Memory Efficient GPU-Accelerated Ray Tracing Using Reduced-Footprint Bounding Volume Hierarchies

Hank Lewis

June 8, 2024

[COMP3391] IS: Comp. Raytracing Approaches
Dr. Daniel Stevenson

Ritchie School of Engineering & Computer Science
University of Denver

Contents

Memory Efficient GPU-Accelerated Ray Tracing Using Reduced-Footprint Bounding Volume Hierarchies

Abstract

This paper discusses the performance and space cost of bounding volume hierarchies (BVH) on the GPU in the context of ray tracing. Its accompanying codebase, `RTRS`, is hosted on GitHub (`https://github.com/hankotanks/rt_rs`).

Using GPU timestamp queries, different 'intersection handlers' are benchmarked. The reduced-footprint BVH handler, which represents internal and leaf nodes as tagged unions, is found to have the best performance as the complexity of scene geometry increases.

1 Premise

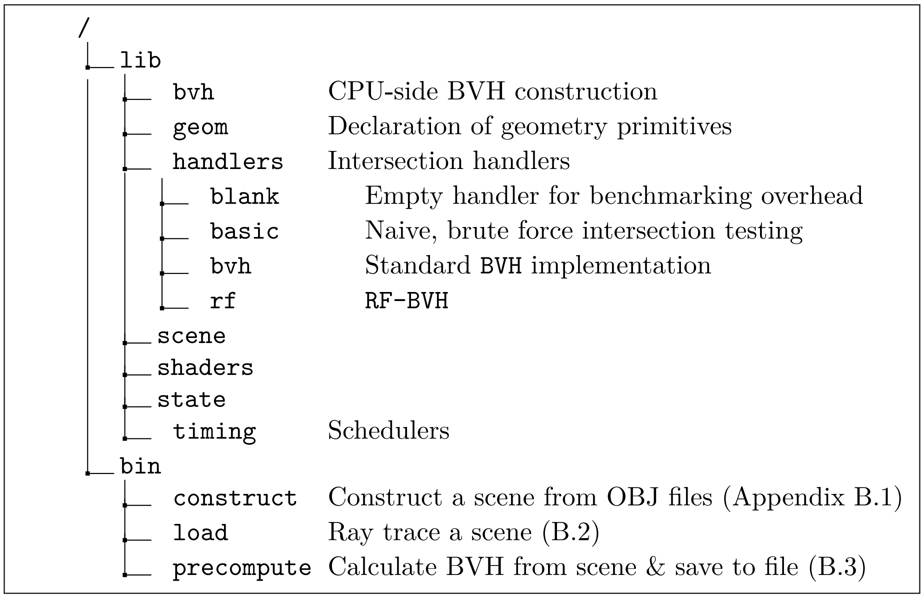
This project is an exploration of memory efficiency in the context of GPU ray tracing. Every realtime ray tracer is bounded by two principle performance problems: concurrency and the cost of intersection tests. The first could be addressed with multithreading, but this only kicks the can down the road—no commercially-available CPU has sufficient cores to simultaneously resolve even a low-resolution image.[1]

A more suitable solution is GPU-acceleration. Not only does the GPU have a far larger core count, but these cores are defined spatially. In the associated codebase, this spatial arrangement of cores is given direct correspondence to the pixels of the produced image. In this manner, each instance of the GPU's compute pass deals with a single pixel, once. This approach eliminates the need for batching, shared mutability, and other concerns that slow and complicate a CPU-based approach.

The cost of intersection tests is a trickier problem to solve. In the same way that programming languages center around primitives—floats, integers, characters, etc—ray tracers deal with geometrical primitives. A collection of these primitives—triangle, spheres, etc—define a *scene*. To ray trace a scene, we must implement a method of finding these primitives efficiently in 3D space.

In this project's ray tracer implementation (called RTRS), this method is called an intersection handler (`IntrsHandler`). At its most basic, an intersection handler may, for every ray, iterate through all of a scene's constituent primitives, testing each for intersection. This approach has an $O(n)$ runtime—it linearly increases with the number of primitives. This naive approach quickly becomes unfeasible. In the following sections, I will disucss two handlers that

[1] While multithreading absolutely improves performance, each thread still must handle a large number of rays.



```
/
└─ lib
   ├─ bvh          CPU-side BVH construction
   ├─ geom         Declaration of geometry primitives
   ├─ handlers     Intersection handlers
   │  ├─ blank        Empty handler for benchmarking overhead
   │  ├─ basic        Naive, brute force intersection testing
   │  ├─ bvh          Standard BVH implementation
   │  └─ rf           RF-BVH
   ├─ scene
   ├─ shaders
   ├─ state
   └─ timing       Schedulers
└─ bin
   ├─ construct    Construct a scene from OBJ files (Appendix B.1)
   ├─ load         Ray trace a scene (B.2)
   └─ precompute   Calculate BVH from scene & save to file (B.3)
```

Figure 1: Module structure of RTRS

drastically reduce the number of intersection tests that need to be performed. Both are based on Bounding Volume Hierarchies, a type of spatial partition tree. The `BVH` handler contains a basic implementation; the `RF-BVH` (reduced-footprint bounding volume hierarchy) improves upon the former by compressing its representation in memory.

2 Project Architecture

RTRS is a **R**ay **T**racer written in **R**u**S**t with the help of WGPU, a backend-agnostic graphics framework. The ray tracer itself is implemented as a library; the crate also contains a number of binaries that faciliate scene generation (Figure 1).

2.1 Scene & Camera

A scene contains spatial data the describes a 3D environment (Listing 1). Although it can be created inline, the `construct` (Appendix B.1) binary allows users to create a scene directly from the command line by positioning object meshes in the world. The resultant scene is then serialized into JSON, and can be run using the `load` tool (B.2).

```
struct Scene {
  camera: CameraUniform,
  camera_controller: CameraController,
  prims: Vec<geom::Prim>,
  vertices: Vec<geom::PrimVertex>,
  lights: Vec<geom::Light>,
  materials: Vec<geom::PrimMat>,
}
```

Listing 1: Layout of the `Scene` struct

RTRS features two camera controllers (`crate::scene::CameraController`). The first is fixed, and will never update its position. The second is an orbital camera, which circles a target point at a fixed radius along the XZ plane. Each scene owns its camera; they are serialized alongside its geometry and lighting (Appendix B.1).

```
struct CameraUniform {
    pos: [f32; 3],
    at: [f32; 3],
}
```

Listing 2: `crate::scene::CameraUniform`

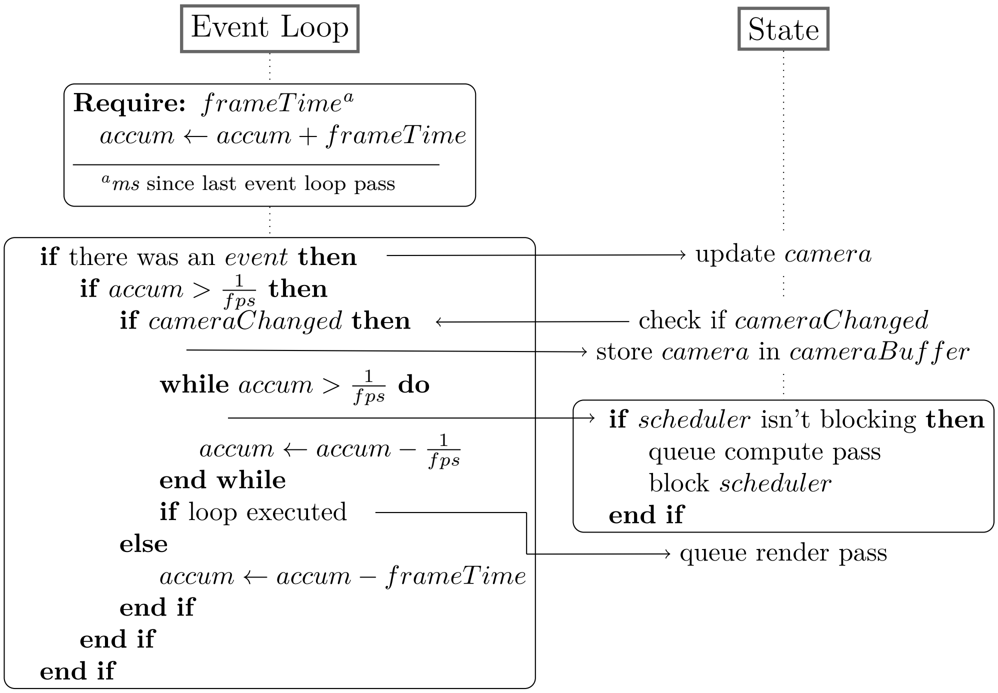
Since ray tracing occurs at a fixed framerate, multiple events may accumulate between renders. For this reason, the `CameraUniform` and `CameraController` are distinct (Listing 2). With each pass of the event loop, user inputs are given to the camera controller, which accumulates changes. Before rendering, these changes are applied to the associated uniform and copied into a buffer (which is later accessed in the compute shader).

2.2 The Event Loop

The ray tracer's operation is best understood through the interaction of its event loop (`winit::EventLoop`) and state (`crate::State`). At its core, the former has 3 main responsibilities: tracking frame length, handling events, and dispatching shader execution (Figure 2). The last two tasks are mediated through the `State` of a loaded scene.

Each event loop pass begins with the accumulation of the previous frame's duration. Then, if a user input resulted in a change to the scene's camera position, the ray tracer updates the texture and submits a draw call to the window. In the event that a ray tracing call exceeds the desired frame rate, the `Scheduler` prevents it from being submitted to the queue (Section 2.3). If multiple ray tracing passes are completable within the current frame—the opposite case—they are repeatedly queued until the display has 'caught up.'

Ray tracing is expensive; the event loop skips frames where the camera doesn't change. When using the fixed camera, only a single frame is rendered. Active cameras only trigger updates when the camera position has changed.



```
Require: $frameTime$[a]
  $accum \leftarrow accum + frameTime$
```

[a] ms since last event loop pass

```
if there was an $event$ then
    if $accum > \frac{1}{fps}$ then
        if $cameraChanged$ then
            while $accum > \frac{1}{fps}$ do
                $accum \leftarrow accum - \frac{1}{fps}$
            end while
            if loop executed
        else
            $accum \leftarrow accum - frameTime$
        end if
    end if
end if
```

```
update $camera$
check if $cameraChanged$
store $camera$ in $cameraBuffer$
if $scheduler$ isn't blocking then
    queue compute pass
    block $scheduler$
end if
queue render pass
```

Figure 2: State interaction within the event loop

2.3 Schedulers

Conceptually, a `Scheduler` is a write blocker. Whenever, `State::update` is called, the scheduler is queried via `Scheduler::ready`. If the result is truthy, a compute pass is enqueued.

Given the asynchronous nature of CPU-GPU communication, there is no direct way to determine if a compute pass has resolved. WGPU provides a callback function, `Queue::on_submitted_work_done`, which nearly fits our requirements. However, it is incompatible with WebGPU and doesn't distinguish between compute and render passes.

```
procedure STATE::UPDATE
    if Scheduler::ready then
        encoder ← wgpu::CommandEncoder
        computePass ← encoder.begin_compute_pass
        Bind buffer to computePass
        Add copy(buffer, bufferRead) operation to encoder
        Submit encoder to queue
        Asynchronously map bufferRead
            When it succeeds, Scheduler is ready
            Unmap bufferRead
    end if
end procedure
```

Figure 3: `State::update` logic from the `Scheduler`'s perspective

Buffers are not able to be mapped for read operations when they are utilized in ongoing GPU operations. This quality can be leveraged to poll the state of the queue without overhead (Figure 3). A tiny, 8-byte[2] buffer is bound to each compute pass alongside its configuration options (`crate::ComputeConfig`). An equally-sized, mappable buffer is created alongside it. Before submitting the compute pass's associated encoder, a copy operation is added. The first buffer—bound to the compute pass—is the source; the copy desination is the second, mappable buffer. Since the compute pass was enqueued first, it ties up the copy operation until its completion. If the copy desination is mappable, then a new pass can be scheduled.

2.3.1 Benchmarking

While the primary goal of the `Scheduler` is to time compute passes, its position in the rendering pipeline makes it the ideal place to perform benchmarks.

`Scheduler` is implemented as a trait, meaning that its behavior can be easily swapped out. RTRS implements two schedulers: `DefaultScheduler` and `BenchScheduler` (both found within `crate::timing`). The first implements

[2] The size of this buffer is given by `wgpu::MAP_ALIGNMENT`. This value provides the smallest mappable buffer size.

basic frame timing control; the second extends it to enable compute pass benchmarking.

Put simply, a query set (`wgpu::QuerySet`) is used in place of the default scheduler's timing buffer. Instead of discarding the results of the copy operation, the data from the query set is read from the desination buffer. The resulting timestamps, corresponding to the beginning and end of the compute pass, are then added to a dataset.[3] Just like the default scheduler, the completion of this copy operation signals the completion of the compute pass.

3 The Bounding Volume Hierarchy

A BVH is a spatial data structure represented by a tree of bounding boxes. Each node contains its extent (given by mimimum and maximum points), its children, and a set of geometrical primitives. Unlike a binary space partition (BSP), these primitives are not cleaved across the partition boundary. Instead, primitives are sorted into nodes based on their centroids; the bounds of these nodes are then resized to fit their contents.

While cleaving the triangles marginally decreases the number of required intersection tests, it also increases the amount of scene primitives that need to be passed to the GPU. The value of this tradeoff is not analyzed in this study, but may deserve further investigation.

While implementing a bounding volume hierarchy drastically reduces intersection tests, it introduces its own problems. Notably, a naive BVH implementation has a large memory footprint. Given the partitioning algorithm implemented in Section 3.1, leaf nodes contain a variable number of primitives.

With an average, per-leaf primitive count α and n primitives, the corresponding BVH tree will contain roughly $\left\lceil \frac{2n}{a} - 1 \right\rceil$ nodes.[4] The linear relationship between the total node count and the number of scene primitives demands careful consideration of the BVH's memory layout on the GPU.

This section describes the construction of a binary BVH on the CPU, as well as two methods of 'packing' data for use in RTRS's compute shaders—a naive, memory-inefficient approach (Section 3.2.1) and another with a reduced memory footprint (3.2.2).

3.1 Recursive Construction

A scene's BVH is recursively constructed on the CPU (Figure 6). The result is a tree where each node contains its bounds (minima and maxima), its children, and a list of indices to its contained primitives. Scene geometry is passed to the GPU separately.

```
struct Aabb {
```

[3] By default, this datset is exported as a line chart at regular intervals (and at program completion). It can be found in the crate's root directory as `benchmark.png`.

[4] All BVHs constructed in this project are binary. A different implementation may allow each node to have a variable number of children, drastically changing the total node count.

```
    fst: OnceCell<Box<Aabb>>,
    snd: OnceCell<Box<Aabb>>,
    minima: [f32; 3],
    maxima: [f32; 3],
    items: Vec<u32>,
}
```

Listing 3: Axis-Aligned Bounding Box (`crate::bvh::Aabb`)

Given a root node containing all scene geometry, `Aabb::split` recursively redistributes the bounding box's contents across child nodes until subdivision is complete. A node is designated as a leaf when its number of constitutent primitives is less than an established maximum, determined by the handler.

3.1.1 The Algorithm

What follows is a generalized description of the BVH construction algorithm used by RTRS. Procedure names have been preserved so that their corresponding functions in the Rust codebase can be easily found.

First, the extent of the constitutent geometry is calculated by recursing over each member and performing component-wise min/max tests (Figure 4).

```
procedure EXTREMA($tris$) → ($min, max$)
    $min \leftarrow \vec{\infty}$
    $max \leftarrow -\vec{\infty}$
    $i \leftarrow 0$
    while $i < \overline{\overline{tris}}$ do                ▷ $\overline{\overline{tris}}$ : number of primitives
        $min = \wedge(min, tris_i)$                  ▷ $\wedge$ : Component-wise minimum
        $max = \vee(max, tris_i)$                    ▷ $\vee$ : Maximum
        $i \leftarrow i + 1$
    end while
    yield $min, max$
end procedure
```

Figure 4: Calculate Extrema Given a Subset of Scene Geometry (`crate::bvh::Bounds::extrema`)

Next, each primitive is sorted into one of two subsets based on the position of its centroid (Figure 5). This partition is performed along the bounding box's largest axis, which combats degeneration by keeping the length of each axis within an order of magnitude of the others.

```
procedure SPLIT($min, max, tris$) → ($tris_a, tris_b$)
    $\vec{d} \leftarrow max - min$
    $min_a, max_a \leftarrow min, max$
    $min_b, max_b \leftarrow min, max$
    if $\vec{d_x} > \vec{d_y}$ and $\vec{d_x} > \vec{d_z}$ then
        $max_{ax} \leftarrow min_x + \vec{d_x} * 0.5$
        $min_{bx} \leftarrow max_{ax}$
    else if $\vec{d_y} > \vec{d_x}$ and $\vec{d_y} > \vec{d_z}$ then
        $max_{ay} \leftarrow min_y + \vec{d_y} * 0.5$
        $min_{by} \leftarrow max_{ay}$
    else
        $max_{az} \leftarrow min_z + \vec{d_z} * 0.5$
        $min_{bz} \leftarrow max_{az}$
    end if
    $tris_a, tris_b \leftarrow \emptyset$
    $i \leftarrow 0$
    while $i < \overline{\overline{tris}}$ do
        if $min_a, max_a$ contains centroid of $tris_i$ then
            $tris_a \leftarrow tris_a \cup \{tris_i\}$
        else
            $tris_b \leftarrow tris_b \cup \{tris_i\}$
        end if
    end while
    yield $tris_a, tris_b$
end procedure
```

Figure 5: Sort Geometry Along Maximal Partition (`crate::bvh::Aabb::split`)

If either of the resulting subsets is empty, a leaf node has been reached. A new node containing the non-empty subset is created and returned. Otherwise, partitioning continues. `Aabb::split` is performed on each subset; the resulting child nodes are appended to an empty parent node. When children are appended to a node, its own list of primitives is cleared—geometry is only assigned to leaves.

```
$tris_{scene} \leftarrow \{...\}$                                    ▷ Set of all triangles in the scene
procedure NEW($tris$) $\rightarrow node$
    $tris \leftarrow \{...\}$ where $\overline{\overline{\overline{tris}}} \subseteq tris_{scene}$
    $min, max \leftarrow bspExtrema(tris)$                                    ▷ Figure 4
    $tris_a, tris_b = bspPartition(min, max, tris)$                           ▷ Figure 5
    Construct $node$ with $(min, max)$ bounds
    if $tris_a$ or $tris_b$ are $\emptyset$ then
        $node$ is a leaf; subdivision stops
        Add $tris$ to $node$
    else
        Assign children $bspSplit(tris_a)$ and $bspSplit(tris_b)$ to $node$
    end if
    yield $node$
end procedure
$root \leftarrow bvhSplit(tris_{scene})$                              ▷ Recursively construct the BSP
```

Figure 6: Recursively construct the BVH (`crate::bvh::Aabb::new`)

Note that the approach detailed in this section limits mutability to the `new` procedure (Figure 6). This preserves the purity of `split`, but does not completely match the RTRS implementation.

3.2 Packing for the GPU

Shader languages—notably WGSL—cannot represent cyclical data structures due to the GPU's inherent structural limitations. The tree detailed in Section 3.1 must be flattened before being placed into a buffer.

Intuitively, the BSP can be represented as a flat array of bounding boxes. The fst and snd members of the CPU-side Aabb structure (Listing 3), representing the node's children, can be substituted for indices within a storage array.

The first approach taken involves directly translating the fields from the CPU-side BVH structure to a the aforementioned index-based approach. The second, reduced-footprint approach introduces the concept of untagged unions to the problem, drastically reducing the memory footprint at the expense of scene size and bounding box precision.

3.2.1 Direct Field Conversion

The CPU representation of the axis-aligned bounding box contains an arbitrary number of triangle indices. On the GPU, the equivalent structure must have a fixed size. Instead of maintaining a list of primitives within the bounding box itself, scene primitives can be ordered such that the triangles contained within each leaf are adjacent. Listing 4 details the result of these changes.[5] Notably, `item_idx` and `item_count` track the starting index of the node's triangles and their number, respectively.

```
struct Aabb {
  fst: u32,
  snd: u32,
  minima: vec3<f32>,
  item_idx: u32,
  maxima: vec3<f32>,
  item_count: u32,
}
```

Listing 4: Axis-Aligned Bounding Box (WGSL)

Introducing this level of indirection results in a fixed-size, shader-ready BVH in the form a storage array with mutually referencing element. The BVH tree is recursively flattened to produce an array of bounding box uniforms that can be passed to the GPU. Each element in the array occupies 40 bytes.

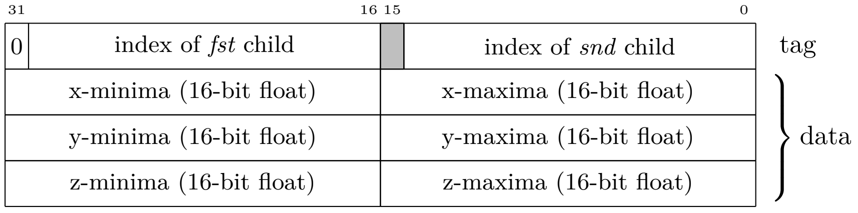
3.2.2 Reduced-Footprint Approach

Unions can be introduced to reduce the memory footprint of the BVH tree on the GPU. The same fields can be used to represent either the extent of a node or its contained primitives.

```
struct RfAabb {
  data: vec3<u32>,
  tag: u32,
}
```

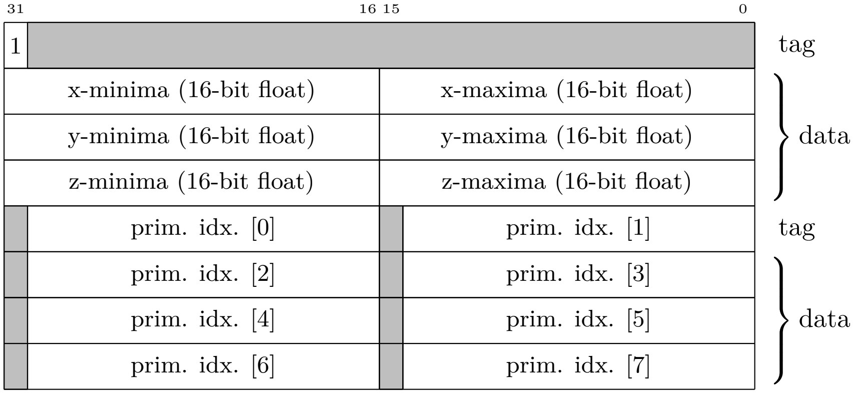
Listing 5: Reduced-Footprint Axis-Aligned Bounding Box (WGSL)

The resulting structure occupies only 16 bytes (Listing 5). Non-leaf nodes store both their minima and maxima within the `data` member. Float-precision is reduced from 32 to 16 bits, then each coordinate's minimum and maximum values are packed into a single 32-bit value. This value is then unpacked on the GPU using WGSL's `unpack2x16float` built-in.



[5] Fields are specifically ordered to prevent alignment issues. In WGSL, 3-dimensional vectors have a size of 12 bytes, but an alignment cadence of 16.

Leaf nodes are represented by two `RfAabb` objects, which are memory-adjacent in the GPU's storage array. The first contains the bounds in the same format as a non-leaf node, while the second contains the indices of the leaf's primitives. If the leaf node contains less than 8 primitives, then all remaining indices are set to 0 and skipped during intersection testing.



31 … 16	15 … 0	
1 ▨	▨	tag
x-minima (16-bit float)	x-maxima (16-bit float)	data
y-minima (16-bit float)	y-maxima (16-bit float)	
z-minima (16-bit float)	z-maxima (16-bit float)	
▨ prim. idx. [0]	▨ prim. idx. [1]	tag
▨ prim. idx. [2]	▨ prim. idx. [3]	data
▨ prim. idx. [4]	▨ prim. idx. [5]	
▨ prim. idx. [6]	▨ prim. idx. [7]	

Node data is retrieved using bitwise functions and the aforementioned WGSL `unpack2x16float` . Listing 6 provides a series of reference functions to extract data from an `RfAabb` object.

```
fn rfaabb_bounds(idx: u32) -> Bounds {
  let bounds = transpose(mat3x2<f32>(
    unpack2x16float(uniforms[idx].data.x),
    unpack2x16float(uniforms[idx].data.y),
    unpack2x16float(uniforms[idx].data.z),
  ));

  return Bounds(bounds[0], bounds[1]);
}

fn rfaabb_children(idx: u32) -> array<u32, 2> {
  let tag = uniforms[idx].tag;
  if((tag >> 31) & 1 == 1) { return array<u32, 2>(0); }

  return array<u32, 2>(
    tag & 0x7FFF,
    (tag >> 16) & 0x7FFF
  );
}

fn rfaabb_items(idx: u32) -> array<u32, 8> {
  if((uniforms[idx].tag >> 31 & 1) == 0) {
    return array<u32, 8>(0);
  }

  let node = uniforms[idx + 1u];
  return array<u32, 8>(
    (node.tag & 0x7FFF),
    (node.tag >> 16) & 0x7FFF,
    (node.data.x & 0xFFFF),
    (node.data.x >> 16) & 0xFFFF,
    (node.data.y & 0xFFFF),
    (node.data.y >> 16) & 0xFFFF,
    (node.data.z & 0xFFFF),
    (node.data.z >> 16) & 0xFFFF,
  );
}
```

Listing 6: `RfAabb` Data Extraction (WGSL)

Compressing the data of the BVH nodes comes at a small cost. Dropping the size of the integers imposes restrictions on scene size. When performing direct field conversion (Section 3.2.1), child nodes are stored as 32-bit unsigned integers. Field compression and the need for a tag byte at the start of the node reduces the size of this same reference to a 15-bit integer.

This drops the number of representable primitives to $2^{15} - 1$, which in turn limits the maximum number of BVH nodes to $\left\lfloor \frac{2^{15}}{3} \right\rfloor$.[6]

3.3 Intersection Testing

Once the BVH has been successfully packed and sent to the GPU, it is traversed and tested for per-pixel intersections (Figure 7). This process is implemented as a depth-first search (DFS) using a stack (Listing 7). The `SIZE` of the stack is based on the number of primitives in the scene and is baked into the shader during its compilation .

In addition to supporting basic stack ADT operations, a boolean flag is set when popping the final element.

```
var<private> stack: array<u32, SIZE>;

fn pop(
  idx: ptr<function, u32>,
  empty: ptr<function, bool>,
) -> u32 {
    if(*idx == 1u) { *empty = true; }
```

[6] Given a binary tree with double-leaf nodes, there are $3n - 1$ nodes for every primitive. This limits the node count to $2^{15} - 1 = 3n - 1$, where n is the maximum number of BVH nodes. This results in a hard limit of $\left\lfloor \frac{2^{15}}{3} \right\rfloor$ nodes. However, since each node can store up to 8 primitives, this number is practically much higher.

```
    *idx = *idx - 1u;
    return stack[*idx];
}

fn push(idx: ptr<function, u32>, bb: u32) {
    stack[*idx] = bb;
    *idx = *idx + 1u;
}
```

Listing 7: Stack Implementation for DFS (WGSL)

Traversal begins with the root node of the BVH, which is always the first element of the node array. It concludes when the stack is depleted. If the corresponding node's bounding box is a leaf, its primitives are tested for intersection (Figure 3.3.2). Otherwise, its children are pushed to the stack. When the stack is exhausted, the closest intersection has been found.

```
procedure TRAVERSE($ray$) → $intrs$
    $idx \leftarrow 0$
    $empty \leftarrow$ FALSE
    push($\&idx, 0$)
    $intrs \leftarrow$ blank intersection target
    while !$empty$ do
        $boxIdx \leftarrow$ pop($\&idx, \&empty$)
        $box \leftarrow$ uniforms[$boxIdx$]
        if collides($box, ray$) then                          ▷ Section 3.3.1
            if $box$ is a leaf then
                $boxIntrs \leftarrow$ blank intersection target
                for $tri$ in $box$ do
                    $temp \leftarrow$ intersection($ray, tri$)         ▷ Section 3.3.2
                    if $temp_t < intrs_t$ then
                        $boxIntrs \leftarrow temp$
                    end if
                end for
                if $boxIntrs_t < intrs_t$ then   ▷ $t$ : distance from $ray$ origin
                    $intrs = boxIntrs$
                end if
            else
                push($\&idx, box_{fst}$)
                push($\&idx, box_{snd}$)
            end if
        end if
    end while
    yield $intrs$
end procedure
```

Figure 7: Generalized Traversal of BVH

3.3.1 Testing AABB Intersection

RTRS implements the slab method for AABB intersection tests (Listing 8). Each plane of the bounding box is checked against the ray—if clipped by any of the 6 planes, an intersection has occurred. The incoming ray is checked against the bounding box's 6 planes. If any of them clip the ray, an intersection has occured. No distance checking is performed in the `collides` method of the compute shader.

```
const EPS: f32 = 0.000002;

fn collides(bb: Aabb, ray: Ray) -> bool {
    let min = bb.bounds.min;
    let max = bb.bounds.max;
    var t0 = (min.x - EPS - ray.origin.x) / ray.dir.x;
    var t1 = (max.x + EPS - ray.origin.x) / ray.dir.x;

    var t_min = min(t0, t1);
    var t_max = max(t0, t1);

    t0 = (min.y - EPS - ray.origin.y) / ray.dir.y;
    t1 = (max.y + EPS - ray.origin.y) / ray.dir.y;

    t_min = max(t_min, min(min(t0, t1), INF_NEG));
    t_max = min(t_max, max(max(t0, t1), INF_POS));

    t0 = (min.z - EPS - ray.origin.z) / ray.dir.z;
    t1 = (max.z + EPS - ray.origin.z) / ray.dir.z;

    t_min = max(t_min, min(min(t0, t1), INF_NEG));
    t_max = min(t_max, max(max(t0, t1), INF_POS));

    return (t_min < t_max);
}
```

Listing 8: Implementation of Slab Method for AABB Collision Testing (WGSL)

When the incoming ray is near-parallel with one of the bounding box's planes, false negatives can occur due to floating point imprecision. To account for this case, a small 'wobble' is added to each plane, guaranteeing a hit. 0.000002 was chosen based on experiential results, and may result in false positives. However, this slight increase in the number of intersection tests is necessary to prevent gaps in the output texture.

3.3.2 Testing Primitives for Intersection

In the current implementation of RTRS, triangles are the only supported primitive. For this reason, the Möller-Trumbore algorithm can be used for all intersection tests (Listing 9). This test accounts for edge cases, including when the incoming ray runs parallel to the triangle's plane.

```
fn intersection(r: Ray, tri: Prim) -> Intrs {
    let a: vec3<f32> = vertices[tri.a];
    let b: vec3<f32> = vertices[tri.b];
    let c: vec3<f32> = vertices[tri.c];

    let e1: vec3<f32> = b.pos - a.pos;
    let e2: vec3<f32> = c.pos - a.pos;

    let p: vec3<f32> = cross(r.dir, e2);
    let t: vec3<f32> = r.origin - a.pos;
    let q: vec3<f32> = cross(t, e1);

    let det = dot(e1, p);

    var u: f32 = 0.0;
    var v: f32 = 0.0;
    if(det > EPS) {
        u = dot(t, p); v = dot(r.dir, q);
        if(u < 0.0 || u > det ||
           v < 0.0 || u + v > det) {
          return intrs_empty();
        }
    } else if(det < -1.0 * EPS) {
        u = dot(t, p); v = dot(r.dir, q);
        if(u > 0.0 || u < det ||
           v > 0.0 || u + v < det) {
          return intrs_empty();
        }
    } else {
        return intrs_empty();
    }

    let w: f32 = dot(e2, q) / det;

    if(w > T_MAX || w < T_MIN) {
        return intrs_empty();
    } else {
        return Intrs(tri, w);
    }
}
```

Listing 9: Implementation of Slab Method for AABB Collision Testing (WGSL)

T_MIN and T_MAX denote the near and far clipping panes of the render, respectively. The EPS value here is distinct from the epsilon value used in collides (Listing 8). In this case, it corresponds to the minimum representable difference between two adjacent floating point values. This value is machine-specific and should be specified in crate::ComputeConfig.

4 Results

4.1 Memory-Efficiency

The space requirements of a scene vary depending on the specified intersection handler crate::handlers::IntrsHandler. The naive (BasicIntrs) and blank (BlankIntrs) handlers do not require additional space, because they rely exclusively on the scene geometry. `BvhIntrs` and `RfBvhIntrs` require access to the precomputed BVH structure.

On average, the reduced-footprint BVH handler maintained parity with the size of the scene geometry, while the simpler BVH construction took $\sim 370\%$ of the space (Figure 8).

Scene	`default`		`teatime`	
`BlankIntrs`	$680 + 0$[a]	0%[b]	$101,160 + 0$	0%
`BasicIntrs`	$680 + 0$	0%	$101,160 + 0$	0%
`BvhIntrs`	$680 + 2,448$	360%	$101,160 + 381,840$	377.46%
`RfBvhIntrs`	$680 + 656$	96.47%	$101,160 + 110,528$	109.26%

[a] $bytes_{prim} + bytes_{intrs}$

[b] Intersection handler's size compared to the size of the scene's geometry ($bytes_{intrs}/bytes_{prim}$).

Figure 8: Scene Space Requirements on the GPU

4.2 Performance

The performance of the two BVH intersection handlers was tested with 4 bounces at a resolution of $384x288$ with a workgroup size of 16. All tests were completed on a Gigabyte Radeon AORUS RX580.

4.2.1 Overhead

The bulk of the compute pass is spent performing intersection tests. When running the two demo scenes (`default` and `teatime`), the `crate::handlers::BlankIntrs` handler had a average runtime of $\sim 0.024ms$.

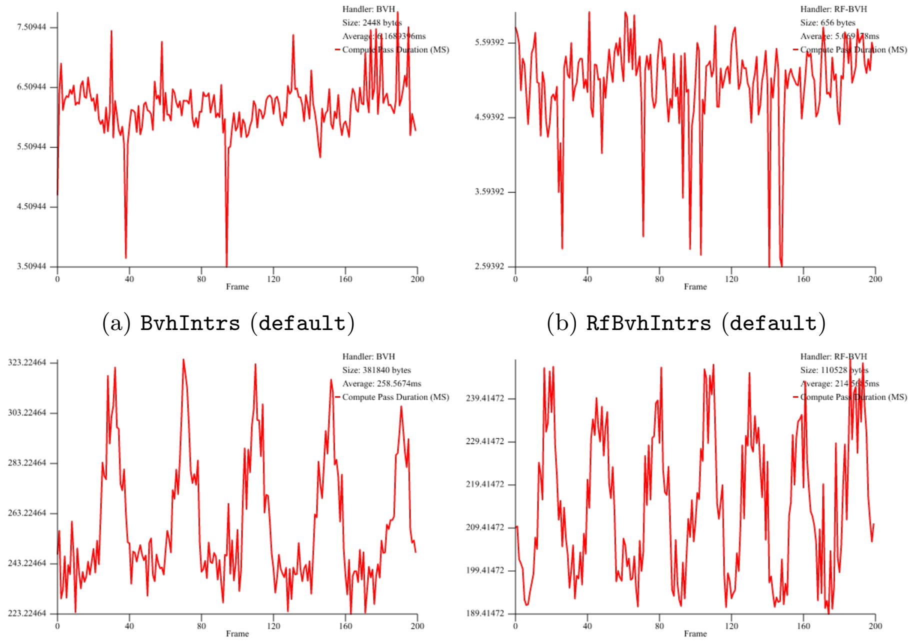
The charts in the following section have not been corrected for this overhead given its minor significance.

4.2.2 BVH vs. RF-BVH

Each handler/scene combination was benchmarked across 200 frames, which accounts for 5 complete rotations around the scene. Spikes in the performance can be attributed to camera angles in which more primitives were tested (Figure 9).

	`default`	`teatime`
`BvhIntrs`	6.168	258.567
`RfBvhIntrs`	5.069	214.569
Pct. Diff.	19.56%	18.59%

In addition to having a vastly smaller memory footprint, the `RfBvhIntrs` also performed better on the target hardware. Across the two scenes tested (`default` and `teatime`), the reduced-footprint intersection handler was roughly



(a) `BvhIntrs` (`default`)

(b) `RfBvhIntrs` (`default`)

(c) `BvhIntrs` (`teatime`)

(d) `RfBvhIntrs` (`teatime`)

Figure 9: Comparison of Performance in `default` and `teatime`

19% faster. On the `default` scene, it took roughly 1ms less time per compute pass, while `teatime` saw a reduction of nearly 50ms.

5 Extensions

Due to the modular, trait-based implementation of the RTRS codebase, it can be easily extended in a variety of ways.

- Different memory layouts can be explored through the implementation of new intersection handlers (`crate::handlers::IntrsHandler`).
- The `crate::scene::CameraController` is represented by an `enum`, allowing new control methods to be easily implemented. One such controller might be a free camera, that tracks mouse events to change the look direction. This would require a slight rework of `CameraController::handle_events`, as it's currently only designed to handle key strokes.

In addition, deeper changes would facilitate a more generalized approach to ray tracing and improve RTRS as a benchmarking tool.

- `crate::handlers::IntrsStats` is rudimentary. Its interaction with `crate::timing::BenchScheduler` is largely hard-coded. In the future, the fields of the intersection statistics should be automatically displayed in its output graphs.

- The shading model is simply-by-design. However, generalizing the shader targets for vertex/fragment passes would facilitate more sophisticated shading models beyond the synthesized Lambertian, Blinn-Phong approach used in the current codebase.

- More advanced scene construction. Users should be able to transform a model before adding it to the scene. This would entail parsing arguments for rotation, translation, and scaling. The simplest approach would be accepting a transformation matrix that is applied to each primitive prior to its addition to the scene.

Lastly, the project has been structured to be WASM-compatible. However, with the limited time available to develop RTRS and perform the study, it has been left neglected.

Currently, the project can be built for web using NPM (`npm run build && npm start`). Performing the build hosts a blank scene at `localhost:8080`. The frontend implements the same resizing behavior as native, and allows users to switch between scenes using a dashboard. However, render options cannot be tweaked and benchmarking is completely disabled—a consequence of WebGPU's lack of support for timestamp queries.

To extend RTRS' accessibility, the web implementation should be brought up to parity with the `load` binary. This can be achieved by allowing visitors to tweak `crate::ComputeConfig` options like bounces and ray tracing resolution.

A Building the Project

RTRS requires Rust 1.77.2 or newer. No nightly features are required. Rust can be easily installed with `rustup` (`https:\rustup.rs`). If deploying locally to the web or GitHub Pages, NPM is needed.

B Using the CLI

Although RTRS' API provides all the tools to run a ray tracing instance from Rust code, it's easier to use the included binaries.

Note that the binaries themselves aren't commands. It's necessary to invoke them through `cargo`, Rust's package manager with the following syntax:
`cargo run --features="cli" --bin [NAME] -- [OPTIONS]`, where `[NAME]` represents the desired binary and `[OPTIONS]` is a list of options. The `cli` feature requires some bulky parsing packages—its inclusion is feature-dependent to reduce binary size.

B.1 Building Scenes (`construct`)

Allows a scene to be constructed from OBJ files, lights, etc. The settings for `scenes/default.json` are provided below as a starting point.

```
cargo run --features=”cli” --bin construct --
  --out scenes/default.json
  --camera-pos 0.0 0.0 10.0 0.0 0.0 0.0
  --camera-orbit
  --light 20.0 20.0 20.0 1.5
  --light 0.0 50.0 25.0 1.8
  --light 30.0 20.0 0.0 1.7
  --material 0.4 0.4 0.3 0.6 0.3 0.1 50.0
  --material 0.3 0.1 0.1 0.9 0.1 0.0 10.0
  --model meshes/dodecahedron.obj 0
  --model meshes/tetrahedron.obj 1
```

Required Parameters

`out` *ex.* `--out scenes/demo.json`

Must be followed by a path a JSON file where the scene will be written. If the file does not exist, it will be created.

`model` *ex.* `--model meshes/teapot.obj 0`

Takes a path to an OBJ file and a material index. Target file can describe object geometry using any primitive. Quads and other components are broken down into triangle primitives. More than one model can be specified. If no `material` flags are provided, the second argument must be 0.

`camera-pos` *ex.* `--camera-pos 10.0 5.0 5.0 0.0 0.0 0.0`

Describes the initial state of the camera. Expects 6 float values. The first three represent the XYZ of the camera; the last three describe its look target.

`camera-fixed` OR `camera-orbit` *ex.* `--camera-fixed`

The `crate::scene::CameraController` to use.

Optional Parameters

`material` *ex.* `--material 0.5 0.1 0.1 0.9 0.1 0.0 10.0`

Specifies a material using 7 float values. The first three represent the material's RGB value, the 3-dimensional albedo component, and its specularity. Materials are assigned indices as they are parsed, so order matters. If no material is provided, index 0 will be populated with a default, semi-matte red material.

`light` *ex.* `--light 0.0 10.0 0.0 1.5`

Describes a light source by its XYZ position and strength.

B.2 Run a Scene (`load`)

Loads a scene constructed with the `load` binary (Section B.2). An example of a 'performance' preset is provided below. If loading a scene with a large amount of geometry, it's worth lowering the quality first to determine the graphic's card capabilities.

```
cargo run ——features=”cli” ——bin load ——
  –w 256 –h 192
  ——handler–bvh–rf
  ——fps 30
  ——bounces 2
  ——path scenes/teatime.json
```

Required Parameters

`width`, `height`, `workgroup-size` *ex.* `-w 640 -h 480 --workgroup-size 16`

These three parameters describe the resolution of the render. They can be provided in three configurations: [1] `workgroup-size`, [2] `width` and `height`, [3] `workgroup-size`, `width` and `height`. If the dimensions of the render are not specified, they will be dynamically resized to match the size of the window provided that the workgroup size is set. A reasonable default for `workgroup-size` is 16—any power of two $< 2^4$ will maintain performance. RTRS will calculate the `workgroup-size` based on the GCD of the window's width and height if dimension flags are omittted.

Optional Parameters

`path` *ex.* `--path scenes/teatime.json`

Specifies the path to the target scene file. If this argument is not specified, `scenes/default.json` will be loaded.

`handler-bvh` OR `handler-bvh-rf` OR `handler-naive`

Indicates the `crate::handlers::IntrsHandler` to be used. Defaults to `crate::handler::BlankIntrs`, which does not perform intersection tests and will produce an empty render. When using `handler-bvh` or `handler-bvh-rf`, a path to a precomputed BVH may be specified. If no argument is provided to the handler flag, the BVH will be computed at runtime.

`benchmark` *ex.* `--benchmark`

Toggles compute pass benchmarking. This produces an a graph in the project's root directory that is updated live as rendering proceeds.

`fps` *ex.* `--fps 30`

Allows the user to specify the target FPS. Defaults to 60 if this flag is omitted.

`bounces` *ex.* `--bounces 8`

Specify the number of times each camera ray should bounce through the scene. Higher bounce count results in more detailed reflections. Defaults to 4.

`camera-light-strength` *ex.* `--camera-light-strength 2.0`

Controls the strength of the camera's light source. By default, the camera does not emit light. If a generated scene contains no light sources, this flag should be set.

`ambience` *ex.* `--ambience 0.4`

Specifies the strength of the scene's ambient lighting. Objects will only be lit by light sources if this paramter is omitted.

B.3 Precompute a BVH (`precompute`)

Allows the user to construct a BVH for a scene ahead of time, reducing start up time. An example of BVH precomputation for the reduced-footprint BVH intersection handler is shown below.

```
cargo run --features=”cli” --bin precompute --
  --out scenes/teatime.bvh.json
  --scene scenes/teatime.json
  --eps 0.8
  --item-count 8
```

Required Parameters

`out` *ex.* `--out scenes/teatime.bvh.json`

The BVH is serialized into JSON and written into this file.

`scene` *ex.* `--scene scenes/teatime.json`

Specifies the target scene.

`item-count` *ex.* `--item-count 8`

Specifies the maximum number of items to allow in each leaf node of the BVH. When precomputing scenes for use with the `crate::handlers::BvhIntrs` handler, this value is unrestricted (although 2 is a reasonable default). `crate::handlers::RfBvhIntrs` requires `item-count`$<$ 8.

Optional Parameters

`eps` *ex.* `--eps 1.0`

Specifies the minimum size of a bounding box. If this parameter is omitted, the default value (0.02) will be used. BVH subdivision stops automatically, but this parameter can limit the depth of the tree if a larger value is provided.