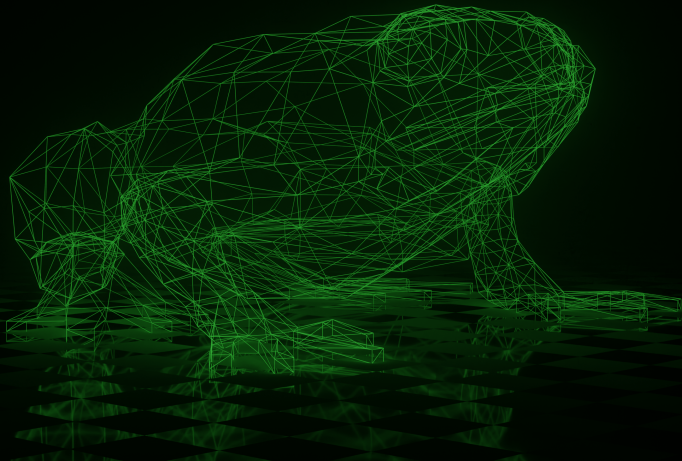


Improving the World's Slowest Raytracer

Friedrich Vock

October 17, 2023



Contents

- Raytracing: A Quick Recap
 - Acceleration Structures
 - Raytracing Pipelines
- Implementing Acceleration Structures
 - LBVH
 - PLOC
- Implementing Raytracing Pipelines
- Status And What's To Come
 - Where We Are
 - Where We're Going

Raytracing: A Quick Recap



Acceleration Structures

- Opaque from the app's POV
- Contains scene geometry in format understood by ray accelerator HW
- Built by the driver, app provides “primitive soup”
- Three primitive types: Triangles, AABBs and instances of other BVHs
 - With AABB primitives, a shader can execute custom code to determine if a ray hits the surface
 - Vulkan defines two levels of acceleration structures: Top-Level and Bottom-Level
 - ▷ TLAS can only have BLAS instances as primitives
 - ▷ BLAS can only have triangles or AABBs as primitives

Raytracing Pipelines

- Special type of pipeline for performing raytracing
- New shader stages: Ray Generation, Any Hit, Intersection, Closest Hit, Miss, Callable

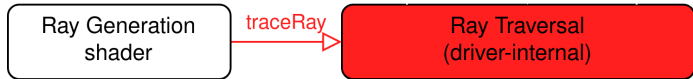
Raytracing Pipelines

- Special type of pipeline for performing raytracing
- New shader stages: Ray Generation, Any Hit, Intersection, Closest Hit, Miss, Callable

Ray Generation
shader

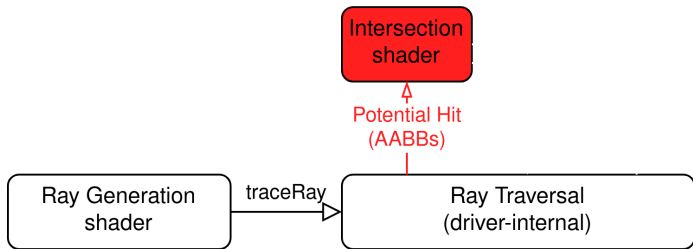
Raytracing Pipelines

- Special type of pipeline for performing raytracing
- New shader stages: Ray Generation, Any Hit, Intersection, Closest Hit, Miss, Callable



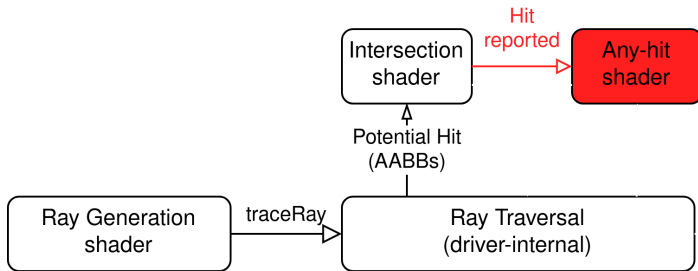
Raytracing Pipelines

- Special type of pipeline for performing raytracing
- New shader stages: Ray Generation, Any Hit, Intersection, Closest Hit, Miss, Callable



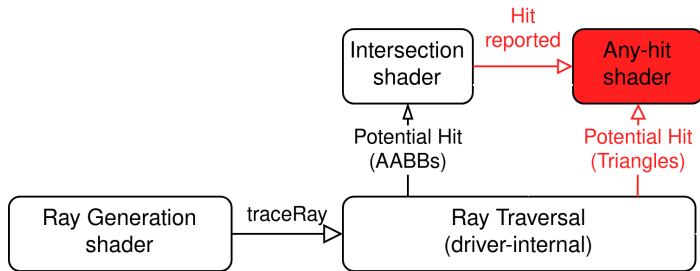
Raytracing Pipelines

- Special type of pipeline for performing raytracing
- New shader stages: Ray Generation, Any Hit, Intersection, Closest Hit, Miss, Callable



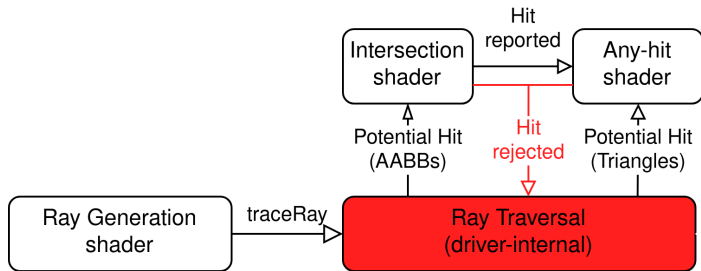
Raytracing Pipelines

- Special type of pipeline for performing raytracing
- New shader stages: Ray Generation, Any Hit, Intersection, Closest Hit, Miss, Callable



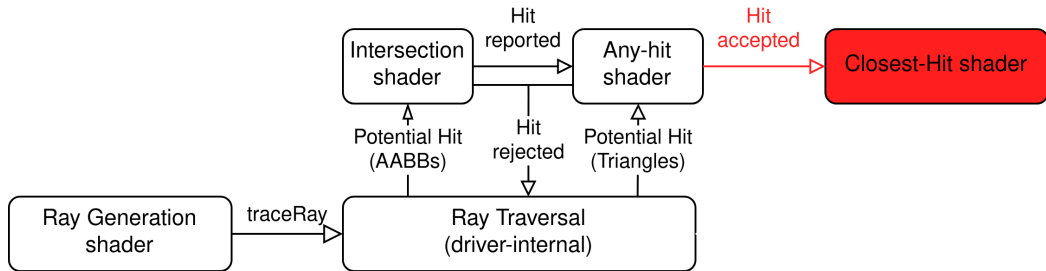
Raytracing Pipelines

- Special type of pipeline for performing raytracing
- New shader stages: Ray Generation, Any Hit, Intersection, Closest Hit, Miss, Callable



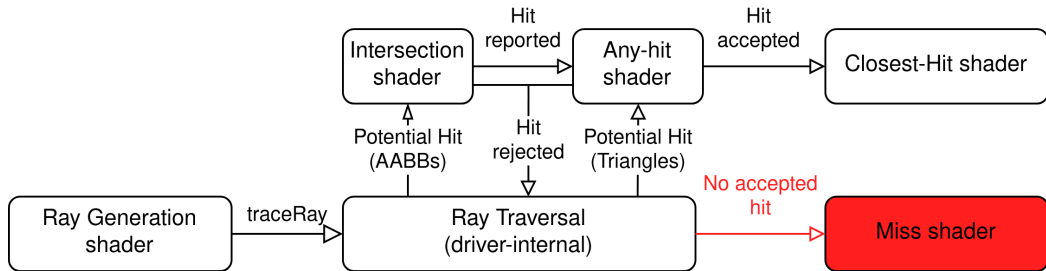
Raytracing Pipelines

- Special type of pipeline for performing raytracing
- New shader stages: Ray Generation, Any Hit, Intersection, Closest Hit, Miss, Callable



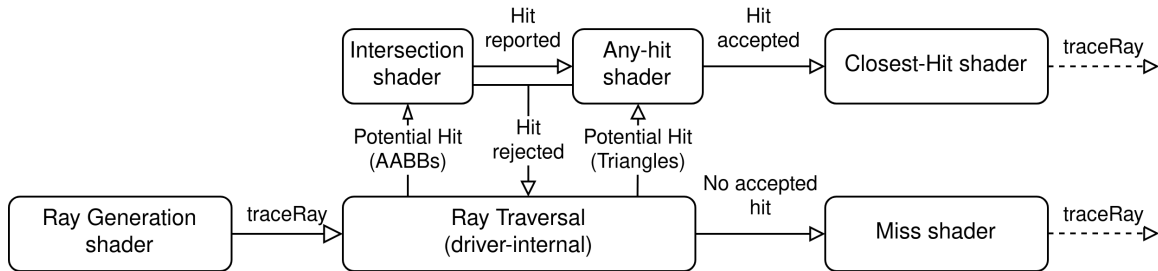
Raytracing Pipelines

- Special type of pipeline for performing raytracing
- New shader stages: Ray Generation, Any Hit, Intersection, Closest Hit, Miss, Callable



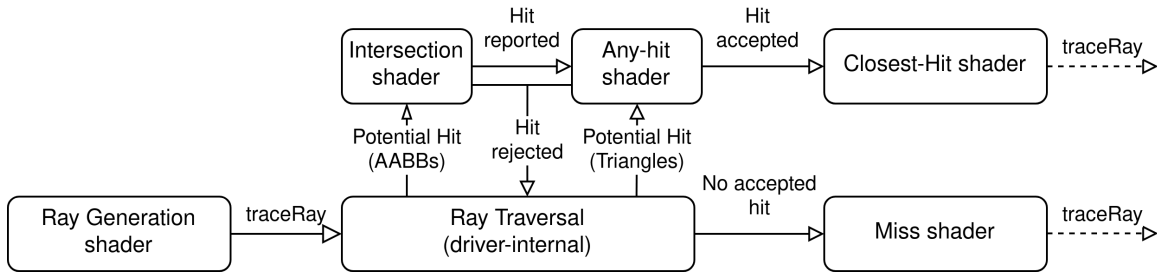
Raytracing Pipelines

- Special type of pipeline for performing raytracing
- New shader stages: Ray Generation, Any Hit, Intersection, Closest Hit, Miss, Callable



Raytracing Pipelines

- Special type of pipeline for performing raytracing
- New shader stages: Ray Generation, Any Hit, Intersection, Closest Hit, Miss, Callable

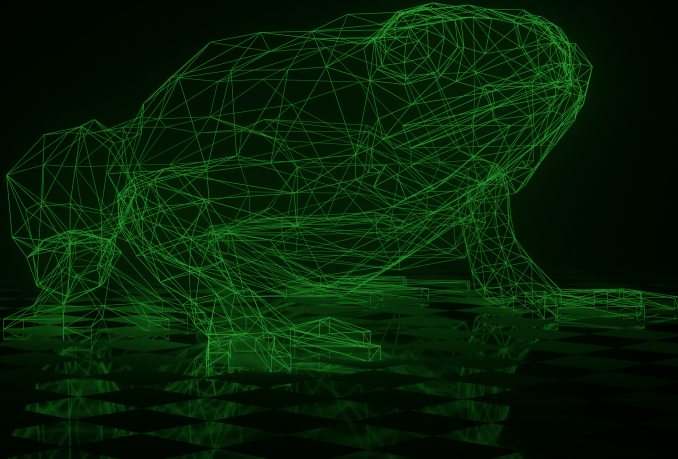


- Callable shaders can be called from RGen/CHit/Miss(/Callable)

Other RT Pipeline concepts

- Shader Binding Tables (SBTs)
 - More than one shader for each stage!
 - App tells the driver which shader to call with a Shader Binding Table
 - Jump Table on the GPU
- Pipeline Libraries
 - Incomplete pipelines that can be linked with other pipelines
 - Useful for e.g. moving commonly-used shaders into common library
 - ▷ Pipelines using these shaders only need to link to the previously-created library

Implementing Acceleration Structures



Acceleration Structure Building

- RADV implements acceleration structures as BVHs
 - Tree of AABBs and triangles
 - Internal nodes are AABBs, each leaf node represents a primitive
 - A node's AABB is the AABB of its child nodes
- Hardware format is BVH4
 - Each internal node can have up to 4 child nodes

Node ID packing:

- Node offset in BVH / 8
- 3 LSBs used for node type

```
struct radv_bvh_box32_node {
    uint32_t children[4];
    radv_aabb coords[4];
    uint32_t reserved[4];
};
```

```
struct radv_bvh_triangle_node {
    float coords[3][3];
    uint32_t reserved[3];
    uint32_t triangle_id;
    /* flags in upper 4 bits */
    uint32_t geometry_id_and_flags;
    uint32_t reserved2;
    uint32_t id;
};
```

Acceleration Structure Building

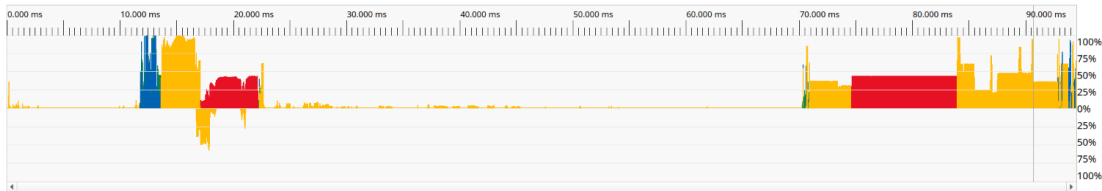
- Building acceleration structures is a multi-step process
 1. Convert input primitives to leaf nodes
 2. Sort nodes
 3. Construct binary BVH
 4. Convert binary BVH to BVH4
- Need barriers in between each step!
 - Bad for building many tiny acceleration structures
 - Vulkan allows for building multiple acceleration structures at a time

Acceleration Structure Building

- Building acceleration structures is a multi-step process
 1. Convert input primitives to leaf nodes
 2. Sort nodes
 3. Construct binary BVH
 4. Convert binary BVH to BVH4
- Need barriers in between each step!
 - Bad for building many tiny acceleration structures
 - Vulkan allows for building multiple acceleration structures at a time

Acceleration Structure Building

- Building acceleration structures is a multi-step process
 1. Convert input primitives to leaf nodes
 2. Sort nodes
 3. Construct binary BVH
 4. Convert binary BVH to BVH4
- Need barriers in between each step!
 - Bad for building many tiny acceleration structures
 - Vulkan allows for building multiple acceleration structures at a time
- This is what happens when you don't batch:



BVH Building Algorithms



Linear BVH

- Algorithm from 2014
- Builds a radix tree from generated morton codes
- Tree can be built up independently for each node
 - Great for parallelization!
- Used for quick builds

Parallel Locally-Ordered Clustering

- Algorithm from 2017
- Better quality than LBVH, though slower
- Original reference implementation was written in CUDA
- Heavily relies on device-wide synchronization
 - No global synchronization primitives available
 - Have to make our own!

The Poor Man's Global Synchronization

- All build shaders are written in Vulkan GLSL
- Compiler and target hardware is known
- Can assume more guarantees in certain aspects
 - Forward Progress
 - Launch ordering
- Work Stealing Queue
 - Each task receives a unique ID
 - Runtime-defined number of tasks to launch before synchronizing

Launching new tasks:

The Poor Man's Global Synchronization

- All build shaders are written in Vulkan GLSL
- Compiler and target hardware is known
- Can assume more guarantees in certain aspects
 - Forward Progress
 - Launch ordering
- Work Stealing Queue
 - Each task receives a unique ID
 - Runtime-defined number of tasks to launch before synchronizing

Launching new tasks:

```
taskID = atomicAdd(taskStartedCounter, 1);
```

The Poor Man's Global Synchronization

Making sure previous work has finished:

- Maintain a “tasks finished” counter and increment it when finished
- If the new work ID is larger than the maximum task ID to launch before synchronizing, busy-wait until the maximum task ID is updated
- The last task to finish updates the maximum task ID

```
finishID = atomicAdd(taskFinishCounter, 1);  
if (finishID == maxTaskID) {  
    atomicAdd(maxTaskID, nextNumberOfTasks);  
} else {  
    while (taskID > maxTaskID) {}  
}
```

Totally sane BVH building code

```

controlBarrier(gl_ScopeWorkgroup, gl_ScopeDevice, gl_StorageSemanticsBuffer,
               gl_SemanticsAcquireRelease | gl_SemanticsMakeAvailable | gl_SemanticsMakeVisible);
if (gl_LocalInvocationIndex == 0) {
    if (did_work)
        atomicAdd(DEREF(header).sync_data.task_done_counter, 1);
    global_task_index = atomicAdd(DEREF(header).sync_data.task_started_counter, 1);

    do {
        /* Perform a memory barrier to refresh the current phase's end counter, in case
         * another workgroup changed it. */
        memoryBarrier(gl_ScopeDevice, gl_StorageSemanticsBuffer,
                     gl_SemanticsAcquireRelease | gl_SemanticsMakeAvailable | gl_SemanticsMakeVisible);

        /* The first invocation of the first workgroup in a new phase is responsible to initiate the
         * switch to a new phase. It is only possible to switch to a new phase if all tasks of the
         * previous phase have been completed. Switching to a new phase and incrementing the phase
         * end counter in turn notifies all invocations for that phase that it is safe to execute.
         */
        if (global_task_index == DEREF(header).sync_data.current_phase_end_counter &&
            DEREF(header).sync_data.task_done_counter == DEREF(header).sync_data.current_phase_end_counter) {
            if (DEREF(header).sync_data.next_phase_exit_flag != 0) {
                DEREF(header).sync_data.phase_index = TASK_INDEX_INVALID;
                memoryBarrier(gl_ScopeDevice, gl_StorageSemanticsBuffer,
                             gl_SemanticsAcquireRelease | gl_SemanticsMakeAvailable | gl_SemanticsMakeVisible);
            } else {
                atomicAdd(DEREF(header).sync_data.phase_index, 1);
                DEREF(header).sync_data.current_phase_start_counter = DEREF(header).sync_data.current_phase_end_counter;
                /* Ensure the changes to the phase index and start/end counter are visible for other
                 * workgroup waiting in the loop. */
                memoryBarrier(gl_ScopeDevice, gl_StorageSemanticsBuffer,
                             gl_SemanticsAcquireRelease | gl_SemanticsMakeAvailable | gl_SemanticsMakeVisible);
                atomicAdd(DEREF(header).sync_data.current_phase_end_counter,
                          DIV_ROUND_UP(task_count(header), gl_WorkGroupSize.x));
            }
            break;
        }

        /* If other invocations have finished all nodes, break out; there is no work to do */
        if (DEREF(header).sync_data.phase_index == TASK_INDEX_INVALID) {
            break;
        }
    }
}

```

Implementing Raytracing Pipelines



Hardware Acceleration Features on AMD

- Instruction for hardware-accelerated ray-BVH intersection:
`image_bvh_intersect_ray`
 - Calculates intersection results for a single BVH node
 - Returns child nodes sorted by intersection distance for internal AABB nodes
 - Returns intersection results (incl. barycentrics for triangles) for leaf nodes
 - New in RDNA3: Ray flag handling and more box sorting modes
- New in RDNA3: Instruction for LDS traversal stack: `ds_bvh_stack_rtn`
 - Optimization for stack handling
 - Not used in RADV (yet)

Hardware Acceleration Features on AMD

- Instruction for hardware-accelerated ray-BVH intersection:
`image_bvh_intersect_ray`
 - Calculates intersection results for a single BVH node
 - Returns child nodes sorted by intersection distance for internal AABB nodes
 - Returns intersection results (incl. barycentrics for triangles) for leaf nodes
 - New in RDNA3: Ray flag handling and more box sorting modes
- New in RDNA3: Instruction for LDS traversal stack: `ds_bvh_stack_rtn`
 - Optimization for stack handling
 - Not used in RADV (yet)
- All the rest: Implemented in software



Raytracing Pipelines



Raytracing Pipelines: The
Hardware-Accelerated Part

Layout of a Raytracing pipeline - XDC 2022

- Inline all raytracing shaders into a single compute shader
- Shaders are referred to by IDs
- Wrapped in a loop that continues executing shaders unless exit is requested

```
void main() {  
    uint32_t shader_id = raygen_sbt[0];  
    while (shader_id != 0) {  
        switch (shader_id) {  
            // paste all shader code in here  
        }  
    }  
}
```


Layout of a Raytracing pipeline - XDC 2022

Problem: Raytracing pipelines can get **big** (2000+ shaders)

- Size of shaders caused stack overflows during passes that operated recursively on blocks
- Does not work well with pipeline libraries - have to copy shader code around and defer actual compilation

Layout of a Raytracing pipeline - XDC 2023

“Monolithic” mode:

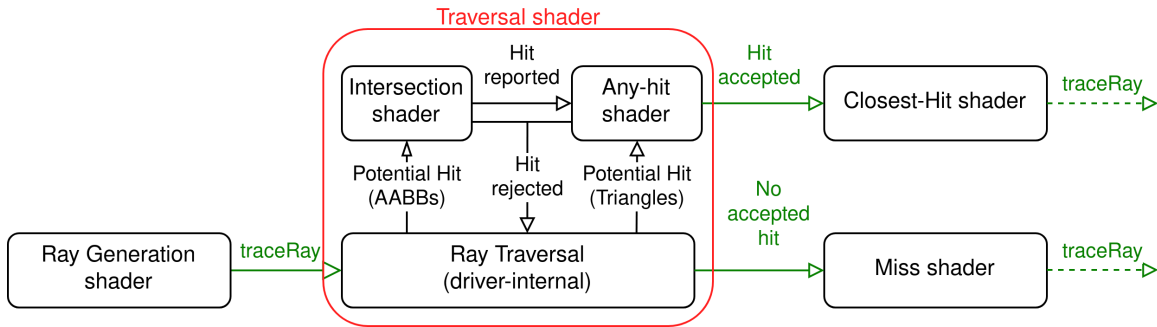
- All shaders are fully inlined
- No shader call overhead, potential for constant propagation
- No recursion, no support for pipeline libraries

```
void traceRay() // always inlined {  
    // paste traversal shader  
    if (hit)  
        // paste closest-hit shader;  
    else  
        // paste miss shader  
}  
  
void main() {  
    // paste raygen shader  
}
```

Layout of a Raytracing pipeline - XDC 2023

“Separate Compilation” mode:

- Shader calls with indirect jump
- Allows shaders from pipeline libraries to be compiled ahead of time



Layout of a Traversal Shader

```
void main() {  
    while (!traversal_complete) {  
        uint32_t node_pointer = get_next_node();  
        intersection_result result = image_bvh_intersect_ray(node_pointer);  
        if (result.hit) {  
            // paste any-hit/intersection shaders here  
            if (!hit_accepted)  
                continue;  
        }  
    }  
    if (hit)  
        call_closest_hit();  
    else  
        call_miss();  
}
```

Why not separately-compile the traversal shader?

- Ray Traversal has lots of live state
- Any-Hit/Intersection shaders are relatively small yet called often
- High shader call overhead, negatively impacts performance

Implementing Indirect Jumps

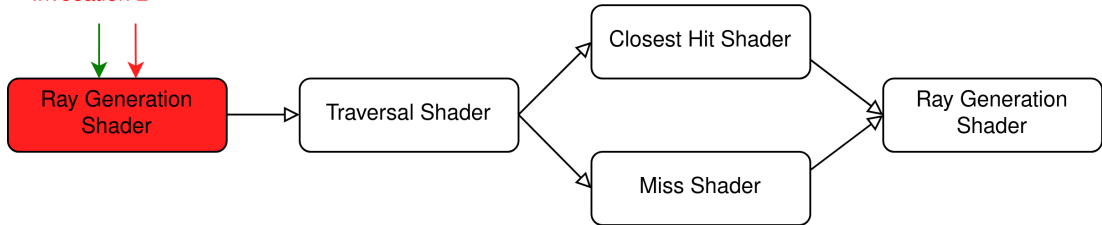
- Jump execution to arbitrary addresses stored in SBT
- Each invocation in a wavefront may have a different address
- Only one program counter per wavefront
- Naive solution: Choose program counter of first valid invocation
- Guard shader invocations to prevent invocations from executing wrong shaders
- Problem: Reconvergent shader calls become inefficient!

Implementing Indirect Jumps

- Jump execution to arbitrary addresses stored in SBT
- Each invocation in a wavefront may have a different address
- Only one program counter per wavefront
- Naive solution: Choose program counter of first valid invocation
- Guard shader invocations to prevent invocations from executing wrong shaders
- Problem: Reconvergent shader calls become inefficient!

Invocation 1

Invocation 2

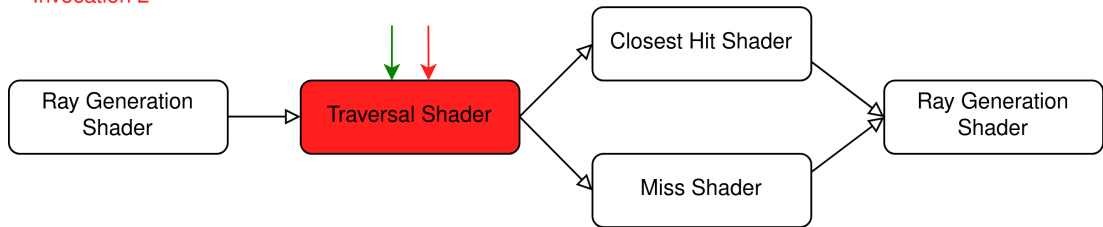


Implementing Indirect Jumps

- Jump execution to arbitrary addresses stored in SBT
- Each invocation in a wavefront may have a different address
- Only one program counter per wavefront
- Naive solution: Choose program counter of first valid invocation
- Guard shader invocations to prevent invocations from executing wrong shaders
- Problem: Reconvergent shader calls become inefficient!

Invocation 1

Invocation 2

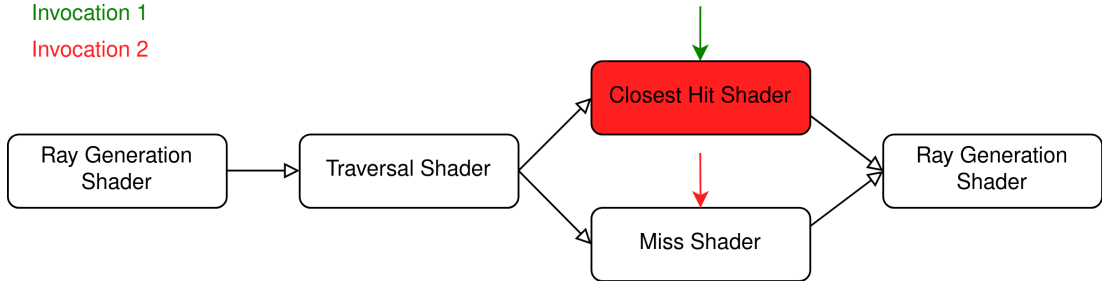


Implementing Indirect Jumps

- Jump execution to arbitrary addresses stored in SBT
- Each invocation in a wavefront may have a different address
- Only one program counter per wavefront
- Naive solution: Choose program counter of first valid invocation
- Guard shader invocations to prevent invocations from executing wrong shaders
- Problem: Reconvergent shader calls become inefficient!

Invocation 1

Invocation 2

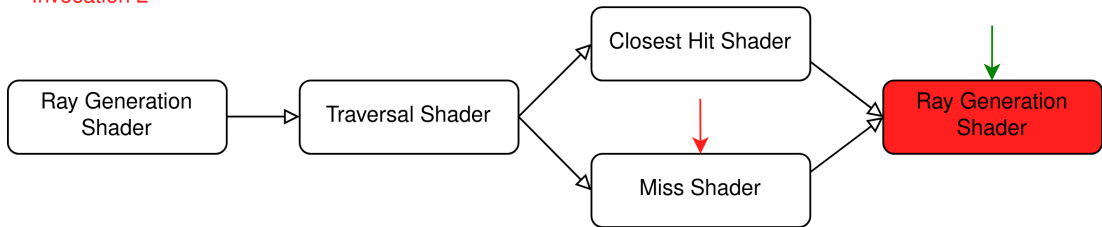


Implementing Indirect Jumps

- Jump execution to arbitrary addresses stored in SBT
- Each invocation in a wavefront may have a different address
- Only one program counter per wavefront
- Naive solution: Choose program counter of first valid invocation
- Guard shader invocations to prevent invocations from executing wrong shaders
- Problem: Reconvergent shader calls become inefficient!

Invocation 1

Invocation 2

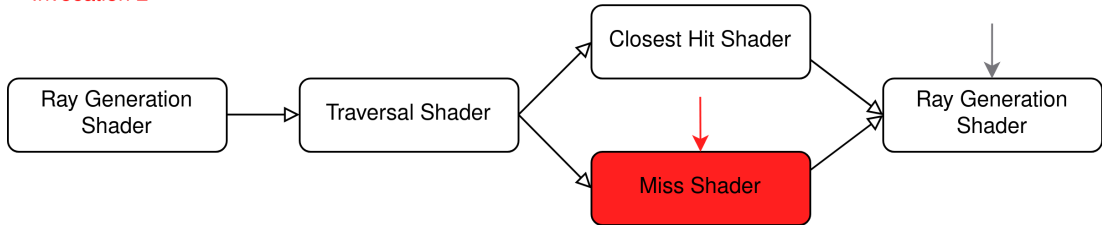


Implementing Indirect Jumps

- Jump execution to arbitrary addresses stored in SBT
- Each invocation in a wavefront may have a different address
- Only one program counter per wavefront
- Naive solution: Choose program counter of first valid invocation
- Guard shader invocations to prevent invocations from executing wrong shaders
- Problem: Reconvergent shader calls become inefficient!

Invocation 1

Invocation 2

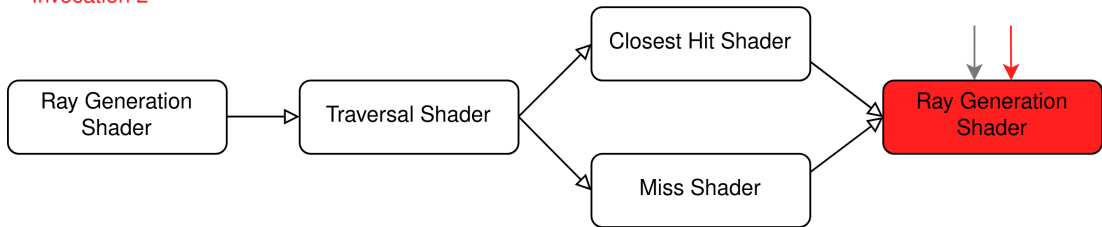


Implementing Indirect Jumps

- Jump execution to arbitrary addresses stored in SBT
- Each invocation in a wavefront may have a different address
- Only one program counter per wavefront
- Naive solution: Choose program counter of first valid invocation
- Guard shader invocations to prevent invocations from executing wrong shaders
- Problem: Reconvergent shader calls become inefficient!

Invocation 1

Invocation 2

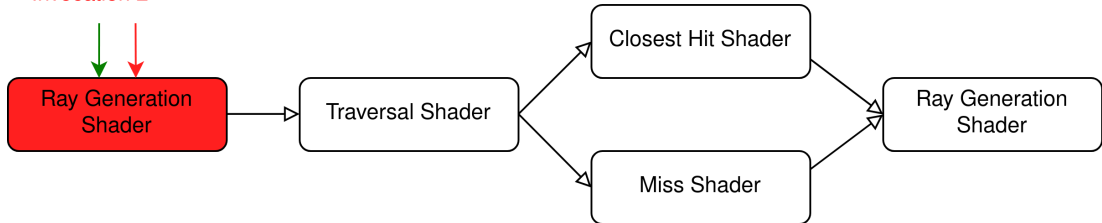


Implementing Indirect Jumps

- Solution: Select shader based on type
- Certain shader types will always be preferred over others
- Prefer Hit/Miss over Traversal shaders
- Execute Raygen only when no other shaders are available
- Shader type is packed into the 2 LSBs of the shader's address

Invocation 1

Invocation 2

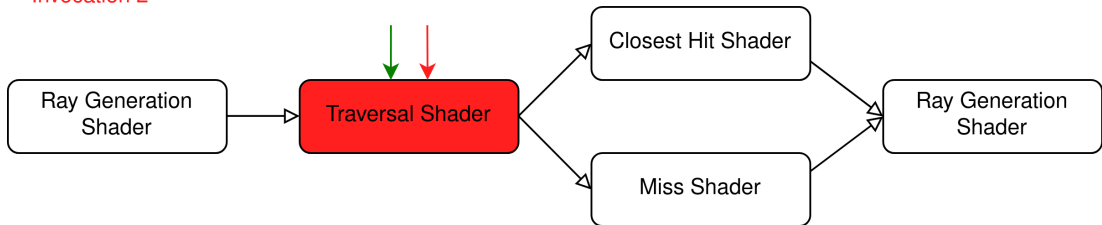


Implementing Indirect Jumps

- Solution: Select shader based on type
- Certain shader types will always be preferred over others
- Prefer Hit/Miss over Traversal shaders
- Execute Raygen only when no other shaders are available
- Shader type is packed into the 2 LSBs of the shader's address

Invocation 1

Invocation 2

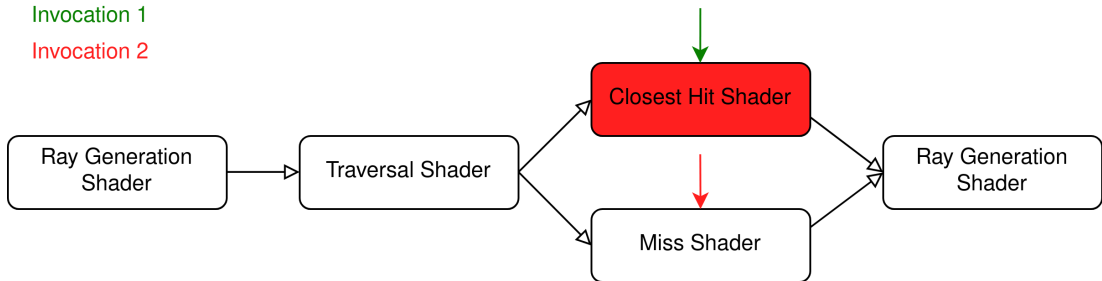


Implementing Indirect Jumps

- Solution: Select shader based on type
- Certain shader types will always be preferred over others
- Prefer Hit/Miss over Traversal shaders
- Execute Raygen only when no other shaders are available
- Shader type is packed into the 2 LSBs of the shader's address

Invocation 1

Invocation 2

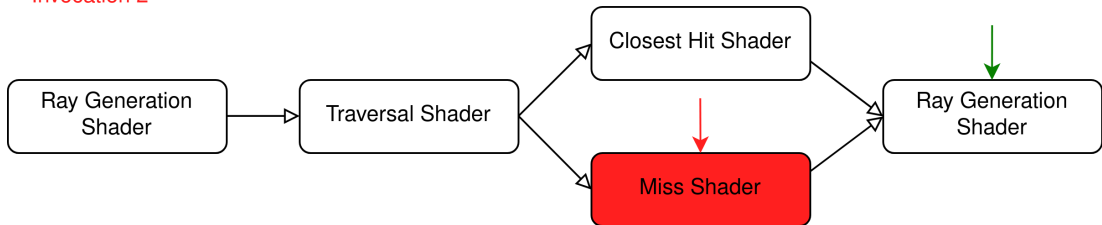


Implementing Indirect Jumps

- Solution: Select shader based on type
- Certain shader types will always be preferred over others
- Prefer Hit/Miss over Traversal shaders
- Execute Raygen only when no other shaders are available
- Shader type is packed into the 2 LSBs of the shader's address

Invocation 1

Invocation 2

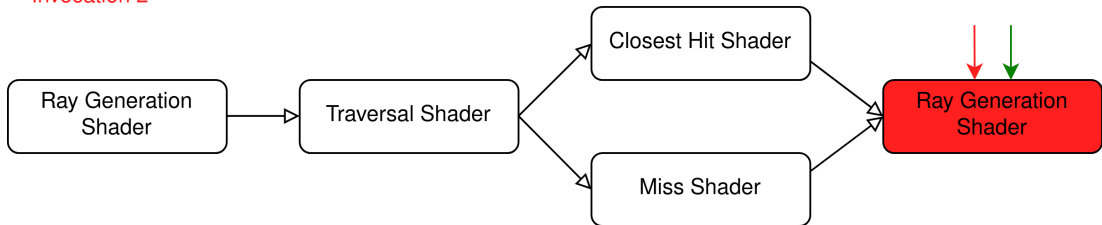


Implementing Indirect Jumps

- Solution: Select shader based on type
- Certain shader types will always be preferred over others
- Prefer Hit/Miss over Traversal shaders
- Execute Raygen only when no other shaders are available
- Shader type is packed into the 2 LSBs of the shader's address

Invocation 1

Invocation 2



Status And What's To Come



Compatibility

XDC 2022:

- Quake II RTX
- Control
- Deathloop
- Resident Evil: Village
- Metro Exodus Enhanced Edition

Compatibility

XDC 2022:

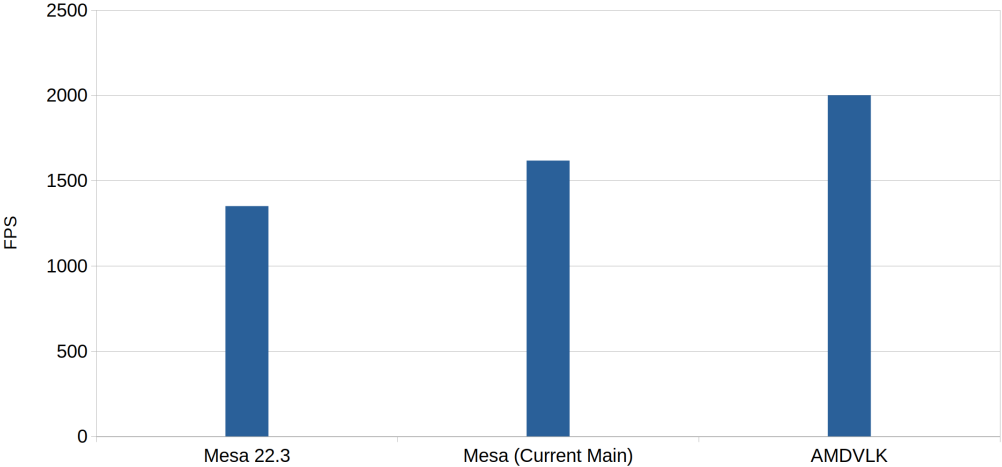
- Quake II RTX
- Control
- Deathloop
- Resident Evil: Village
- Metro Exodus Enhanced Edition

XDC 2023:

- General expectation is that new stuff works
- Some known issues (Witcher 3, Cyberpunk 2077 can hang currently)
- Something else broken?
 - <https://gitlab.freedesktop.org/mesa/mesa/-/issues/new>

Performance

Raytracing Reflections Sample (4K, 6700 XT)



Future Work

- Make remaining games work
- Make remaining games work **fast**
- Explore more sophisticated shader call techniques
 - Shader Execution Reordering?
- Make the monolithic path work with pipeline libraries
- Take advantage of RDNA3 features
- Build higher quality BVHs
- AS Updates
- Tons of microoptimizations

Thank You!

Questions?

