Comptime Ray Tracing Pipeline

A GPU-Style Architecture in Zig

Mirroring RTX/DXR/OptiX Pipeline Design
with Compile-Time Specialization

Architecture Documentation

Contents

1 Introduction

This document describes a compile-time ray tracing pipeline architecture implemented in Zig. The design mirrors GPU ray tracing pipeline architecture (RTX/DXR/OptiX) while leveraging Zig's powerful `comptime` features for zero-cost abstractions and full optimization across the entire trace path.

1.1 Design Philosophy

The key insight is that ray tracing pipelines have well-defined stages that can be composed at compile time:

- **Ray Generation**: Creates rays (e.g., per pixel) and receives final results
- **Traversal**: Walks acceleration structures to find candidate primitives
- **Intersection**: Tests rays against actual primitive geometry
- **Hit/Miss Programs**: Execute shading or environment sampling

By making these stages compile-time parameters, the compiler generates one specialized version of the entire pipeline per unique configuration—no vtables, no function pointers, full inlining and optimization.

2 Pipeline Architecture

2.1 Visual Overview

```
┌─────────────────────────────────────────────────────────────────┐
│                        RayGenProgram                            │
│  - Generates rays (e.g., per pixel)                             │
│  - Receives final results                                       │
│  - Controls dispatch (how many rays, which pixels, etc.)        │
└─────────────────────┬───────────────────────────────────────────┘
                      │ emits Ray
                      ▼
┌─────────────────────────────────────────────────────────────────┐
│                     Traversal (Octree)                          │
│  - Walks acceleration structure                                 │
│  - Calls intersection program at leaves                         │
└─────────────────────┬───────────────────────────────────────────┘
                      │ candidate primitives
                      ▼
┌─────────────────────────────────────────────────────────────────┐
│                   IntersectionProgram                           │
│  - Tests ray against actual primitive geometry                  │
│  - Reports hit distance, attributes                             │
└─────────────────────┬───────────────────────────────────────────┘
                      │ hit/miss
                      ▼
┌──────────────────────────────┐    ┌─────────────────────────────┐
│        HitProgram            │ or │       MissProgram           │
│  - Shading, material eval    │    │  - Sky color, environment   │
│  - May spawn secondary rays  │    │                             │
└──────────────────────────────┘    └─────────────────────────────┘
```

2.2 Data Flow

1. **RayGenProgram** dispatches rays based on pixel coordinates or other criteria
2. **Traversal** walks the acceleration structure (octree/BVH) calling intersection tests
3. **IntersectionProgram** performs geometric ray-primitive tests
4. Based on results, either **HitProgram** or **MissProgram** executes
5. Results flow back to **RayGenProgram** for final output

3 Core Types

3.1 Vector and Ray Types

```
pub const Vec3 = struct {
    x: f32,
    y: f32,
    z: f32,

    pub fn dot(a: Vec3, b: Vec3) f32 {
        return a.x * b.x + a.y * b.y + a.z * b.z;
    }

    pub fn scale(v: Vec3, s: f32) Vec3 {
        return .{ .x = v.x * s, .y = v.y * s, .z = v.z * s };
    }

    pub fn add(a: Vec3, b: Vec3) Vec3 {
        return .{ .x = a.x + b.x, .y = a.y + b.y, .z = a.z + b.z };
    }
};

pub const Ray = struct {
    origin: Vec3,
    direction: Vec3,
    inv_direction: Vec3,  // precomputed for AABB tests
    t_min: f32 = 0.001,
    t_max: f32 = 1e30,
};
```

3.2 Hit Record and Trace Result

```
pub const HitRecord = struct {
    t: f32,           // distance along ray
    prim_id: u32,     // which primitive was hit
    normal: Vec3,     // surface normal at hit point
    uv: [2]f32,       // texture coordinates
};

pub const TraceResult = union(enum) {
    hit: HitRecord,
    miss,
};
```

4 Program Traits

The pipeline uses compile-time trait validation to ensure each program provides the required interface.

4.1 RayGenProgram Trait

```zig
pub fn RayGenProgramTrait(comptime Self: type) type {
    return struct {
        pub const PayloadType: type = Self.PayloadType;

        comptime {
            if (!@hasDecl(Self, "generate")) {
                @compileError("RayGenProgram must have 'generate'");
            }
            if (!@hasDecl(Self, "onResult")) {
                @compileError("RayGenProgram must have 'onResult'");
            }
        }
    };
}
```

Required interface:

- `PayloadType`: Type for per-ray data carried through the pipeline
- `generate(ctx, dispatch_id) -> ?Ray`: Create a ray for the given dispatch ID
- `onResult(ctx, dispatch_id, ray, result, payload) -> void`: Handle final result

4.2 IntersectionProgram Trait

```zig
pub fn IntersectionProgramTrait(comptime Self: type) type {
    return struct {
        comptime {
            if (!@hasDecl(Self, "intersect")) {
                @compileError("IntersectionProgram must have 'intersect'");
            }
        }
    };
}
```

Required interface:

- `intersect(ctx, ray, prim_id) -> ?HitRecord`: Test ray against primitive

4.3 Hit and Miss Program Traits

```zig
pub fn HitProgramTrait(comptime Self: type) type {
    return struct {
        comptime {
            if (!@hasDecl(Self, "onHit")) {
                @compileError("HitProgram must have 'onHit'");
            }
        }
    };
}
```

```
pub fn MissProgramTrait(comptime Self: type) type {
    return struct {
        comptime {
            if (!@hasDecl(Self, "onMiss")) {
                @compileError("MissProgram must have 'onMiss'");
            }
        }
    };
}
```

5 Pipeline Implementation

5.1 Pipeline Definition

```zig
pub fn RayTracingPipeline(comptime Config: type) type {
    // Validate config at comptime
    comptime {
        _ = RayGenProgramTrait(Config.RayGen);
        _ = IntersectionProgramTrait(Config.Intersection);
        _ = HitProgramTrait(Config.Hit);
        _ = MissProgramTrait(Config.Miss);
    }

    return struct {
        const Self = @This();
        pub const Payload = Config.RayGen.PayloadType;
        pub const Context = Config.Context;

        octree: *const Octree,

        pub fn dispatch(
            self: *const Self,
            ctx: *Context,
            dispatch_count: u32,
        ) void {
            var dispatch_id: u32 = 0;
            while (dispatch_id < dispatch_count) : (dispatch_id += 1) {
                self.dispatchSingle(ctx, dispatch_id);
            }
        }
        // ... continued
    };
}
```

5.2 Single Ray Dispatch

```zig
pub fn dispatchSingle(
    self: *const Self,
    ctx: *Context,
    dispatch_id: u32,
) void {
    // 1. Generate ray
    const maybe_ray = Config.RayGen.generate(ctx, dispatch_id);
    const ray = maybe_ray orelse return;

    // 2. Initialize payload
    var payload: Payload = if (@hasDecl(Config.RayGen, "initPayload"))
        Config.RayGen.initPayload(ctx, dispatch_id)
    else
        undefined;

    // 3. Trace
    const result = self.trace(ctx, ray);
    // 4. Execute hit/miss program
    switch (result) {
        .hit => |hit| Config.Hit.onHit(ctx, ray, hit, &payload),
        .miss => Config.Miss.onMiss(ctx, ray, &payload),
    }

    // 5. Return result to ray gen
    Config.RayGen.onResult(ctx, dispatch_id, ray, result, &payload);
}
```

5.3 Trace Implementation with Octree Traversal

```
fn trace(self: *const Self, ctx: *Context, ray: Ray) TraceResult {
    var closest: ?HitRecord = null;

    // Octree traversal (inlined)
    var stack: [64]u32 = undefined;
    var stack_top: usize = 1;
    stack[0] = 0; // root

    while (stack_top > 0) {
        stack_top -= 1;
        const node = self.octree.nodes[stack[stack_top]];

        // AABB test
        if (!rayAabbIntersect(ray, node.bounds, closest)) {
            continue;
        }

        // Test primitives at this node
        for (node.prim_ids[0..node.prim_count]) |prim_id| {
            // Comptime-inlined intersection
            if (Config.Intersection.intersect(ctx, ray, prim_id)) |hit| {
                if (closest == null or hit.t < closest.?.t) {
                    closest = hit;
                }
            }
        }

        // Push children
        if (!node.is_leaf) {
            for (node.children) |child_idx| {
                if (child_idx != NULL_NODE) {
                    stack[stack_top] = child_idx;
                    stack_top += 1;
                }
            }
        }
    }

    return if (closest) |hit| .{ .hit = hit } else .miss;
}
```

6 Example: Primary Ray Caster

6.1 Scene Context

```
const Scene = struct {
    triangles: []const Triangle,
    camera: Camera,
    framebuffer: []Vec3,
    width: u32,
    height: u32,
};
```

6.2 Ray Generation Program

```
const PrimaryRayGen = struct {
    pub const PayloadType = struct {
        color: Vec3,
    };

    pub fn generate(ctx: *Scene, dispatch_id: u32) ?Ray {
        const x = dispatch_id % ctx.width;
        const y = dispatch_id / ctx.width;
        return ctx.camera.generateRay(x, y, ctx.width, ctx.height);
    }

    pub fn initPayload(_: *Scene, _: u32) PayloadType {
        return .{ .color = Vec3{ .x = 0, .y = 0, .z = 0 } };
    }

    pub fn onResult(
        ctx: *Scene,
        dispatch_id: u32,
        _: Ray,
        _: TraceResult,
        payload: *PayloadType,
    ) void {
        ctx.framebuffer[dispatch_id] = payload.color;
    }
};
```

6.3 Intersection, Hit, and Miss Programs

```
const TriangleIntersection = struct {
    pub fn intersect(ctx: *Scene, ray: Ray, prim_id: u32) ?HitRecord {
        const tri = ctx.triangles[prim_id];
        return rayTriangleIntersect(ray, tri, prim_id);
    }
};

const SimpleHit = struct {
    pub fn onHit(
        ctx: *Scene,
        _: Ray,
        hit: HitRecord,
        payload: *PrimaryRayGen.PayloadType,
    ) void {
        _ = ctx;
        // Simple normal shading
        payload.color = Vec3{
            .x = hit.normal.x * 0.5 + 0.5,
            .y = hit.normal.y * 0.5 + 0.5,
            .z = hit.normal.z * 0.5 + 0.5,
        };
    }
};

const SkyMiss = struct {
    pub fn onMiss(
        _: *Scene,
        ray: Ray,
        payload: *PrimaryRayGen.PayloadType,
    ) void {
        // Gradient sky
        const t = 0.5 * (ray.direction.y + 1.0);
        payload.color = Vec3{
            .x = 1.0 - t * 0.5,
            .y = 1.0 - t * 0.3,
            .z = 1.0,
        };
    }
};
```

6.4 Pipeline Assembly and Usage

```
const PrimaryRayPipeline = RayTracingPipeline(.{
    .Context = Scene,
    .RayGen = PrimaryRayGen,
    .Intersection = TriangleIntersection,
    .Hit = SimpleHit,
    .Miss = SkyMiss,
});

pub fn render(scene: *Scene, octree: *const Octree) void {
    const pipeline = PrimaryRayPipeline{ .octree = octree };
    const pixel_count = scene.width * scene.height;
    pipeline.dispatch(scene, pixel_count);
}
```

7 Example: Shadow Rays with Any-Hit

7.1 Shadow Ray Generation

```
const ShadowRayGen = struct {
    pub const PayloadType = struct {
        in_shadow: bool,
    };

    pub fn generate(
        ctx: *Scene,
        hit_point: Vec3,
        light_pos: Vec3,
    ) ?Ray {
        const to_light = Vec3.sub(light_pos, hit_point);
        const dist = Vec3.length(to_light);
        const dir = Vec3.scale(to_light, 1.0 / dist);

        return Ray{
            .origin = Vec3.add(hit_point, Vec3.scale(dir, 0.001)),
            .direction = dir,
            .inv_direction = .{
                .x = 1.0 / dir.x,
                .y = 1.0 / dir.y,
                .z = 1.0 / dir.z,
            },
            .t_min = 0.0,
            .t_max = dist - 0.001,
        };
    }

    pub fn initPayload(_: *Scene, _: u32) PayloadType {
        return .{ .in_shadow = false };
    }

    pub fn onResult(
        _: *Scene, _: u32, _: Ray, _: TraceResult, _: *PayloadType
    ) void {}
};
```

7.2 Any-Hit Early Termination

For shadow rays, we only need to know if **any** intersection exists—we can terminate early:

```
const ShadowAnyHit = struct {
    pub fn onHit(
        _: *Scene,
        _: Ray,
        _: HitRecord,
        payload: *ShadowRayGen.PayloadType,
    ) void {
        payload.in_shadow = true;
    }
};
```

```
// Pipeline with any_hit flag for early termination
const ShadowPipeline = RayTracingPipeline(.{
    .Context = Scene,
    .RayGen = ShadowRayGen,
    .Intersection = ShadowIntersection,
    .Hit = ShadowAnyHit,
    .Miss = ShadowMiss,
    .any_hit = true,  // enables early termination
});
```

The pipeline modification for any-hit:

```
const is_any_hit = @hasDecl(Config, "any_hit") and Config.any_hit;

// In trace function:
if (Config.Intersection.intersect(ctx, ray, prim_id)) |hit| {
    if (comptime is_any_hit) {
        // Early exit on first hit
        Config.Hit.onHit(ctx, ray, hit, payload);
        return .{ .hit = hit };
    }
    // Otherwise find closest...
}
```

8 Example: Path Tracer with Recursive Rays

8.1 Path Trace Context and Payload

```
const PathTraceContext = struct {
    scene: *Scene,
    rng: *std.rand.DefaultPrng,
    max_bounces: u32,
};

const PathTracePayload = struct {
    throughput: Vec3,
    radiance: Vec3,
    bounce: u32,
    next_ray: ?Ray,
};
```

8.2 Path Trace Hit Program

```
const PathTraceHit = struct {
    pub fn onHit(
        ctx: *PathTraceContext,
        ray: Ray,
        hit: HitRecord,
        payload: *PathTracePayload,
    ) void {
        const material = ctx.scene.materials[hit.prim_id];

        // Add emission
        payload.radiance = Vec3.add(
            payload.radiance,
            Vec3.mul(payload.throughput, material.emission),
        );

        if (payload.bounce >= ctx.max_bounces) {
            return;
        }

        // Sample BRDF for next direction
        const hit_point = Vec3.add(
            ray.origin, Vec3.scale(ray.direction, hit.t)
        );
        const new_dir = sampleCosineHemisphere(hit.normal, ctx.rng);

        // Update throughput
        const brdf = material.albedo;
        const cos_theta = Vec3.dot(new_dir, hit.normal);
        payload.throughput = Vec3.scale(
            Vec3.mul(payload.throughput, brdf),
            cos_theta * std.math.pi,
        );

        // Queue next ray
        payload.next_ray = Ray{
            .origin = Vec3.add(hit_point, Vec3.scale(hit.normal, 0.001)),
            .direction = new_dir,
            .inv_direction = Vec3.inverse(new_dir),
            .t_min = 0.0,
            .t_max = 1e30,
        };
        payload.bounce += 1;
    }
};
```

8.3 Recursive Ray Loop

```
pub fn dispatchSingle(self: *const Self, ctx: *Context, dispatch_id: u32) void {
    var maybe_ray = Config.RayGen.generate(ctx, dispatch_id);
    var payload = if (@hasDecl(Config.RayGen, "initPayload"))
        Config.RayGen.initPayload(ctx, dispatch_id)
    else
        undefined;

    while (maybe_ray) |ray| {
        const result = self.trace(ctx, ray);

        switch (result) {
            .hit => |hit| Config.Hit.onHit(ctx, ray, hit, &payload),
            .miss => Config.Miss.onMiss(ctx, ray, &payload),
        }

        // Check for continuation ray (recursive bounce)
        if (@hasField(Payload, "next_ray")) {
            maybe_ray = payload.next_ray;
            payload.next_ray = null;
        } else {
            break;
        }
    }

    Config.RayGen.onResult(ctx, dispatch_id, maybe_ray orelse undefined, .miss,
&payload);
}
```

9 Example: Multi-Hit for Transparency

For transparency or CSG operations, we need all hits along the ray:

```
const MultiHitConfig = struct {
    pub const max_hits = 16;
};

const TransparencyIntersection = struct {
    const HitList = std.BoundedArray(HitRecord, MultiHitConfig.max_hits);

    pub fn intersectAll(
        ctx: *Scene,
        ray: Ray,
        prim_id: u32,
        hits: *HitList,
    ) void {
        if (rayTriangleIntersect(ray, ctx.triangles[prim_id], prim_id)) |hit| {
            hits.append(hit) catch {}; // drop if full
        }
    }
};
```

The multi-hit pipeline variant collects all intersections and sorts by distance:

```
fn trace(self: *const Self, ctx: *Context, ray: Ray) []const HitRecord {
    var hits = TransparencyIntersection.HitList{};

    // Traversal collects ALL hits...

    // Sort by distance
    std.sort.sort(HitRecord, hits.slice(), {}, struct {
        fn lessThan(_: void, a: HitRecord, b: HitRecord) bool {
            return a.t < b.t;
        }
    }.lessThan);

    return hits.constSlice();
}
```

10 Comptime Inlining Summary

Component	**Inlined?**	**Reason**
`RayGen.generate`	✓	Comptime function parameter
`Intersection.intersect`	✓	Comptime function parameter
`Hit.onHit`	✓	Comptime function parameter
`Miss.onMiss`	✓	Comptime function parameter
Traversal loop	✓	Part of monomorphized pipeline
AABB tests	✓	Internal to pipeline

The compiler generates **one specialized version** of the entire pipeline per unique `Config` type. This enables:

- No vtables or function pointers
- Full inlining across program boundaries
- Dead code elimination for unused features
- Constant propagation for comptime-known values

11 Appendix: Ray-AABB Intersection

```
fn rayAabbIntersect(ray: Ray, bounds: AABB, closest: ?HitRecord) bool {
    const t_max = if (closest) |h| h.t else ray.t_max;

    const t1x = (bounds.min.x - ray.origin.x) * ray.inv_direction.x;
    const t2x = (bounds.max.x - ray.origin.x) * ray.inv_direction.x;
    const t1y = (bounds.min.y - ray.origin.y) * ray.inv_direction.y;
    const t2y = (bounds.max.y - ray.origin.y) * ray.inv_direction.y;
    const t1z = (bounds.min.z - ray.origin.z) * ray.inv_direction.z;
    const t2z = (bounds.max.z - ray.origin.z) * ray.inv_direction.z;

    const t_enter = @max(
        @max(@min(t1x, t2x), @min(t1y, t2y)),
        @min(t1z, t2z)
    );
    const t_exit = @min(
        @min(@max(t1x, t2x), @max(t1y, t2y)),
        @max(t1z, t2z)
    );

    return t_exit >= t_enter and t_exit >= ray.t_min and t_enter <= t_max;
}
```

This implementation uses the slab method with precomputed inverse directions for efficient AABB intersection testing during octree traversal.