Ray-tracing in Vulkan pt. 2

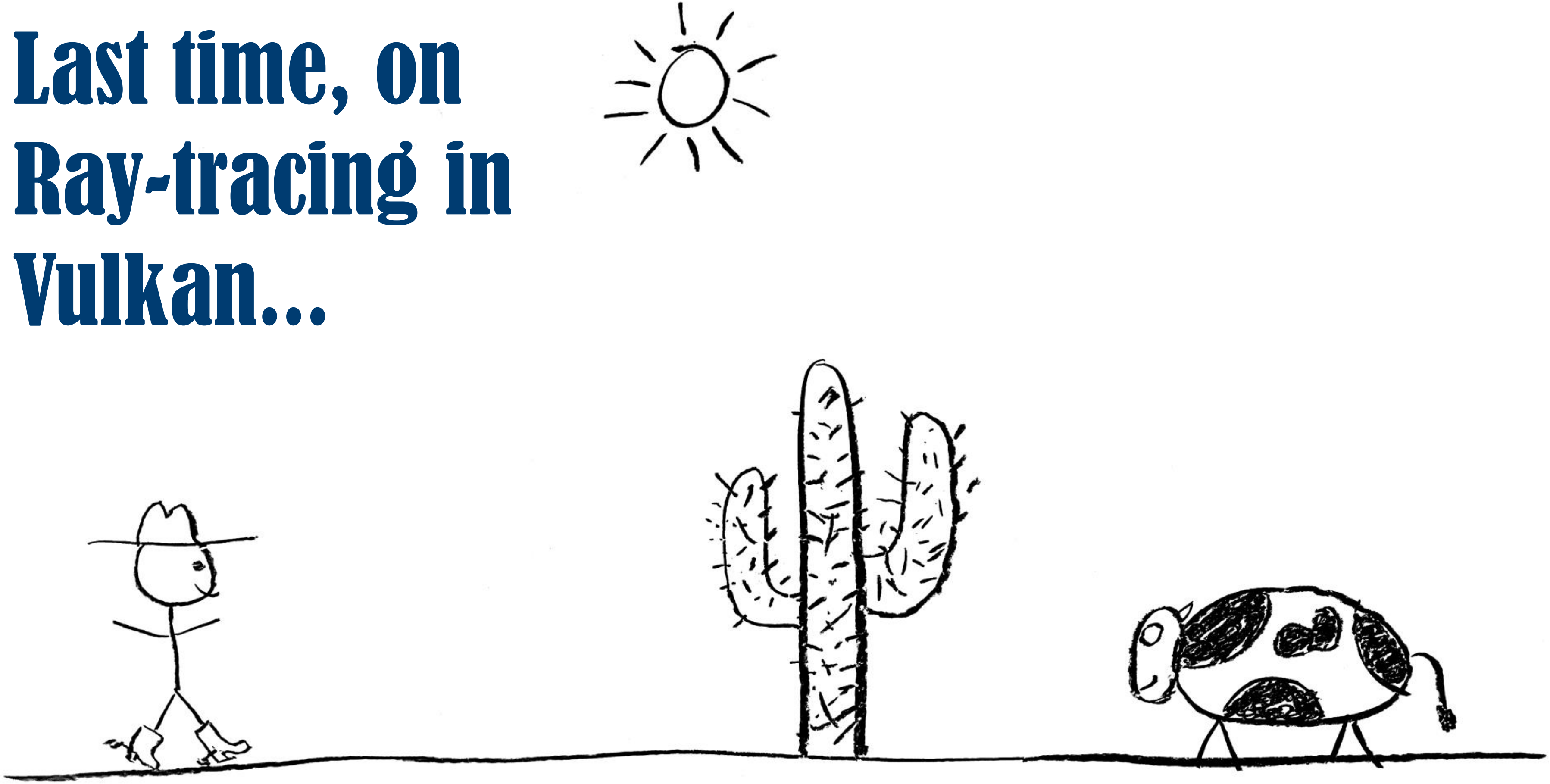
A look at the Intel Vulkan ray-tracing implementation

Jason Ekstrand, XDC 2021

Who am I?

- Name: Jason Ekstrand
- Employer: Intel
- First freedesktop.org commit: wayland/31511d0e dated Jan 11, 2013
- What I work on: Everything Intel but not OpenGL front-end
  - src/intel/*
  - src/compiler/nir
  - src/compiler/spirv
  - src/mesa/drivers/dri/i965
  - src/gallium/drivers/iris

Last time, on Ray-tracing in Vulkan…



Shader calls

Shader calls in Vulkan

A quick run-down:

- All possible callable shaders are provided as part of one mega-pipeline
- All callable shaders are provided via **S**hader **B**inding **T**ables (SBTs)
  - Filled with shader group handles queried from the pipeline
- A callable shader can be invoked from another shader via OpExecuteCallableKHR
- Data is passed between shaders via variables decorated with the CallableDataKHR storage class
- When the called shader completes, control returns to the calling shader

Bindless shaders in Intel HW

- COMPUTE_WALKER has a **B**indless **T**hread **D**ispatch (BTD) mode
  - Incompatible with shared memory (no real "local workgroup" concept)
  - Causes it to generate a thread ID for each invocation
- Each callable shader is represented by a BINDLESS_SHADER_RECORD
  - Contains the start address, SIMD width, and local data offset
- Bindless shaders are invoked using the BTD_SPAWN message
  - Takes a pointer to a BINDLESS_SHADER_RECORD and a thread ID
- Threads are terminated with a BTD_SPAWN with the "retire" bit set

Bindless shaders in Intel HW

The bindless shaders themselves are pretty bare-bones:

- Thread ID
- Global data pointer (passed into BTD_SPAWN)
- Local data pointer (BSR address + local data offset in BSR)

It's up to you to build whatever you want out of these primitives

Mapping Vulkan to Intel hardware

- Hardware has no call stack or BTD_RETURN message
- We have to manage the stack manually:
  - Spill and fill around shader calls
  - Stash return BSR addresses and call parameters on the stack
- Shader calls use BTD_SPAWN to launch the child shader
- Return is implemented as BTD_SPAWN to launch the return shader
  - Invokes a whole new shader (it can't just jump back)

Mapping Vulkan to Intel hardware

```
void main() {
   /* foo */
   executeCallableEXT(/* child() */);
   /* bar */
   return; /* End the thread */
}

void child() {
   /* baz */
   return;
}
```

```
void main0() {
   /* foo */
   __push_stack();
   __push_BTD_addr(/* main1() */);
   __btd_spawn(/* child() */);
}

void main1() {
   __pop_stack();
   /* bar */
   __btd_spawn(RETIRE);
}

void child() {
   /* baz */
   __btd_spawn(__pop_BTD_addr());
}
```

In functional programming, this is called **C**ontinuation-**P**assing **S**tyle (CPS)

What if the call happens inside a nested loop?

Mapping Vulkan to Intel hardware

- nir_lower_shader_calls() transforms the shader to continuation-passing style
- Requires two new NIR intrinsics:
  - rt_execute_callable: Takes a SBT index and a pointer to the payload
  - rt_resume: Marks a resume point
    - Comes before any instructions in the resume shader that access the stack (constants, undefs, etc. may come before it.)
  - Both intrinsics have a pair of constants:
    - Call index: Indicates which call in the original shader it pertains to
    - Stack size: Size of the stack (in bytes) at that shader call

Mapping Vulkan to Intel hardware

- brw_nir_lower_rt_intrinsics() lowers the core NIR intrinsics to Intel ones
- rt_execute_callable:
  - Place return shader BSR address and payload address on the stack
  - Modify the per-thread stack stack offset (push the stack)
  - Insert BTD_SPAWN to start the callable
  - nir_jump_halt to the end in case we're inside a loop
- rt_resume:
  - Read and modify the per-thread stack offset (pop the stack)

Callable Shader I/O

Callable shader I/O

Callable/calling shaders have three new types of I/O:

- CallableDataKHR:
  - A block of data which can be passed to a callable shader.
  - A pointer to this block is passed to OpExecuteCallableKHR.

Callable shader I/O

Callable/calling shaders have three new types of I/O:

- CallableDataKHR:
- IncomingCallableDataKHR:
  - Only exists in the called shader
  - Aliases the CallableData block passed to the OpExecuteCallableKHR
  - Can be read to get data from the caller or written to pass data back.

Callable shader I/O

Callable/calling shaders have three new types of I/O:

- CallableDataKHR:
- IncomingCallableDataKHR:
- ShaderRecordBufferKHR:
  - A tiny UBO placed after the shader handle in the SBT

Callable shader I/O lowering

First, each variable is lowered:

- CallableDataKHR:
  - Converted to nir_var_shader_temp
- IncomingCallableDataKHR:
  - Each var deref is replaced with the payload pointer stored on the stack
- ShaderRecordBufferKHR:
  - Each var deref is replaced with the local data pointer in the payload

Then run nir_lower_explicit_io()

Ray-tracing

Ray-tracing

Ray-tracing works the same as callable shaders if you know the mappings:

- Ray-gen -> callable or compute shader depending on $DETAILS
- Any-hit, closest-hit, miss, and intersection shaders -> callable (bindless)
- RayPayloadKHR -> CallableDataKHR
- IncomingRayPayloadKHR -> IncomingCallableDataKHR
- OpTraceRayKHR sort-of maps to OpExecuteCallableKHR with extra stuff

Ray-gen shaders

- Ray-gen shaders are specified through the API as a 1-element SBT
  - Dispatched with BTD_SPAWN like any other bindles shader
- vkCmdTraceRaysKHR() launches a "trampoline" compute shader:
  - Loads the ray-gen handle
  - Sets up the per-thread scratch space
  - Launches the ray-gen shader with BTD_SPAWN
- If the pipeline contains only a few ray-gen, the trampoline can be avoided

OpTraceRay

- Similar to OpExecuteCallable
  - Same shader splitting and conversion to continuation-passing style
  - Same I/O lowering
- Except it communicates with the ray-tracing hardware:
  - Sets up the initial HW RayData structure used for tracing
    - Acts as an interator for the ray-tracing operation
    - Points to the root of the BVH
    - Contains ray origin/direction, hit shader tables, miss shader pointer, etc.
  - Calls TRACE_RAY to invoke the ray-tracing hardware

Hit and miss shaders

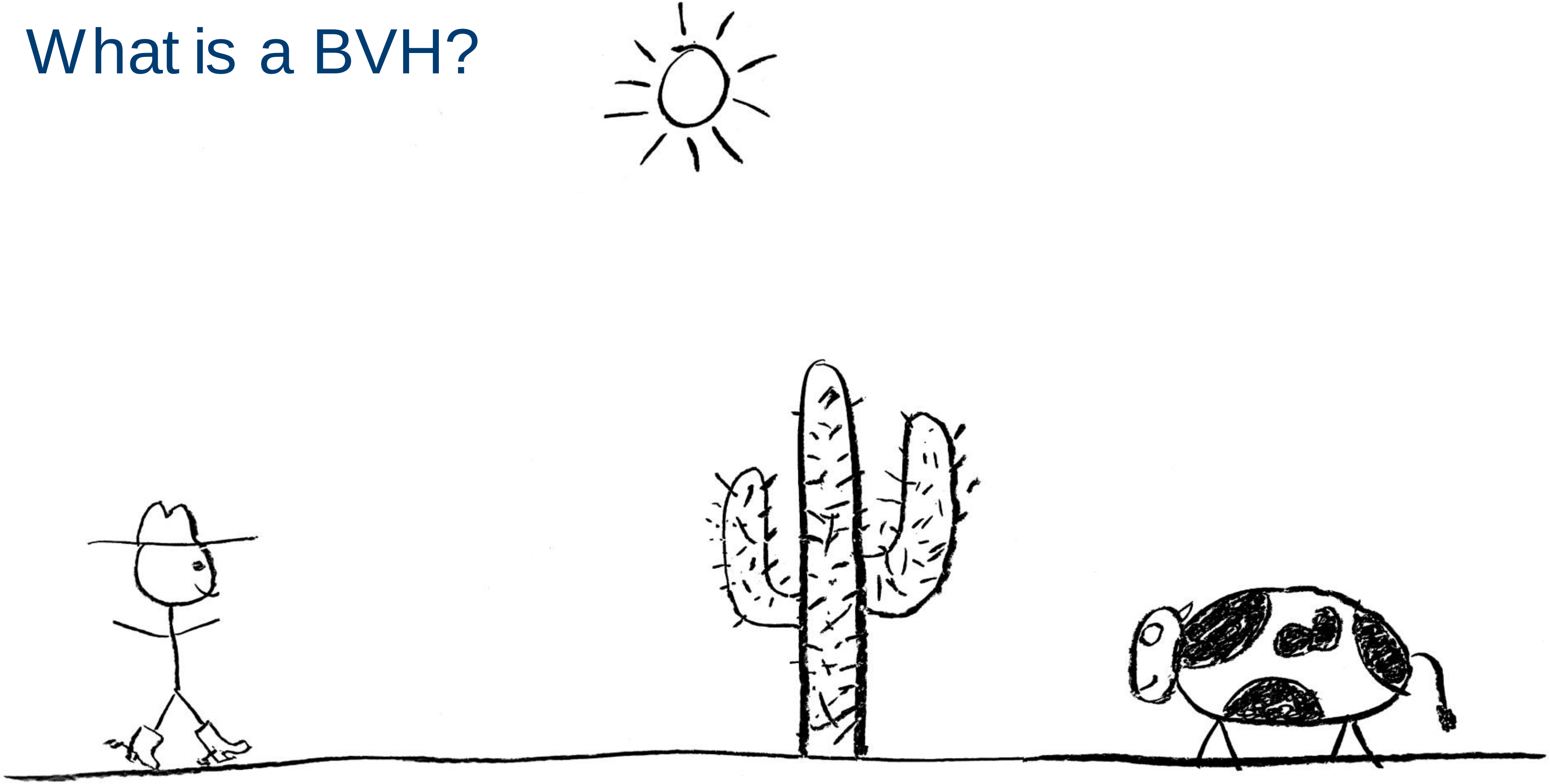
- Hit and miss shaders are just callable shaders
  - Built-ins such as RayOriginKHR come from inspecting the RayData iterator
  - Built-ins such as RayGeometryIndexKHR come directly from the BVH
- Any-hit shaders don't normally return up the stack
  - The ray-tracing hardware may call any number of them
  - They may return if OpTerminateRayKHR is called
- Closest-hit and miss shaders return up the stack
- A "trivial return" shader which is invoked if no miss or closest-hit

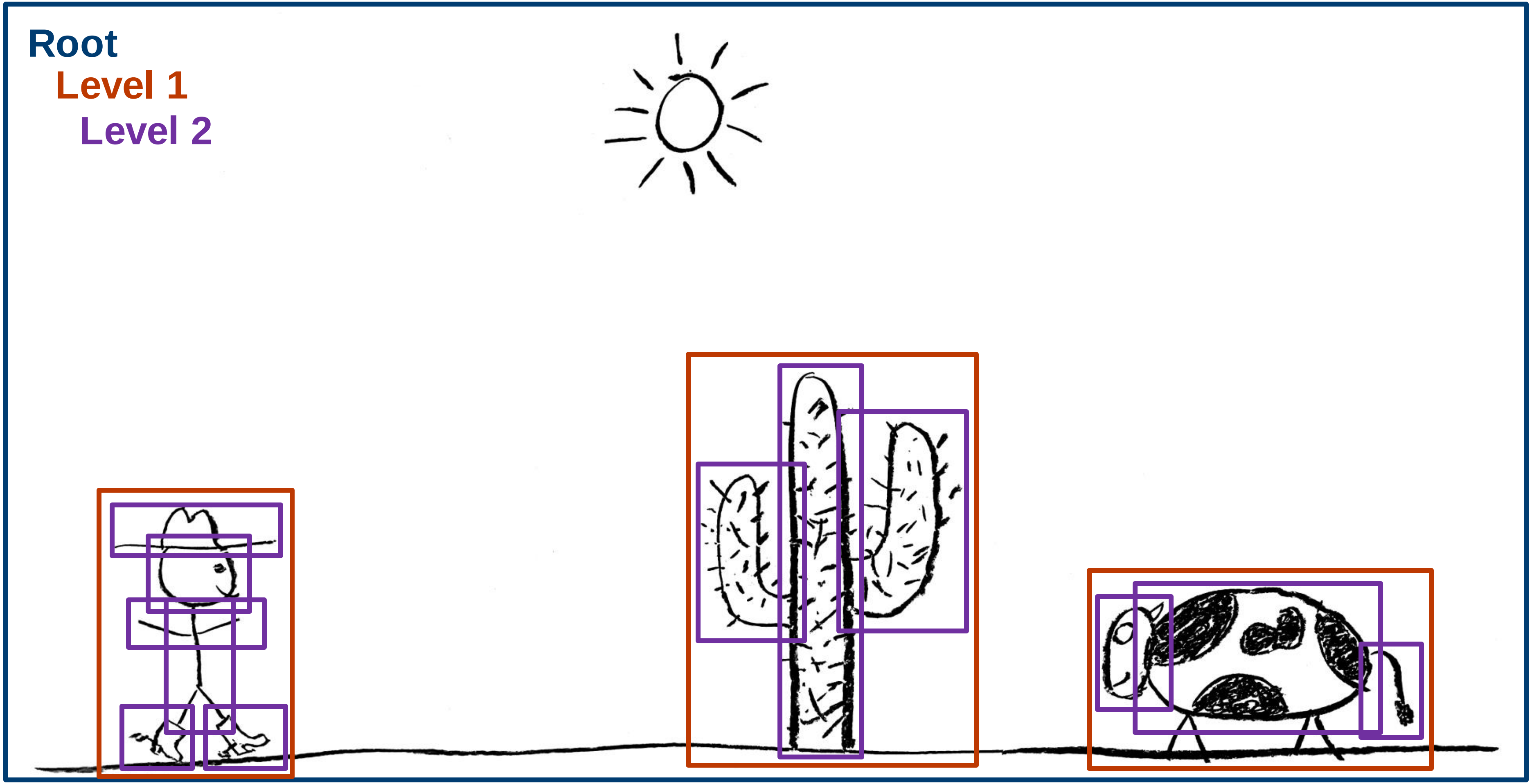
Intersection shaders

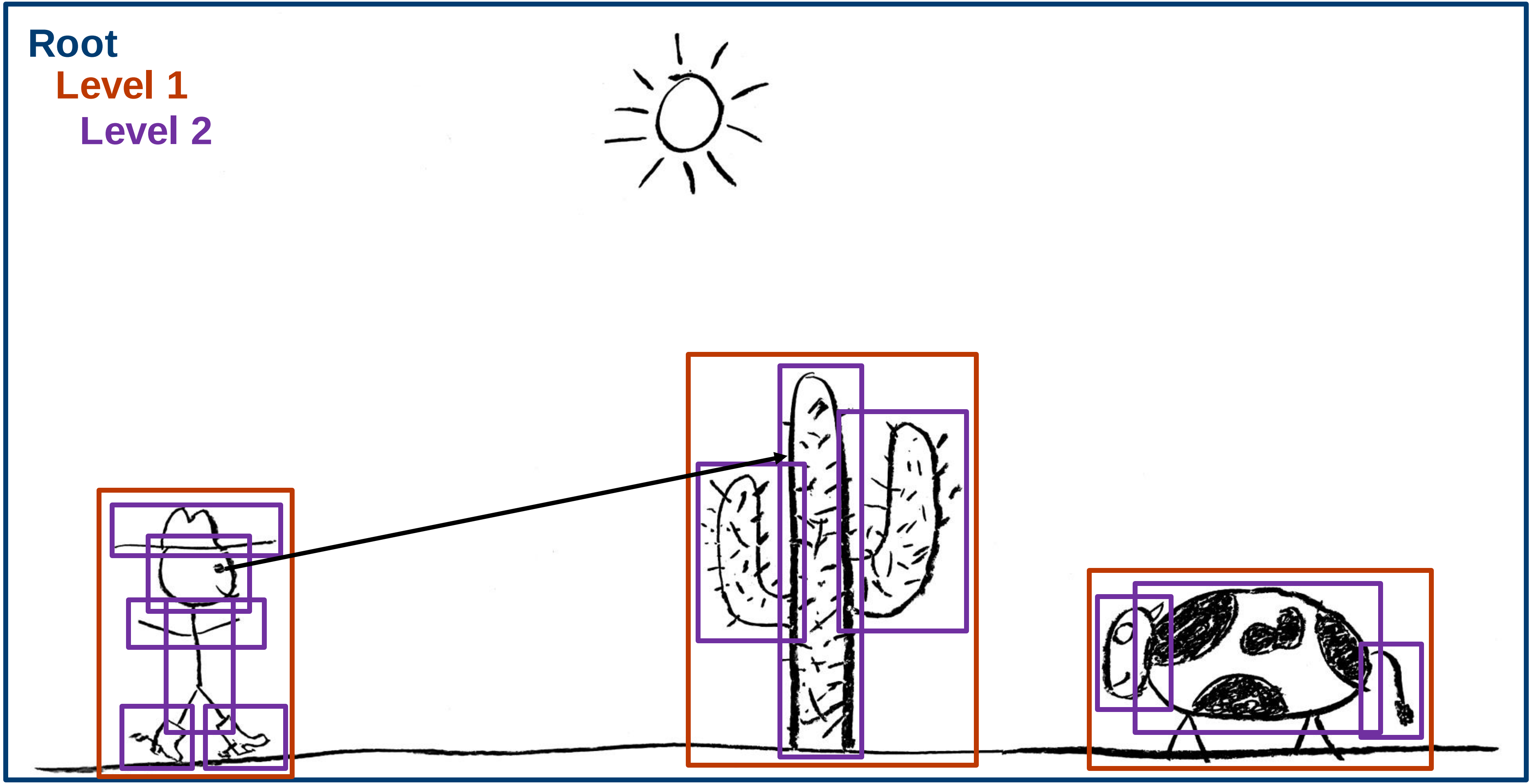
- Intersection shaders don't exist, not really....
- Intersection shaders are just any-hit shaders for AABBs
  - OpReportIntersection sets up a hit and calls the client any-hit shader
  - Depending on the results of any-hit shaders, it reports or ignores the hit
- Any-hit shaders are inlined into the corresponding intersection shader
  - Vulkan requires they always be paired
  - brw_nir_lower_intersection_shader()

BVH Building

What is a BVH?



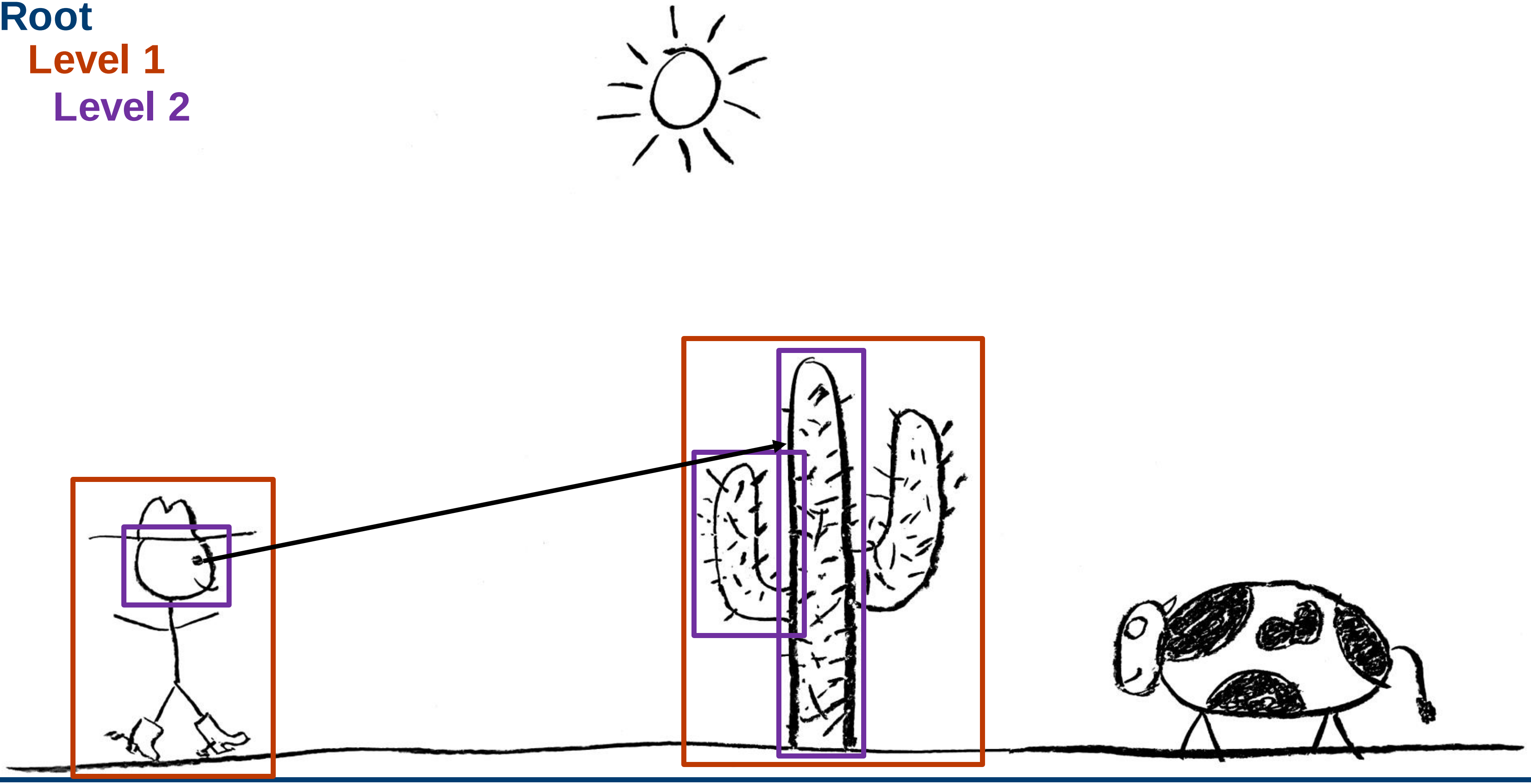

Root
Level 1
Level 2


Root
Level 1
Level 2

Root

Level 1

Level 2



CPU building with Embree

- For driver bring-up, we built our BVHs on the CPU with Embree
- Embree is an open-source ray-tracing framework from Intel:
  - https://github.com/embree
- CPU BVH building is a three-step process from the driver PoV:
  - Parse Vulkan BVH data into boudning boxes
  - Invoke Embree to sort it into a BVH
  - Read the Embree BVH and write out the HW BVH format

CPU building with Embree

- CPU BVH building has a lot of advantages for driver bring-up:
  - Easier to see what's going on and debug
  - Lets you bring up ray-tracing pipelines and GPU BVH building separately
  - GPU BVH building on a HW simulator takes forever
- Current implementation isn't production-ready
  - Embree spawns threads behind the client's back
  - Doesn't tie into VK_KHR_deferred_operation
  - Not optimized

No one cares about CPU builds; DXR doesn't have them

We're not going to talk about BVH building algorithms

Instead, we'll focus on dispatching BVH building kernels

BVH building kernels

- BVH building kernels are written in OpenCL C
  - Makes them easier to develop/debug
  - Same kernels used for Vulkan and D3D12
- Compiled at build-time and embedded in the drivery binary
  - We wrote a little intel_clc build tool
  - Goes OpenCL C -> SPIR-V -> NIR -> intel back-end
  - Based on the OpenCL work from Karol, Jesse, Boris, etc.
- Vulkan driver now sort-of understands the OpenCL dispatch model

BVH building Meta-kernels

- A single BVH build requires multiple kernels:
  - Init, parse API data, sorting algorithms, BVH output
  - May be dispatched with different workgroup sizes
  - Dispatch sizes, number of dispatches, etc. may not be static
    - vkCmdBuildAccelerationStructuresIndirectKHR

BVH building Meta-kernels

- A single BVH build requires multiple kernels:
- We developed a new meta-kernel language called GRL
  - Executes on the command streamer
  - Read/write values to/from memory
  - Basic arithmetic
  - Control-flow
  - Can launch a kernel (possibly with indirect dispatch)

BVH building Meta-kernels

- A single BVH build requires multiple kernels:
- We developed a new meta-kernel language called GRL
- GRL parser currently written in Python
  - Basic parser using PLY (Python Lex-Yacc)
  - Basic optimizer (mostly copy-prop and DCE)
  - Outputs C with mi_builder commands

BVH Building Meta-meta-kernels

- Someone has to figure out how to launch meta-kernels:
  - Select BVH building algorithm
  - Compute sizes and allocate memory
    - Kernels and meta-kernels have inputs and need scratch memory
  - Launch the right meta-kernels in the right order
    - It's not just one meta-kernel per build. That would be too easy!

Are we having fun yet?

BVH Building Meta-meta-meta-kernels?

Alt-text: "This is the reference implementation of the self-referential joke."

https://xkcd.com/917/

Open Questions

- Can we better share code with Windows?
  - We do share the OpenCL C kernel source and GRL files
  - Different GRL file parsers and meta-kernel launch code:
    - Windows is C++-based with a big templated launcher system
    - Effectively duplicates mi_builder but more complex
    - I wanted something simpler which re-used mi_builder
  - Every time we pull new OpenCL C and GRL files, we get divergence
    - This is bad....

Open Questions

- Can we better share code with Windows?
- Can we put more in the GRL files themselves?
  - Might let us share memory allocation and launch algorithms
  - Would come at the cost of GRL getting more complex

Open Questions

- Can we better share code with Windows?
- Can we put more in the GRL files themselves?
- Can we share code with RADV?
  - Ideally, we'd like to, obviously
  - Can AMD do the command streamer stuff GRL requires?
  - How do we abstract binary BVH formats?
  - Should RADV just use Intel BVHs?
    - AMD's hardware design probably makes this possible

Open Questions

- Can we better share code with Windows?
- Can we put more in the GRL files themselves?
- Can we share code with RADV?
- Should we compile GRL files to NIR?
  - Have a NIR back-end that generates MI commands via mi_builder
  - I really, really, really wish this were a joke....
  - If we're sharing with RADV, it might be a good idea

Questions?

The Intel open-source Linux 3D driver team is hiring!
Talk to me (jekstrand) on IRC if you're interested.

intel®