BSP, where the space is subdivided equally into octants using three splitting planes at each level in the hierarchy. The root node is an AABB containing the entire scene, and each child node is the result of splitting equally along each axis. The tree is recursively subdivided until a depth limit or some other terminating conditions [Meagher&Donald80].

![Octree subdivision](https://www.math10.com/en/geometry/geogebra/geogebra.html)

**Figure 8.** Octree subdivision. Image created from <https://www.math10.com/en/geometry/geogebra/geogebra.html>

Due to their uniformity, octrees have the benefit of being simple to construct quickly. However, objects inside a scene often have irregular patterns and features that straddle across the octants, so it isn't optimal for ray tracing.
3.4 Bounding volume hierarchy (BVHs)

The *bounding volume hierarchy* (BVH) is the spatial data structure of choice in this thesis. It is relatively simple to construct, while providing good flexibility to adapt to a variety of ray tracing scenarios. Whereas kd-trees, octrees, or grids, can create disjoint nodes where a primitive can belong to multiple leaves, BVH guarantees that a primitive belongs uniquely to a single leaf. A spatial split BVH relaxes this restriction and combines the benefit of both kd-trees and BVHs. BVHs also have a quick build time compared to kd-trees, and can implement refitting logic for dynamic scenes.

3.4.1 The ideal BVH

When designing BVH, Klosowski suggested the following characteristics to consider [Klosowski98]:

1) The degree of branching factor, typically two- or four- fan out
2) Top-down versus bottom-up construction
3) Choice of bounding volume
4) Splitting rules, such as equal counts, binned SAH, spatial split

SAH splits when the cost of splitting is cheaper than cost inserting primitive into a leaf node. The ideal BVH should minimize:

1. # of ray/primitive intersections
2. # of ray/node intersections
To do so, the BVH should employ surface area heuristics partitioning to reduce the number of ray-intersections by half [MacDo89].

Compared to kd-trees, BVHs struggle with large overlapping triangles. Kd-trees can handle more complex scene configurations with overlapping triangles due to their flexibility in splitting the dimension to reduce the number of traversal steps. However, this means that if we relax the restriction of one node per triangle to allow for dynamic spatial splitting during BVH construction, the BVH can potentially overcome its limitation.
3.4.2 Bounding volumes

BVHs operate on the foundation of bounding volumes for a triangle or multiple triangles. Bounding volumes should tightly encapsulate the mesh or geometry primitives within it. The more fitted the bounding volume is, the less likely that the ray traversal cost is wasted. More complex bounding volumes can come with the cost of more expensive ray-volume intersection.

Some good criteria for bounding volumes are:

1. Convex and easy to test for a ray intersect.
2. Minimal volume that closely approximate the geometry of the enclosed object(s)
3. Efficient to store and compute

Linear volumes

The most common bounding volume for meshes is the axis aligned bounding box (AABB). AABBs are simple to construct and to compute ray intersections.

The bounding box is not restricted to align with the axes, as it can be any set of n-dimensional axes. Such bounding box is called oriented bounding box (OBB).

For more efficient collision detection, Klosowski
proposed k-dops. K-dops are convex polytopes (a n-dimensional geometry with flat sides) that are easily computed while simultaneously tighten the bounding volume [Klosowski98].

![Figure 9: From left to right: Axis aligned bounding box (AABB), oriented bounding box (OBB), and k-dop (where k = 8)](image)

**Quadratic volumes**

The bounding sphere is a simple but effective bounding volume. Its ray-intersection cost is cheap, just a ray-sphere equation, and it is quick to construct. Bounding spheres fit well for objects with round shapes.

The bounding capsule is an extension of the bounding sphere. It is common to use in games to enclose human shaped player avatar or other non-playable characters.

It has been demonstrated that tight fitting bounding volumes can provide better performance, such as rectangle swept spheres (RSS), over simpler bounding volumes choice like AABB [Larsen99].
3.5 Splitting BVH

When there are many large overlapping triangles, BVH can be bogged down by having to perform unnecessary traversal steps.

There has been research into eliminating these overlapping issues prior to the spatial split BVH. Ernst and Greiner demonstrated a pre-splitting technique called early split clipping [Ernst&Greiner07], and Dammertz and Keller investigated using edge volume heuristics [Dammertz&Keller08] as a pre-process step.

3.5.1 Early Split Clipping

In early split clipping (ESC) [Ernst&Greiner07] proposed a preprocess phase before BVH construction by splitting primitive bounding boxes to create new primitive references with tight bounding boxes. Each primitive is split recursively until its surface area is less than a maximum surface area threshold $SA_{\text{max}}$.

![Figure 10: ESC recursively pre-splits triangles at the median line, until its AABB's surface area is below the user-defined threshold. When constructing the BVH hierarchy, the newly split AABBs are used to grow the BVH nodes.](image)
ESC improves the performance of ray tracing overall. However, its efficiency is
dependent on a user-defined threshold for $S_{A_{\text{max}}}$ that is difficult to tune for generic
scenarios, making it not an ideal approach for splitting.
3.5.2 Edge volume heuristics

Another approach to pre-splitting triangles is edge volume heuristics [Dammertz&Keller08]. The idea is to recursively subdivide triangles along the longest edges until a threshold $\mu$ is met. At each subdivision, the volume of the largest AABB of the three AABBs computed for each of the edges is compared against this threshold $\mu$. Thus, this edge volume heuristic allows for dynamically split large triangles into smaller ones before the tree construction happens. The threshold $\mu$ is a ratio of the total scene AABB that can be adjusted with a user-defined $t$ parameter. The value of $t$ is highly dependent on the scenes, thus making EVH heavily reliant on the user's prior knowledge of the rendering scene.

![Figure 11: Recursively splitting a triangle based on edge volume heuristic](image)
3.6 Spatial splits in bounding volume hierarchy (SBVHs)

Both ESC and EVH explored the concept of splitting triangle primitives for BVH. The *spatial split BVH* (SBVH) augments traditional BVHs with similar inspiration from ESC and EVH by introducing the concept of dynamic triangle splitting during the construction phase [Stich09]. The ideal splitting plane is selected at each subdivision step based on binned SAH, as described in Section 3.2.1

Construction

At each subdivision level, SBVH decides to split, partition, or create leaf nodes based on the cost of each action. Constructing a SBVH builds the hierarchy from top down using primitive’s AABBs. The BVH partitioning uses binned SAH as in kd-trees, with primitives sorted to the left and right of the splitting plane based on their centroid’s locations relative to the plane. The partitioning for SBVH also uses binned SAH, but it splits primitive AABBs for straddling primitives to generate multiple tighter AABBs. The binned AABBs are then used to grow the node’s AABB.

![Figure 12: In binned SAH for SBVH, each straddling primitive is split at the bin's boundaries. The top two numbers keep tracking of the number of entering primitives and exiting primitives.](image)
When iterating through each splitting plane to compute the cost for spatial splitting, the entering and exiting primitive counts can be used for the cost equation.
Figure 13. The primitives are split at the bin's boundary and tighter AABBs are created. The final AABBs for the left node and right node are built from union of all split AABBs on the left and right side of the splitting plane.

To simplify implementation, a temporary array of PrimInfo struct in Listing 1 is used to store primitive information for both a full reference and fragments. This does not necessarily mean that a primitive is duplicated, it's just that two or more nodes can point to the same primitive.

```c
struct PrimInfo {
    PrimID primitiveID;
    AABB bbox;
}
```

Listing 1.

The PrimInfo is then used to construct leaf nodes when splitting is no longer beneficial.

The construction algorithm follows:
if primInfo is empty then
    return
end if
if primInfo.size < 4 then
    createLeaf(context)
    return
end if
if childAABB’s volume is empty then
    createLeaf(context)
    return
end if
loop each primInfo
    leafCost ← CalculateLeafCost(context)
    loop each bin
        objCost ← BestObjCost(context)
        if childBoxes overlap then
            spatialCost ← BestSpatialCost(context)
        end if
        if objCost is best then
            (left, right) ← PartitionObj(context)
        else if spatialCost is best then
            (left, right) ← PartitionObj(context)
        else
            createLeaf(context)
        end if
    end loop
end loop

Algorithm 1. SBVH construction algorithm [Fuetterling16]

SBVH space layout

SBVH construction requires temporary memory to store fragments (from triangle split) and primitive references (full triangle) created during splitting. Previously, the memory was packed tightly inside an array and shuffled to partition the references into left and right sides of the split. However, this has created complication with additional fragments due to triangle splits where child nodes can have larger memory than parent nodes. So, before construction starts, an
additional 20% memory of buffer space is allocated for the potential fragments. The available buffer space is assigned to the function stack in each recursive call using the first and last pointers. During the partitioning phase, the fragments of the left and right child sets are aligned to the upper and lower boundary and grow toward the center. Free space is reserved in between. Fragments are distributed recursively, applying to both spatial and object splits, until a leaf node is created. Once no free memory is available, the construction switches to using full object splits.

The advantage of this division scheme is that the application can determine ahead of time how much memory to allocate, and each local task has full access to all its available memory without affecting other tasks. For potential parallel SBVH construction, this is extremely beneficial for lock-free multithreading since tasks don’t need memory synchronization.

![Figure 14. The number on the right represents hierarchy levels. At each subdivision, the sets are then aligned to the upper and lower boundary of the buffer space and grow towards the center](image-url)
Some scenes can potentially generate a deep tree, so the number of spatial splits can be restricted with a split budget. It is a counter that decrements each time a spatial split happens. Once the budget is fully consumed, the construction switches to using full object splits.

In some performance tests, SBVH can save up to 20-30ms of traversal time relative to binned SAH, with the cost of increased memory. However, not all scenes benefit from SBVH, although SBVH never performs less than BVH. Tweaking the spatial split budget and limiting tree depth significantly affect performance.

Spatial SBVH splits vs. kd-tree splits

There are a few key differences between SBVH and kd-tree splits. Kd-tree splits can result in empty nodes, while in a BVH, there is at least a primitive contained in a node. Kd-tree splits can also switch for higher quality split by adaptively search for a better splitting plane to produce minimal cost, since the most optimal split is not guaranteed to be at one of the bin's boundaries. For SBVH, the partition simply falls back to the object split strategy, and does not attempt to further optimize for splitting position. This keeps the construction and designing of SBVH faster than kd-trees.

Restricting spatial splits

The advantage of spatial split is to reduce overlapping nodes, but it comes with a potentially higher memory consumption that takes away the benefit of BVHs compared to other acceleration structures. Therefore, we can examine the overlapping surface area of the children nodes against a threshold to restrict
spatial splitting in case that the overlapping surface area is not significant. We use the best object split to compute:

$$\lambda = SA(B_1 \cap B_2)$$

$$\frac{\lambda}{SA(B_{root})} > \alpha$$

**Equation 2.** Restricting spatial split condition [Hapala11]

$\lambda$ denotes the overlapping surface area of the children nodes $B_1$ and $B_2$. When the ratio of $\lambda$ with the root node's surface area is less than a user-defined threshold $\alpha$, the entire step of spatial split is skipped. The optimal value for $\alpha$ is $10^{-5}$ [Stich09].

**Unsplitting references**

Occasionally, it is undesirable to split a primitive if its overlapping area is significantly small that the cost of splitting can be higher than storing the entire primitive in one of the child nodes.

**Figure 14.1:** Ray 1 sorts A as near child and B as far child, while ray 2 sorts B as near child and A as far child [Hapala11]
With each straddling primitive, we can further compare the cost of splitting with the cost of storing it fully in one of the children nodes. For N straddling primitives, there are $3^N$ possible partitions, so SBVH employs simple conservative heuristics as in Equation 2 to compute the cost of each partitioning choice for each straddling primitive:

$$C_{\text{split}} = SA_1 * N_1 + SA_2 * N_2$$
$$C_1 = (SA_1 \cup SA_\Delta) * N_1 + SA_2 * (N_2 - 1)$$
$$C_2 = SA_1 * (N_1 - 1) + (SA_2 \cup SA_\Delta) * N_2$$

**Equation 3.** Cost of putting primitive entirely inside left or right child [Hapala11]

- $C_{\text{split}}$ denotes the cost of splitting.
- $C_1$ denotes the cost of putting the primitive entirely into child node 1.
- $C_2$ denotes the cost of putting the primitive entirely into child node 2.
- $SA_\Delta$ is the surface area of the primitive tested for unsplitting.

Here, the cheapest cost is used. If either $C_1$ or $C_2$ is less than $C_{\text{split}}$, then the primitive reference is put into either child node 1 or child node 2. By unsplitting primitives, the total cost of SAH is improved slightly.
3.7 BVH stack-less traversal

The ray tracing algorithm requires a traversal of the BVH tree. In a CPU, this traversal is typically done recursively on a stack, but for a GPU, the approach must either use a stackless or short stack algorithm. The BVH stackless traversal is used here [Hapala11].

The stackless traversal algorithm falls into three categories: a restart of traversal, links to traverse between siblings and parent-child, or exploiting regularity in the data structure. This thesis implemented a link strategy, where each node keeps an additional link back to its parent.

For this stack-less traversal to work, the BVH needs to be constructed with the following requirements:

1. Binary BVH tree with exactly two children nearChild and farChild, sorted based on distance from the ray origin. All primitives are stored at leaf nodes.
2. Each node has a pointer to parent.
3. Each inner node has a unique traversal for a given ray from nearChild to farChild. This order can be different for each ray but must be the same order for the same ray. The subtrees are sorted along the maximum extent axis of its children nodes.
4. Internal nodes only store a bounding box of all primitives in their sub-tree.
Figure 15: Ray 1 sorts A as near child and B as far child, while ray 2 sorts B as near child and A as far child [Hapala11]

With the parent’s pointers, now we can iterate through the BVH structure using a simple state logic. In the following diagrams for stackless traversal, C denotes current node, P denotes parent, N denotes near child, F denotes far child, double rings denote that a bounding volume intersection needs to be performed, dotted lines denote that last traversal step, and bold arrow lines denote possible next traversal steps. There are only three traversal states that a node can be entered from the current node:

1. FROM_CHILD:

Figure 16. In the FROM_CHILD case the current node was already tested when traversing down the tree, and does not have to be re-tested. The next node to traverse is either current’s sibling farChild (if current is nearChild), or its parent (if current was farChild [Hapala11]
2. FROM_SIBLING:

![Diagram of FROM_SIBLING case]

**Figure 17.** In the FROM_SIBLING case, we know that we are entering `farChild` since it cannot be reached in any other way, and that we are traversing this node for the first time (i.e. a bounding volume test must be performed). If the node is missed, we back-track to its `parent`; otherwise, the current node must be processed. If it is a leaf node, we perform ray-intersection with all its primitives, and proceed to parent. Otherwise, we enter current’s subtree by performing a FROM_PARENT step to current’s first child [Hapala11]

3. FROM_PARENT:

![Diagram of FROM_PARENT case]

**Figure 18.** In the FROM_PARENT case, we know that we are entering `nearChild` and we do the same as in the previous case, except that every time we would have gone to `parent` we go to `farChild` [Hapala11]

At each state, it can be determined where to transition for the next iteration. If it is an interior node, a ray/box intersection is performed to decide whether to continue in this subtree. If it is a leaf node, a ray/primitive intersection is performed instead and the point of intersection and its surface's normal are returned.
3.8 Linear bounding volume hierarchy (LBVHs)

*Linear bounding volume hierarchy* (LBVH) stores nodes in a linear array using a Morton code indexing scheme. Morton code puts nodes that are closer in their hierarchy also close in memory, which increases cache hits when traversing the tree [Lauterbach09].
3.9 Bonsai BVH

*Bonsai BVH* [Ganestam15] is a combination of constructing multiple subtrees concurrently via multiple threads with top down binned SAH partitioning (also referred to as sweep SAH), and then build up the hierarchy to the top nodes.

Before any construction happens, triangles are first grouped into sets based on the mid-points of their AABBs. The number of triangles for each set is kept relatively small. Once the sets are selected, each set can be given to an available thread to construct a "mini-tree". Once all the mini-trees are built using top-down sweep SAH, then the top nodes can be constructed. At the end of the construction, there is a "pruning" option to fix badly constructed mini-trees.

Bonsai takes advantage of parallel construction to reduce the build time for BVH.
PART III

4. Vulkan

For the full Vulkan specification, see footnote\(^1\). The Vulkan SDK \(^2\) used in this is from LunarG Developers.

Vulkan is a modern graphics API designed by Khronos. It is an explicit API for the graphics hardware and keeps the drivers work to the minimal. Because of this, Vulkan also provides high flexibility for programmers to optimize. The API is designed for the new generation of graphics hardware, for AR/VR, and for multithreading rendering.

Vulkan is designed for high performance with low overhead, so the API comes with little to no validation and debugging tools by default. Fortunately, as part of the SDK, optional layers can be enabled during development and turned off for final release. Vulkan also comes with different platform-specific extensions, and must be enabled individually. The \texttt{VK_KHR_swapchain}, for example, is needed for creating a platform-specific swap chain – a set of image buffers to be presented to the screen each frame.

The LunarG developers have provided us with a great SDK for Vulkan on Windows and Linux operation systems. The SDK provides runtime components, as well as

\footnote{1 \url{https://www.khronos.org/registry/vulkan/specs/1.0/html/vkspec.html}}
\footnote{2 \url{https://www.lunarg.com/}}
validation and debugging tools for Vulkan. It also comes with a SPIR-V compiler to translate GLSL code.

Before going into the implementation of the renderer created for this thesis, let's take a tour of the different components of Vulkan. This is a broad look at the surface of the API, with an attempt to explain the moving parts of our renderer without overwhelming the reader with pedantic details. For a complete and comprehensive description of Vulkan, please consult the Vulkan spec documentation. There is also an excellent Vulkan tutorial online by Alexander Overvoorde [Overvoorde17].

Instance, Devices, and Queues

Instance

Every Vulkan application starts with a Vulkan instance `VkInstance`. The instance enables the application to access the graphics hardware and exposes an interface through which to communicate with the host device.

Physical device

Vulkan exposes physical devices, or the graphics hardware, through `VkPhysicalDevice`. This could be a discrete GPU, an integrated GPU, or both. Each physical device has different specifications and limitations. The application can query this information and select the proper device for the task. For rendering on desktop, picking a discrete GPU, if available, is often sufficient.

The physical device handle can be used to access one or more queues with the `VkQueue` handle that can process work asynchronously. Queues are used to submit
commands with VkCommandBuffer to the device to allow specification of shader programs, kernels, and the data used by the kernels. Commands can also be used to affect states of the pipeline and allocate new resources.

Logical device

Whereas the physical device interface is used to query hardware information, queues, and various hardware limits, the actual communication between the application client and the host is done via a logical device. VkDevice is the object handle that creates this abstraction layer for the application. The device handle is needed when allocating resources, creating commands, or affecting states of the pipeline.

Surface & Swapchain

For a typical rendering application, the program starts by selecting the physical device appropriate for rendering, then creates a logical device handle to be used for the rest of the application lifetime. Then, the program begins with platform-specific code for opening a window surface and preparing the display device onto which to draw. Since accessing the window surface is platform-dependent, the GLFW library provides the excellent benefit for creating surface on both Windows and Linux. At the time of this writing, MacOS has not yet fully supported Vulkan.

Once the window surface is created, we need to instantiate a swap chain. A swap chain is the infrastructure to present images onto the screen and must be explicitly created in Vulkan. Essential, a swap chain is a queue of images to be rendered to and presented to the screen. The application first renders to an available swap chain image, present it, then while the first frame is presented, the application can
start rendering onto the second image in the chain as not to sit and wait idly. The process can continue for as many images as there are in the queue. Once the presented image is finished, it is put back into the end of the queue, and the process repeats.

Resources

Shaders need two types of primary resources: *images* and *buffers*. VkImage is used to store textures as sampled by the shaders. Its layout can also be transitioned to be linear or optimal tiling for memory. Access to VkImage is through a VkImageView, which specifies how to read the image layout in memory. Frame buffers for the swap chain are just a list of VkImages presented to the screen.

On the other hand, a VkBuffer is essentially a linear array of ray data that can be used to store data for uniforms, vertex buffers, or index buffers. Access to VkBuffer is through a VkBufferView.

Shaders access image views, buffer views, samplers, or combined image samplers through a set of bindings called *descriptor sets*. The VkDescriptorSet describes the resources for each binding. The bindings layout is specified with VkDescriptorSetLayout.

Pipeline

Vulkan exposes two types of pipelines: the *graphics pipeline* and the *compute pipeline*. The graphics pipeline uses the graphics queue to process its commands, while the compute pipeline uses the compute queue. The graphics queue also functions as a generic queue, and in fact can carry out both graphics and compute
commands. Before using a pipeline, the application must query the \texttt{VkQueue}'s support through the physical device interface.

The \texttt{VkPipeline} for graphics allow for specification at the following stages:

- Shader modules
- Vertex input assembly
- Tessellation
- Viewport / Scissor
- Color blending
- Multisampling
- Depth / Stencil
- Dynamic state
- Rasterization

Except for the dynamic state, all the other stages of the pipeline cannot be modified once the pipeline has been created.

The compute pipeline only allows for specification of the compute shader module.

The \texttt{VkPipelineLayout} can point to any \texttt{VkDescriptorSetLayout}, and the \texttt{VkDescriptorSet} can be updated via the command buffer. The data of the binding descriptor set at the time of queue submission is the one used for the shaders.

\textbf{Renderpass}

A render pass specifies the render targets for the draw call, where a \texttt{VkFramebuffer} contains the actual image.

Each render pass can contain multiple subpasses. The \texttt{VkSubpass} specifies which attachments for the render target, and multiple subpasses are connected via \texttt{VkSubpassDependency}.
For example, a render pass can contain the following subpasses:

Subpass 0 → Color attachment 1
    → Color attachment 2

Subpass 1 → Depth attachment
    → Color attachment 2

Subpass dependencies:
Subpass 0 → Subpass 1

Listing 2.

The first subpass 0 contains two color attachments, 1 and 2, while the subpass 1 contains a depth attachment and the same color attachment 2. The subpass dependency specifies that subpass 0 will render first, followed by subpass 1.
Command buffers
Command buffers are used to record commands to be submitted to the device queues. This design of the Vulkan model significantly diverges from OpenGL command submission, where the commands are implemented invoked during draw calls. In Vulkan, the commands are first recorded into the `VkCommandBuffer`, and then submitted to the appropriate queue (graphics, compute, or transfer). The queue submission is carried out asynchronously, so that no GPU bubble occurs. As a side effect for development in Vulkan, the application programmer needs to manage command dependencies explicitly via the synchronization objects provided by the API.

Synchronization
Synchronization of resource access and pipeline dependencies in Vulkan is the responsibility of the application programmer. Vulkan provides the following schemes for synchronization:

- **VkFence**: fences are used to synchronize tasks carried out on a queue. Fences have two states: signaled and unsignaled. A fence can be signaled as part of a queue submission, and other tasks depending on if the fence can wait for its state to change before executing.

- **VkSemaphore**: semaphores are used to control access to resources across multiple queues. Semaphores also have two states: signaled and unsignaled. The state can be signaled after the batch execution completed and waited for before the batch execution begins.

- **VkEvent**: events provide a fine-grained synchronization control for dependency between commands submitted to the same queue. Events also
has two states: signaled and unsignaled. The application can directly signal and unsignal a `VkEvent` without using the queue submission command.

- **Pipeline barriers:** pipeline barriers provide a synchronization for commands but at a single point, instead of through signaled and unsignaled states. The dependency between source stage and destination stage is specified explicitly through `VkPipelineStageFlags`.

- **Memory barriers:** memory barriers are used to control explicit access to resources, such as transferring memory between queues, transitioning image layouts, and defining availability and visibility operations.

**SPIR-V**

Standard Portable Immediate Representation (SPIR-V) is an open standard, cross-API intermediate language for shaders. Common shader language such as GLSL can be compiled down to SPIR-V and fetch to the graphics pipeline in Vulkan. Vulkan uses SPIR-V since its cross-API support enables compiling vendor-independent shaders into unified shader code.
Memory management

The big advantage of using Vulkan is the flexibility of controlling resource memory. There can be thousands of meshes within a given scene, so to achieve cache utilization and limit the amount of costly memory allocation, vertex indices and vertex attributes data can be packed tightly for each mesh into the same VkDeviceMemory allocation and the same VkBuffer, while leaving the uniform buffer in its own VkDeviceMemory since it’s updated every frame. This also helps reduce the initialization time to load each scene since we no longer have to create a new VkBuffer and allocate a new VkDeviceMemory for each attribute.

Instead of directly mapping memory from the host, a temporary buffer is used for staging and transferring the data from client memory onto device memory this way.

![Figure 19: vertex buffer memory layout][Hebert16]

In this layout scheme, we still need to partition based on each mesh data because when the meshes are extracted from a glTF 3D model, each of them have their unique buffer view that needs to be handled properly. It’s possible that we can just directly copy a glTF’s buffer view into VkDeviceMemory and offset the VkBuffer correctly from there [Hebert16].
5. Implementation

Our application is structured as followed:

Figure 20: The application can select different rendering engine based on a configuration at start. The scene class manages loading models and textures, and constructing the acceleration structure (BVH) once the meshes are loaded.

The application first opens a scene file to load all the models and textures into the acceleration structure. Once the scene loading completes, the rendering engine then initializes the Vulkan instance, device and queues. Once the hardware for rendering is ready, then we need to allocate the vertex buffers for the models and image buffers for textures. For hybrid rendering, the graphics pipelines for deferred shading frame buffer presentation are created, and the compute pipeline for ray tracing is created, along with their respective renderpass, descriptors, descriptor layouts, and synchronization objects.
The rendering pipeline is broken down into three stages: deferred, raytracing, and on screen output.

**Figure 21**: As you can see in the image above, the CPU first computed the BVH structure of our scene. The deferred pass is rendered off-screen, then passes the g-buffer to the compute shader to perform raytracing. The result is then taken from the image texture from compute shader and rendered on-screen.
Other features

Fast ray-primitive intersection

The ray-primitive intersection uses the fast, minimum storage ray/triangle intersection by [Möller97]. This method doesn't require computation on the fly or storing of the plane equation which increases memory savings for triangle meshes.

Screen space ambient occlusion (SSAO)

The engine implements SSAO as a post process to enhance shadows at creases. SSAO is computed in the fragment shader by first sampling random points in a hemisphere of each fragment. The depth values of these samples are compared with the fragment's depth to accumulate an occlusion factor. The occlusion is then applied with a 32x32 kernel to darken the points that are occluded. To minimize aliasing, a Gaussian blur phase is applied to smoothen the image.
Figure 22: SSAO is off. Notice the lambert softening at the jeans folding.

Figure 23: SSAO is on. With SSAO, the occlusion at the jeans folding is enhance, giving a more realistic look.
PART IV

6. Evaluating spatial splits in BVH

Due to its compact memory characteristics, and flexibility to adapt to a variety of scenes, SBVH was chosen for evaluation. The purpose of this survey is to determine the ideal situations for its usage.

To carry out the evaluation, a variety of scenes are used and compared between the naive BVH split (equal count on the median line), binned SAH, and spatial split SBVH.

The following scenes are used:
Scene 1: Large overlapping triangles

Figure 24: Large overlapping triangles. The reference is of radius 1.

Chart 1. Scene 1 build time in ms/frame

Chart 2. Scene 1 trace time in ms/frame
Scene 2: Small overlapping triangles

Figure 25: Small overlapping triangles. The reference is of radius 1.

Chart 3. Scene 2 build time in ms/frame

Chart 4. Scene 2 trace time in ms/frame
Scene 3: Overlapping triangles of various sizes in pairs

**Figure 26**: Overlapping triangles of various sizes in pairs

**Chart 5.** Scene 3 build time in ms/frame

**Chart 6.** Scene 3 trace time in ms/frame
Scene 4: Overlapping triangles and decreasing in sizes

**Figure 27:** Overlapping triangles and decreasing in sizes. The tree only grows on one side since all the primitives are distributed to right side of the splitting plane.

**Chart 7.** Scene 4 build time in ms/frame

**Chart 8.** Scene 4 trace time in ms/frame
Scene "Office":

**Figure 28:** The office has many objects with roughly the same sizes. The “Office” model is from alexatica at <https://clara.io/view/8e3f1876-c643-4b40-ae9f-0d9693c507b6>

**Chart 9.** Scene “Office” build time in ms/frame

**Chart 10.** Scene “Office” trace time in ms/frame
Scene "Ellie":

Figure 29: A highly detailed model of a human figure, with triangles of roughly equal sizes. The “Ellie” mode is from dpakc at <https://free3d.com/3d-model/the-last-of-us-ellie-18325.html>

Chart 11. Scene “Ellie” build time in ms/frame

Chart 12. Scene “Ellie” trace time in ms/frame
Scene "Bears":

![Bears Scene Image]

**Figure 39:** Low poly bear models uniformly placed on a ground plane. The “bear” model is courtesy of Charles Li Wang.

**Chart 13.** Scene “Bears” build time in ms/frame

**Chart 14.** Scene “Bears” trace time in ms/frame
Scene “Dragon Temple”:

**Figure 40:** A high detailed dragon model placed in the middle of a low poly temple.

**Chart 15.** Scene “Dragon Temple” build time in ms/frame

**Chart 16.** Scene “Dragon Temple” trace time in ms/frame
Scene “Pillars”:

**Figure 41:** All the pillars are tilted 45 degrees to simulate the worst-case scenario for AABBs.

**Chart 17.** Scene “Pillars” build time in ms/frame

**Chart 18.** Scene “Pillars” trace time in ms/frame
The build time node count for each scene is described in the table **Table 1:**

<table>
<thead>
<tr>
<th>Scene</th>
<th>Primitive count</th>
<th>Equal counts build time</th>
<th>Equal count node count</th>
<th>Binned SAH build time</th>
<th>Binned SAH node count</th>
<th>Spatial split build time</th>
<th>Spatial split node count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scene 1</td>
<td>16636</td>
<td>13818 ms</td>
<td>58695</td>
<td>19101.9 ms</td>
<td>49276</td>
<td>20194 ms</td>
<td>44030</td>
</tr>
<tr>
<td>Scene 2</td>
<td>16636</td>
<td>12659 ms</td>
<td>58657</td>
<td>15979 ms</td>
<td>48760</td>
<td>22456 ms</td>
<td>46474</td>
</tr>
<tr>
<td>Scene 3</td>
<td>264</td>
<td>0.59 ms</td>
<td>92</td>
<td>11.25 ms</td>
<td>100</td>
<td>15.04 ms</td>
<td>70</td>
</tr>
<tr>
<td>Scene 4</td>
<td>101</td>
<td>0.46 ms</td>
<td>215</td>
<td>10.4 ms</td>
<td>94</td>
<td>24.8 ms</td>
<td>188</td>
</tr>
<tr>
<td>Office</td>
<td>33502</td>
<td>87166 ms</td>
<td>111229</td>
<td>70771 ms</td>
<td>62928</td>
<td>61526 ms</td>
<td>47407</td>
</tr>
<tr>
<td>Ellie</td>
<td>31535</td>
<td>92884 ms</td>
<td>111571</td>
<td>64384 ms</td>
<td>81976</td>
<td>74336 ms</td>
<td>77395</td>
</tr>
<tr>
<td>Bears</td>
<td>29752</td>
<td>67299 ms</td>
<td>101767</td>
<td>53576 ms</td>
<td>78611</td>
<td>60353 ms</td>
<td>74014</td>
</tr>
<tr>
<td>Dragon temple</td>
<td>871630</td>
<td>5.97e6 ms</td>
<td>820129</td>
<td>2.69e6 ms</td>
<td>424720</td>
<td>3.41e6 ms</td>
<td>449957</td>
</tr>
<tr>
<td>Pillars</td>
<td>5292</td>
<td>1007.78 ms</td>
<td>16431</td>
<td>1369.4 ms</td>
<td>9946</td>
<td>2569.81 ms</td>
<td>9217</td>
</tr>
</tbody>
</table>

**Table 1.** Build time for a variety of scenes using equal count/median, binned SAH, and spatial split partitioning.

The trace time and node count for each scene is described in **Table 2:**

<table>
<thead>
<tr>
<th>Scene</th>
<th>Primitive count</th>
<th>Equal counts trace time</th>
<th>Binned SAH trace time</th>
<th>Spatial split trace time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scene 1</td>
<td>16636</td>
<td>17.53 ms</td>
<td>16.8 ms</td>
<td>16.98 ms</td>
</tr>
<tr>
<td>Scene 2</td>
<td>16636</td>
<td>15.92 ms</td>
<td>16.1 ms</td>
<td>15.05 ms</td>
</tr>
<tr>
<td>Scene 3</td>
<td>264</td>
<td>0.092 ms</td>
<td>0.109 ms</td>
<td>0.139 ms</td>
</tr>
<tr>
<td>Scene 4</td>
<td>101</td>
<td>0.46 ms</td>
<td>0.2 ms</td>
<td>0.9 ms</td>
</tr>
<tr>
<td>Office</td>
<td>33502</td>
<td>45.7 ms</td>
<td>31.3 ms</td>
<td>26 ms</td>
</tr>
<tr>
<td>Ellie</td>
<td>31535</td>
<td>8.67 ms</td>
<td>7.34 ms</td>
<td>13.48 ms</td>
</tr>
<tr>
<td>Bears</td>
<td>29752</td>
<td>20 ms</td>
<td>13 ms</td>
<td>5.14 ms</td>
</tr>
<tr>
<td>Dragon temple</td>
<td>871630</td>
<td>22 ms</td>
<td>9.31 ms</td>
<td>3.9 ms</td>
</tr>
<tr>
<td>Pillars</td>
<td>5292</td>
<td>50.54 ms</td>
<td>44.36 ms</td>
<td>37.2 ms</td>
</tr>
</tbody>
</table>

**Table 2.** Trace time for a variety of scenes using equal count/median, binned SAH, and spatial split partitioning.
6.1 Analysis

Best for binned SAH

In scenes that have triangles with roughly equal sizes but are not placed uniformly, such as the human model “Ellie”, Binned SAH surpasses both Equal Counts and SBVH.

Best for Equal Counts

“Scene 3” presents itself to be the best to split based on the medians. Even though the scene has triangles with varying sizes, the uniform placement of triangles of identical sizes in pairs make it efficient for constructing and tracing using Equal Counts, instead of the other two techniques.

Best for SBVH

Spatial splitting excels when there is a significant variation in term of triangle sizes and their placement overlap. For example, in the "Dragon Temple" scene, the dragon (871000 triangles) is placed in the middle a very low poly environment (about 300 triangles) that are much larger. The low poly triangles overlap with the tessellated dragon at the important top levels of the tree. By splitting up these large polygons, there is a significant boost in tracing time. With the environment taken away, there isn't any improvement with splitting so the tree converges to a full BVH. Similarly, the “Office” scene contains multiple small objects enclosed inside the large polygons that make up the office’s walls. This scene presents itself as a good candidate for spatial splitting. Notice that the SBVH might appear to be faster at build time and generate fewer nodes, but it is because the depth limit and split budget restricted the SBVH build to branch further.
Another scenario where a spatial splitting excels is with scenes where the geometries aren't axis aligned. In scene “Pillars”, all the pillar geometries are tilted 45 degrees to present the worst case for BVH. This is because the bounding volumes are axis aligned and not object aligned. With spatial, we can clip these large overlapping surface areas into tighter bounding volumes, which significantly brings the tracing speed from 50ms/frame to 37ms/frame. Without using spatial splitting, the bounding volumes should be object oriented bounding boxes or k-dops bounding boxes to reduce the traversal steps.

Restricting spatial split

The threshold to restrict spatial splitting as in Equation 2. is $10^{-5}$, suggested by the authors of SBVH [Stich09]. To investigate, I used the “Pillars” scene, which has performed well with SBVH. First, the number of bins is fixed at 15 since this is also the best choice this scene. The test varies the threshold alpha value from full SBVH to full Binned SAH. The result in Chart 20 shows an interesting drop in trace time just by simply switching back to Binned SAH.
Chart 19. The number of spatial splits allowed. At alpha = 1e-6, it is full SBVH, and alpha = 1, it's Binned SAH.

Chart 20. The trace time remains consistent, but has a sudden spike when it switches to Binned SAH.
Number of bins for SAH

The “Pillars” scene is also used to test for varying the number of bins for SAH.

**Chart 20.** The build time is roughly consistent and starts to increase when bin’s count = 19

**Chart 21.** The trace time shows a very interesting valley, where the most optimal number of bins is in the range 13-18, with everything outside drops significantly in performance
This shows that there is a limit when using more bins for chopping primitives could saturate due to high number of fragments that potentially increases ray-intersection time.

Limit depth

The maximum recursion depth is suggested to be $\sim 8 + 1.3\log(N)$ in PBRT [Pharr, Jakob&Humphreys16].

The maximum depth of the tree should be at the minimum $\log(N)$ for a balanced tree, but then not too deep as to prevent pathological cases of stack overflow.
7. Conclusions

Picking the right splitting strategies is important to improve the ray tracing performance and reduce the unnecessary overhead of constructing and storing the BVH in memory, especially when working with GPU ray tracing. As our evaluation demonstrated for BVH, each scene has the best and worst option: Equal Counts is good for uniform placing of objects with the overlapping objects having roughly equal sizes, Binned SAH works best with also roughly equal sized objects, but can tackle non-uniform placement of the objects, and SBVH excels at mixed scenes with varying object sizes that overlap each other.

Future work

Dynamic scenes will require updating the BVH structure per frame due to object transformation. The BVH generation or refitting overhead can become expensive. One could potentially design and implement a GPU refitting or BVH construction, such as Bonsai [Ganestam15]. Then the application can take advantage of Vulkan’s multithreading capability for concurrent queue submission to the GPU kernel.
Figure 42: With Vulkan, the scene generation step can be processed concurrently while the graphics pipeline renders
Appendix A: glTF

This thesis uses the glTF format for all models to sample data necessary to evaluate SBVH.

The GL transmission format (glTF) is a vendor- and runtime-neutral asset delivery format. The standard is designed by Khronos Group to bridge the gap between 3D content creation tools and modern GL applications by providing a flexible, lightweight, and interoperable format for 3D contents.

With the advent of WebGL and OpenGL ES, more than ever that assets are delivered via the web and process by the GPU. However, traditional 3D formats such as OBJ or COLLADA were designed for offline applications and lack the ability for streaming and processing large files.

For desktop applications seeking high performance, 3D assets are typically converted into proprietary format optimized for fast loading time. However, this creates a fragmented market of incompatible formats that can't work across multiple applications.

The glTF solves these problems by having an API-neutral standard designed for future graphics applications across different domain. For this reason, glTF couples well with the design philosophy of Vulkan targeting modern graphics hardware, including AR/VR.
glTF uses the JSON format with external data to describe complete 3D scenes with scene hierarchy, meshes, camera, materials, animation data, etc. The format mirrors closely with GPU API so that mesh buffers can be loaded directly into GPU as a simple buffer copy. Other external information such as textures and animation data can be stored as independent files and referenced via URIs inside the gltf file. The implementation of this thesis supports glTF 1.0 and loading physically based materials specified with glTF 2.0.

For the full specification of glTF 1.0, see details in [glTF16].
References


[glTF16] "glTF 1.0 specification". ULR:
<https://github.com/KhronosGroup/gltF/tree/master/specification/1.0>


[Hebert16] Hebert C. 2016, "Vulkan Memory Management". URL:
<https://developer.nvidia.com/vulkan-memory-management>


[Wald04] Ingo Wald PhD thesis. URL: <http://www.sci.utah.edu/~wald/PhD/wald_phd.pdf>


Credits

The “Bear” model used in this thesis is the courtesy of Charles Li Wang.

The glTF sample models from <https://github.com/KhronosGroup/gltf-Sample-Models>.

The “Ellie” model is from dpakc at <https://free3d.com/3d-model/the-last-of-us-ellie-18325.html>.

The “Office” model is from alexatica at <https://clara.io/view/8e3f1876-c643-4b40-aef9-0d9693c507b6>