Ray-tracing in Vulkan

A brief overview of the provisional VK_KHR_ray_tracing API

Jason Ekstrand, XDC 2020

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

Vulkan ray-tracing history:

- On March 19, 2018, Microsoft announced DirectX Ray-tracing (DXR)
- On September 19, 2018, Vulkan 1.1.85 included VK_NVX_ray_tracing for ray-tracing on Nvidia RTX GPUs
- On March 17, 2020, Khronos released *provisional* cross-vendor extensions:
  – VK_KHR_ray_tracing
  – SPV_KHR_ray_tracing
- Final cross-vendor Vulkan ray-tracing extensions are still in-progress within the Khronos Vulkan working group

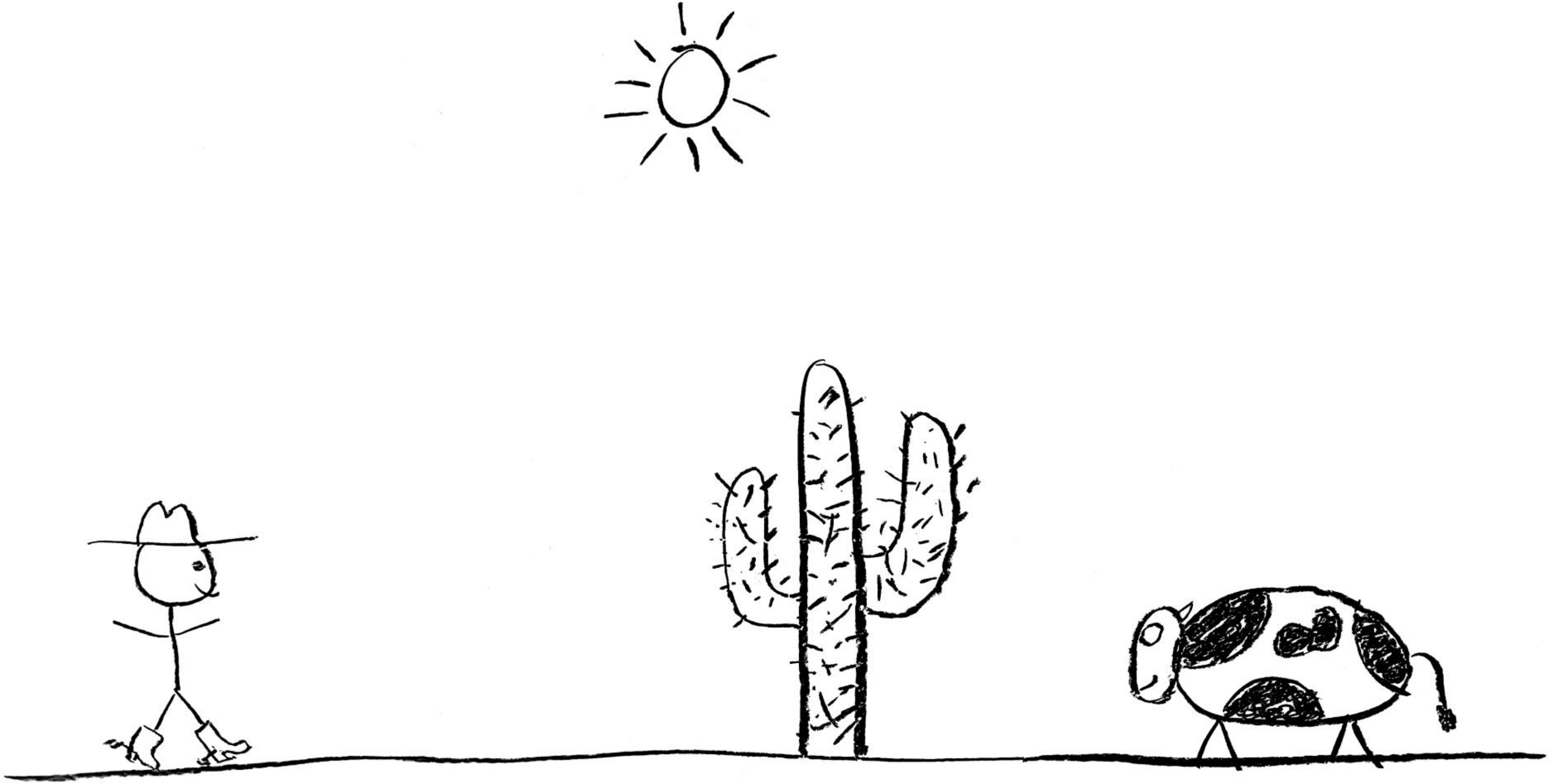
Overview:

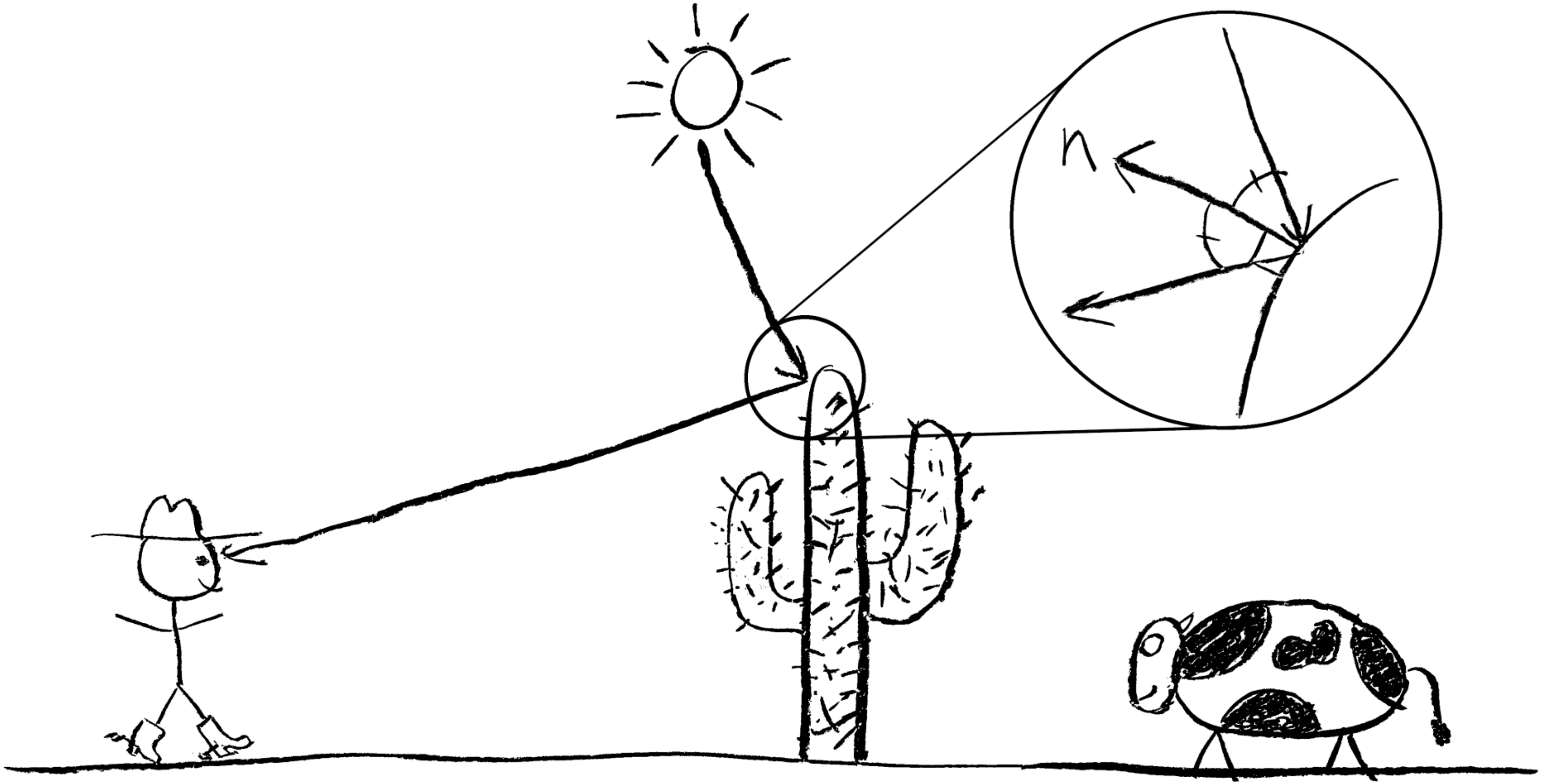
My objective with this presentation is mostly educational

- Quick overview of ray-tracing concepts
- Walk through how it all maps to the Vulkan ray-tracing API
- Focus on the *provisional* VK/SPV_KHR_ray_tracing extension
  - There are several details that will likely be different in the final extension
    - None of that is public yet, sorry.
  - The general shape should be roughly the same between provisional and final
- Not going to discuss details of ray-tracing on Intel GPUs

What is ray-tracing?

All 3D rendering is a simulation of physical light

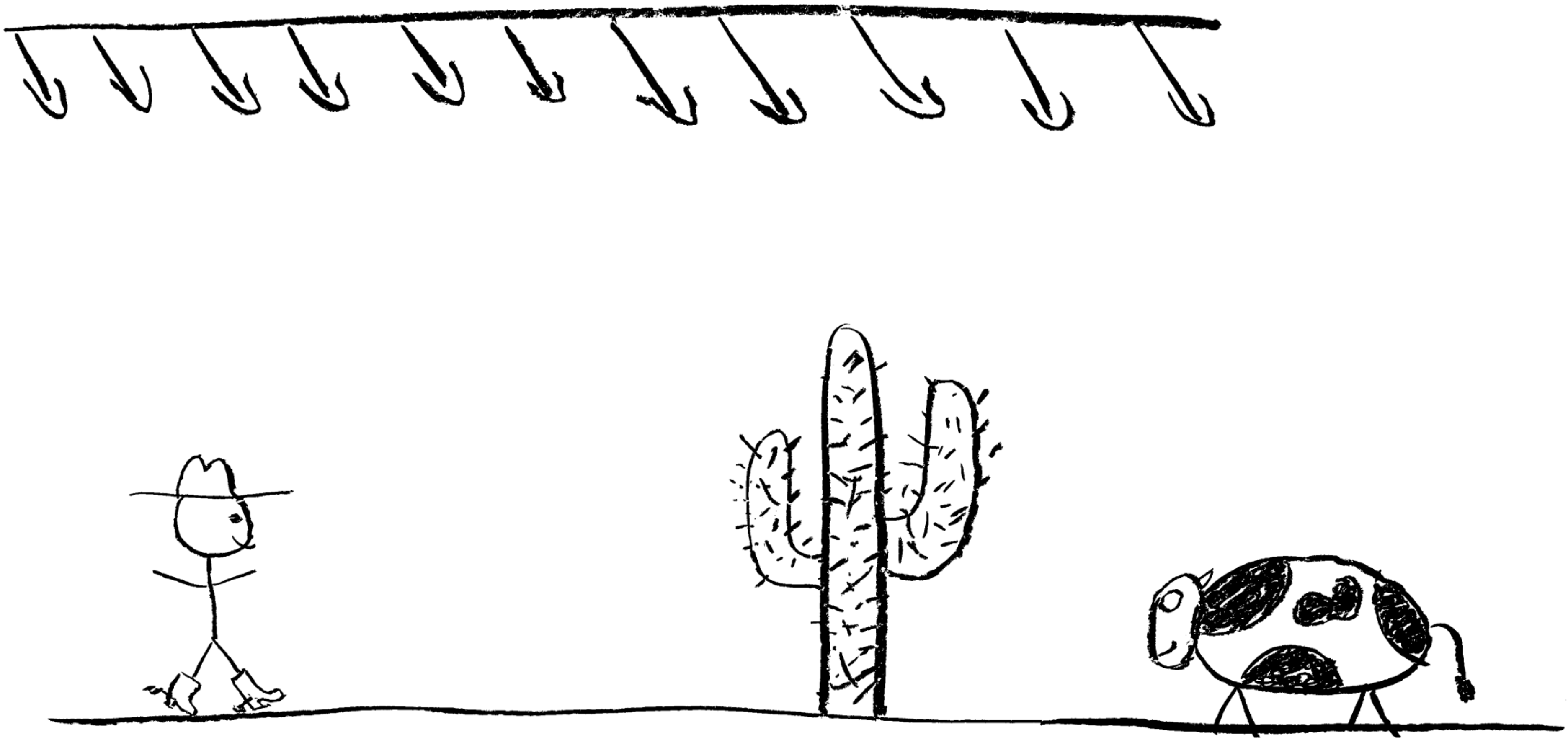



n

Why don't we do all 3D rendering this way?

The primary problem here is wasted rays

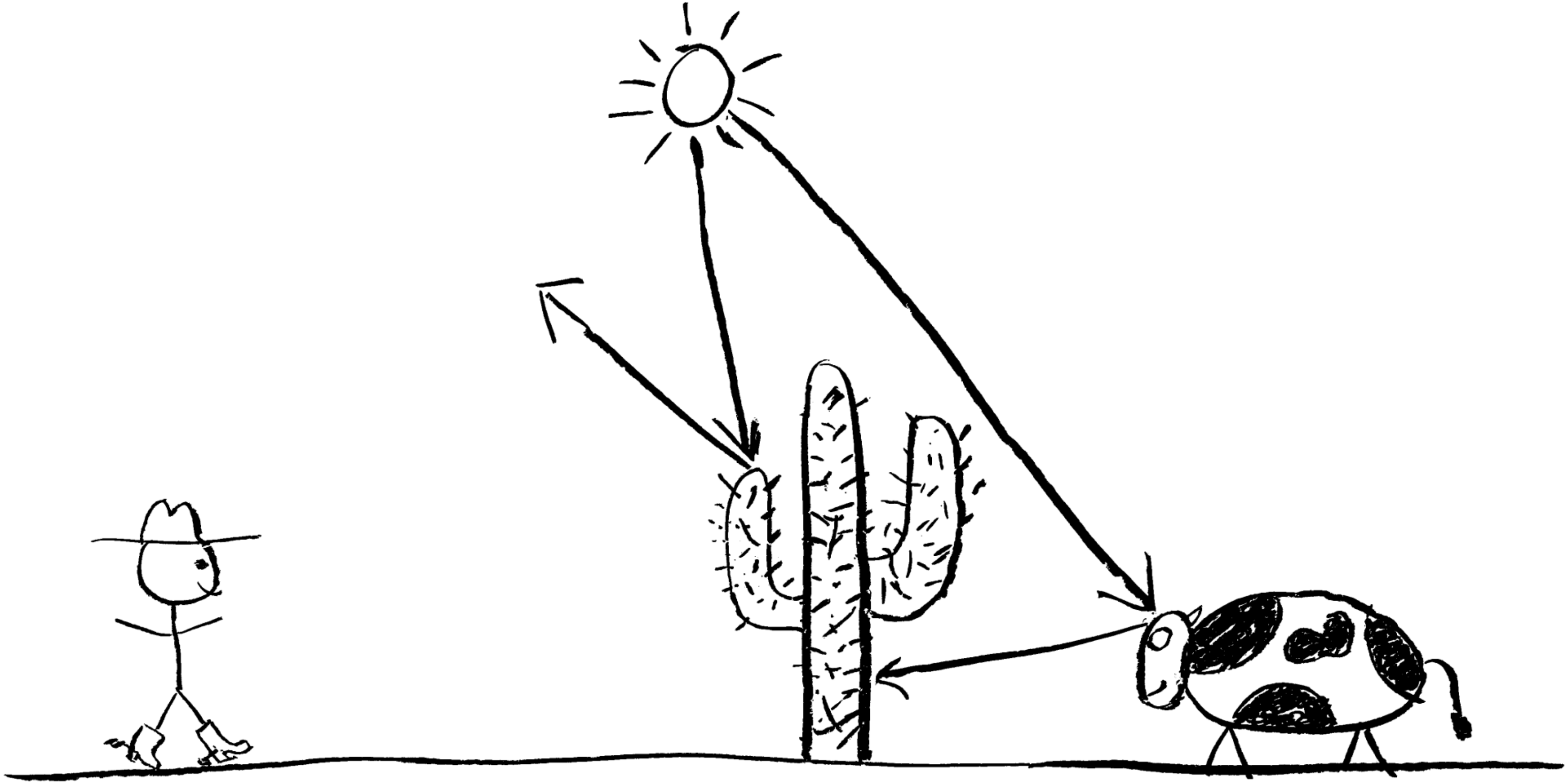
- The chances of a random photon from the sun hitting your scene is tiny
  - About 1 in every 2.1×10^9 photons from the sun even hit the earth
  - About 1 in every 4.5×10^{10} of those will hit the 100ft (30m) radius of your scene
  - The sun emits about 10^{45} photons per second
    - You'll never simulate that!
  - We can avoid this by approximating the sun as a plane light source

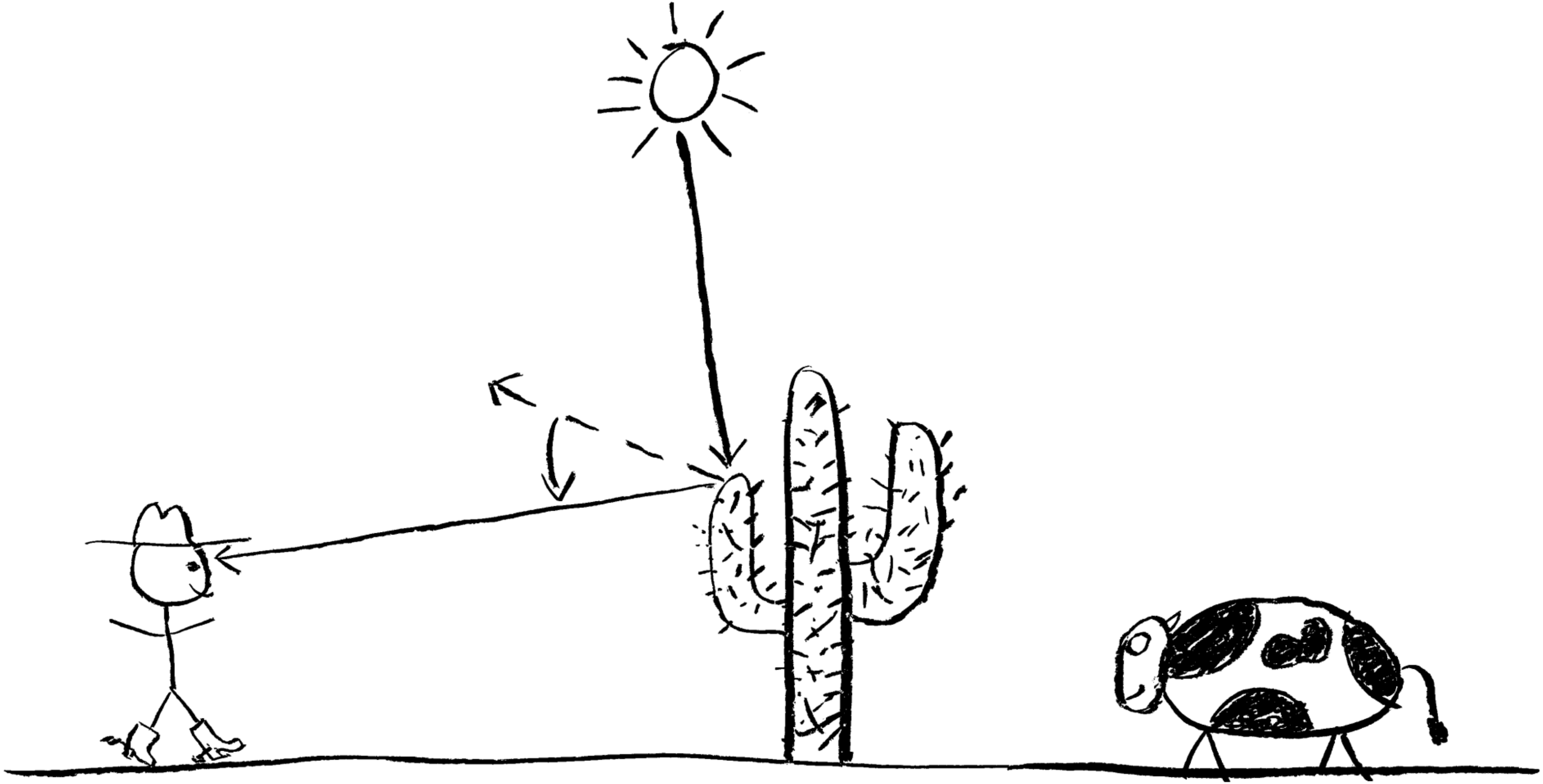


Why don't we do all 3D rendering this way?

The primary problem here is wasted rays

- The chances of a random photon from the sun hitting your scene is tiny
  - We can avoid this by approximating the sun as a plane light source
- The chances of a random photon bouncing off an object into the eye is tiny
  - It may bounce off in the wrong direction
  - It may run into some other object first
  - We can mitigate some of this by intentionally "targeting" our rays
    - Don't make reflected rays random
    - Reflect towards the eye/camera when possible



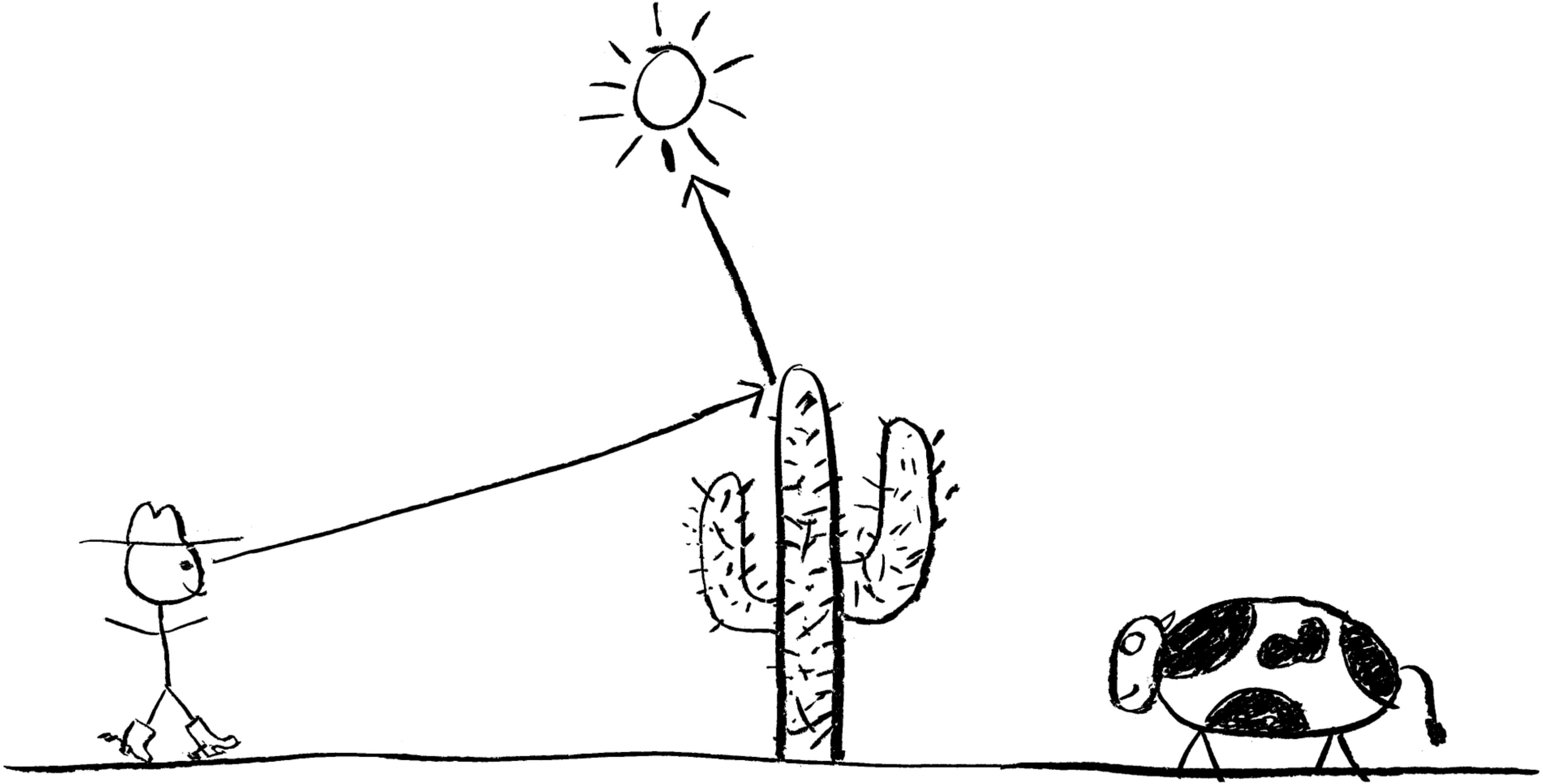


Why don't we do all 3D rendering this way?

The primary problem here is wasted rays

- The chances of a random photon from the sun hitting your scene is tiny
  – We can avoid this by approximating the sun as a plane light source
- The chances of a random photon bouncing off an object into the eye is tiny
  – We can mitigate some of this by intentionally "targeting" our rays
- But is there a better way?

Camera-to-light ray-tracing

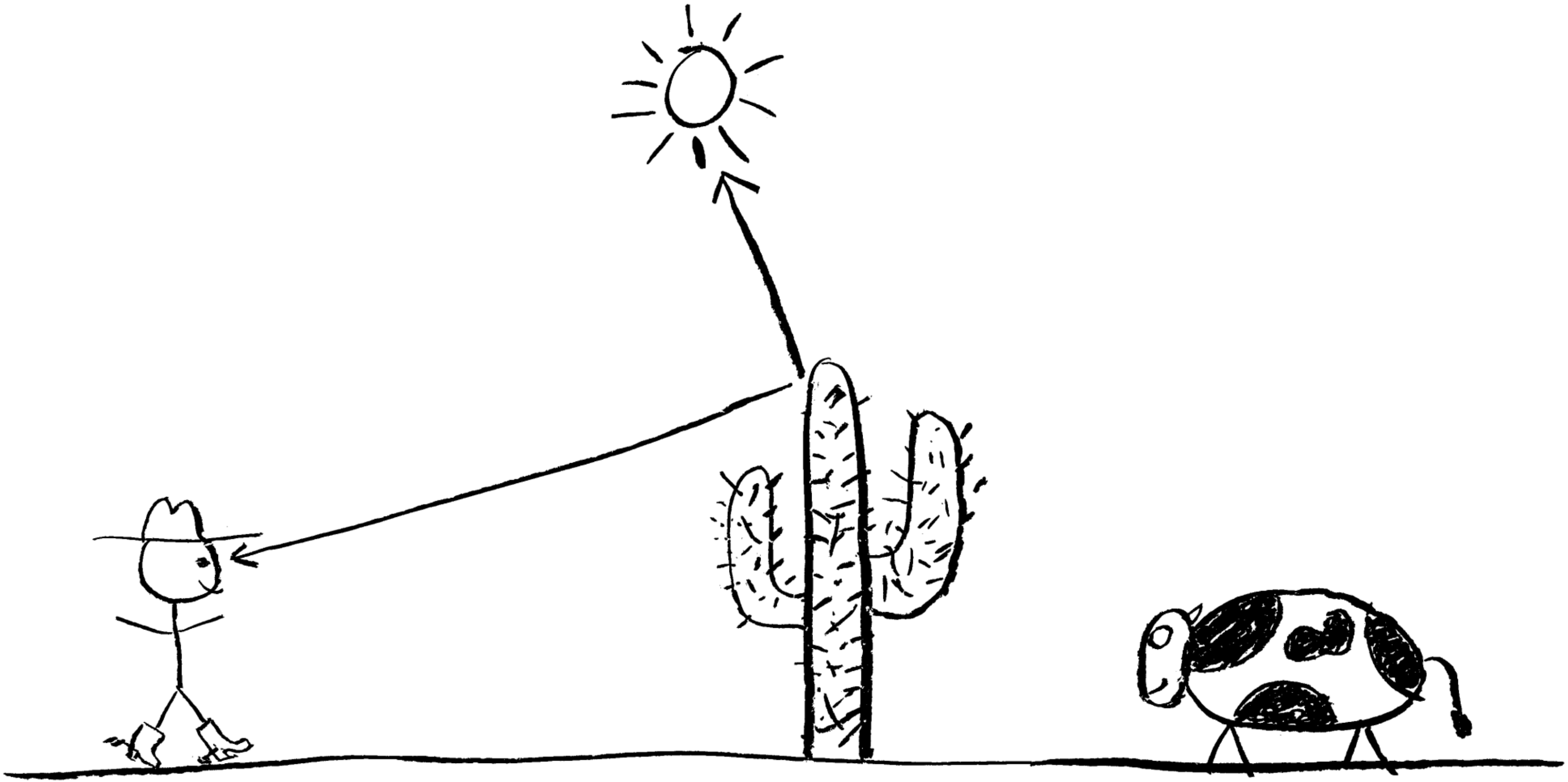


Advantages of camera-to-light ray-tracing

- Better precision
  - You can perfectly align every ray leaving the camera with a pixel
- Every ray leaving the camera (primary ray) is necessary
  - No wasted primary rays! They all hit the camera by definition
  - Every primary ray will either hit a visible object or background
- Secondary rays can still miss
  - The same targeting technique works; you target lights instead of the camera

Forward rendering

Forward rendering is ray-tracing from the perspective of the cactus



Forward rendering as ray-tracing

For each triangle that's part of the cactus:

- Rays are cast in the direction of the camera:
  - The geometry pipeline transforms geometry into camera space
  - The rasterizer determines which pixels (rays) intersect that triangle
- For each rasterized pixel, rays are cast towards lights
  - The fragment shader typically walks the list of lights and computes the ray direction and distance to each light to calculate the color
- Wasted rays still exist from overdraw and depth test fails

Ok, so it's not perfect. I tried, alright? :-P

Disadvantages of ray-tracing

- Geometry intersection from arbitrary origin points is expensive
  - Forward rendering transforms all the geometry so the camera is the origin
  - Ray-tracing invovles computing ray geometry intersections of arbitrary rays with any origin point and direction in 3D space
- Ray-tracing tends to be noisy
  - The "interesting" techniques with ray-tracing usually involve multiple secondary rays and are somewhat statistical in nature
  - You usually need a post-processing denoise filter
- Worse memory access patterns for vertex data etc.
  - Forward rendering is nicely streamed; ray-tracing involves lots of random access

Advantages of ray-tracing

Some things are easier and simpler when ray-tracing

- Shadows
  - In forward rendering, it typically requires a Z-pass per light
  - When ray-tracing, it's "does this ray hit anything on its way to the light"
- Global illumination (light reflecting off one object onto another)
  - Can be faked with SSAO etc. but requires multiple post-processing passes
  - When ray-tracing, reflect twice and allow tertiary rays
- Reflections
  - Impossible to get correct without ray-tracing

Accelerating ray-tracing with the Vulkan API

Accelerating ray-tracing with the Vulkan API

Ray-tracing in Vulkan has two primary parts:

- Acceleration structures
  - Hold all of the geometry for your entire scene
  - Allow acceleration of arbitrary ray geometry intersections
- Six new shader stages
  - Dispatch rays, handle hits and misses, and define procedural geometry
  - Lots of new system-values and intrinsics

It's basically a new 3D rendering API

Acceleration structures

Acceleration structures

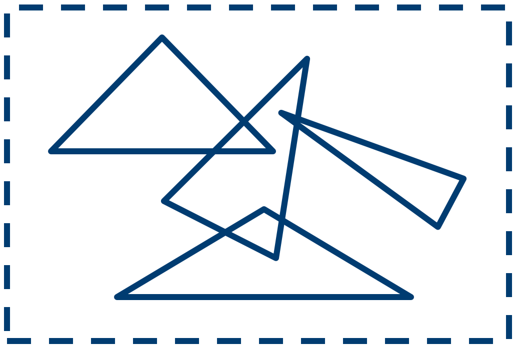
- A data structure to accelerate ray-geometry intersection
  - `VkAccelerationStructureKHR` is a memory-backed object like an image
  - Contain all of the geometry for an entire scene
  - Built from geometry data with new Vulkan commands:
    - `VkCmdBuildAccelerationStructureKHR()`
    - `VkCmdBuildAccelerationStructureIndirectKHR()`
    - `VkBuildAccelerationStructureKHR()` (for CPU builds)
  - Bound via descriptor sets and passed into `traceRayEXT()` in the shader
  - Vulkan doesn't specify how they work internally

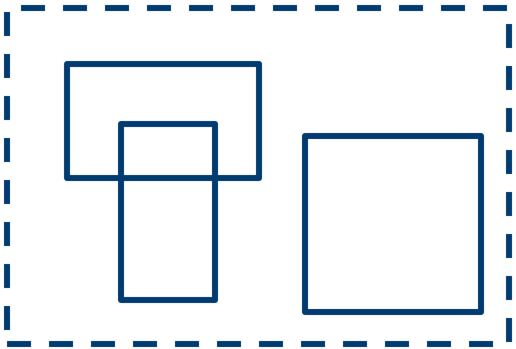
Acceleration structures

- A data structure to accelerate ray-geometry intersection
- There are two types of acceleration structures: top and bottom
  - Bottom-level acceleration structures contain actual geometry
  - Top-level acceleration structures contain bottom-level AS with transform matrices

Acceleration structures

- A data structure to accelerate ray-geometry intersection
- There are two types of acceleration structures: top and bottom
- Geometry comes in two types: triangles and AABBs
  - AABB = **A**xis-**A**ligned **B**ounding **B**ox, used for procedural geometry





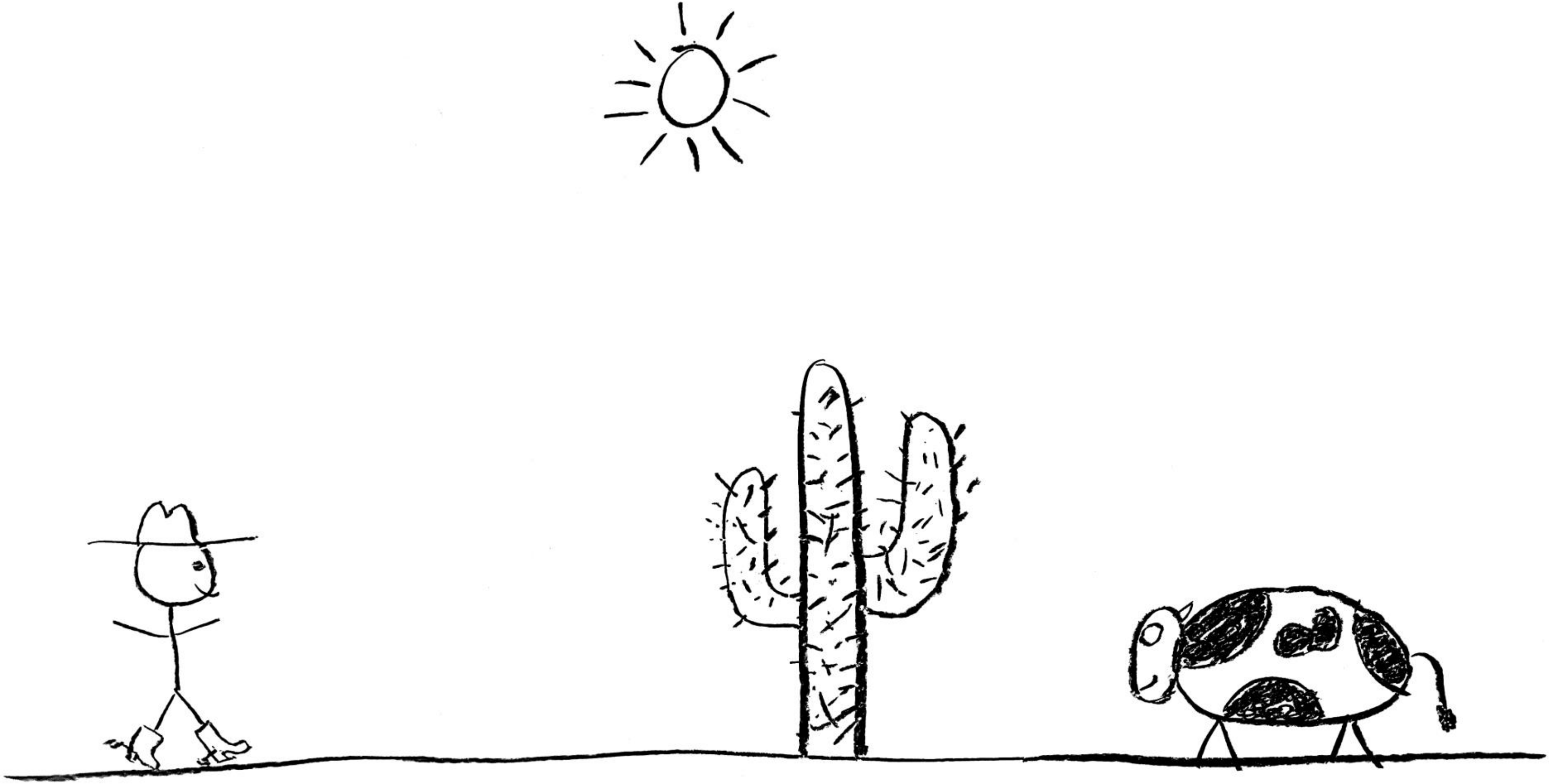
Acceleration structures

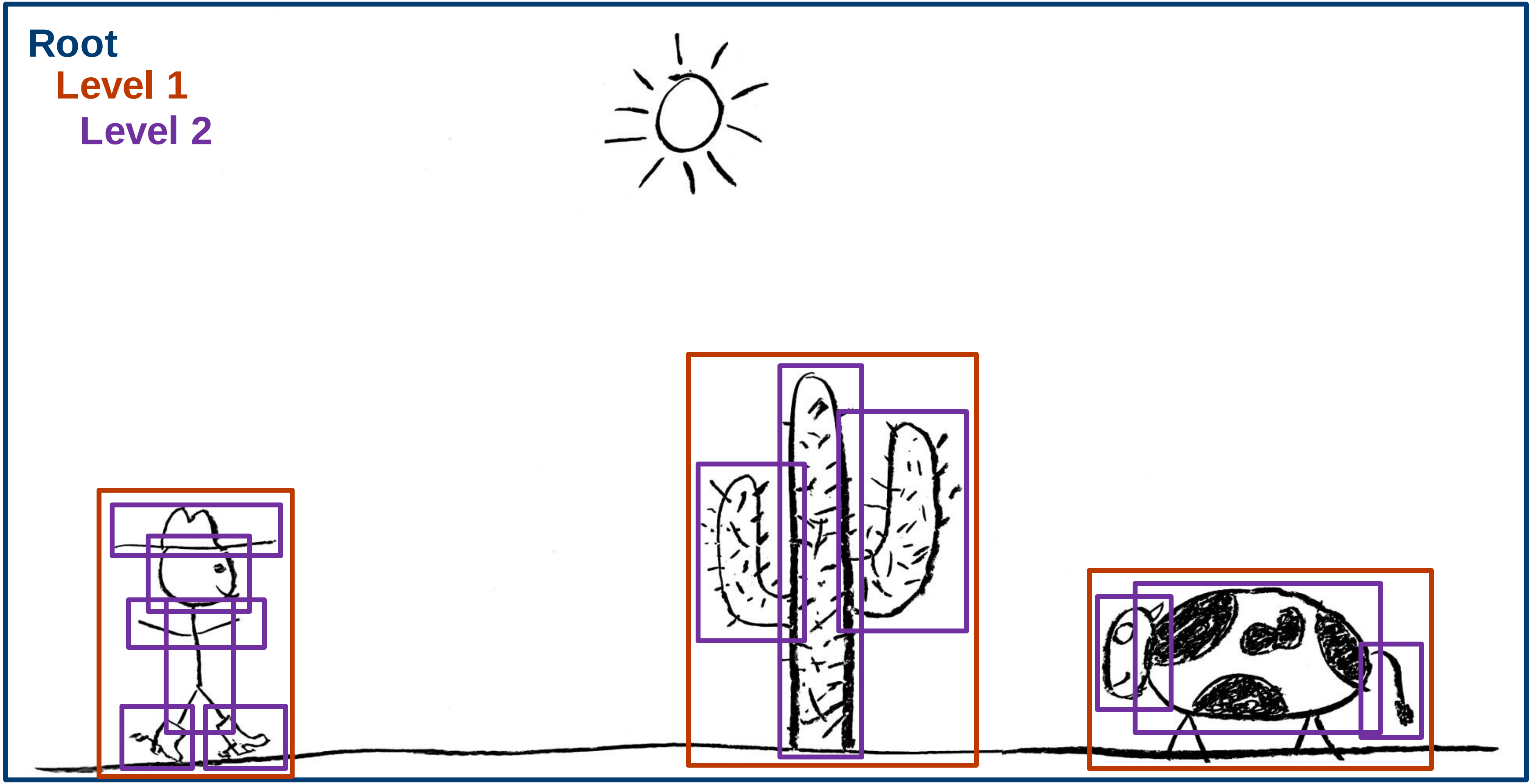
- A data structure to accelerate ray-geometry intersection
- There are two types of acceleration structures: top and bottom
- Geometry comes in two types: triangles and AABBs
- Bottom-level AS have a two-level hierarchy of geometry data:
  - An array of geometries, indexed by `gl_GeometryIndexEXT`
  - Each geometry contains an array of primitives of the same type (triangles or AABBs) indexed by `gl_PrimitiveID`
  - Only contains position data. Any other geometry input data (texture coordinates, colors, etc.) must be fetched by the shader based on those indices

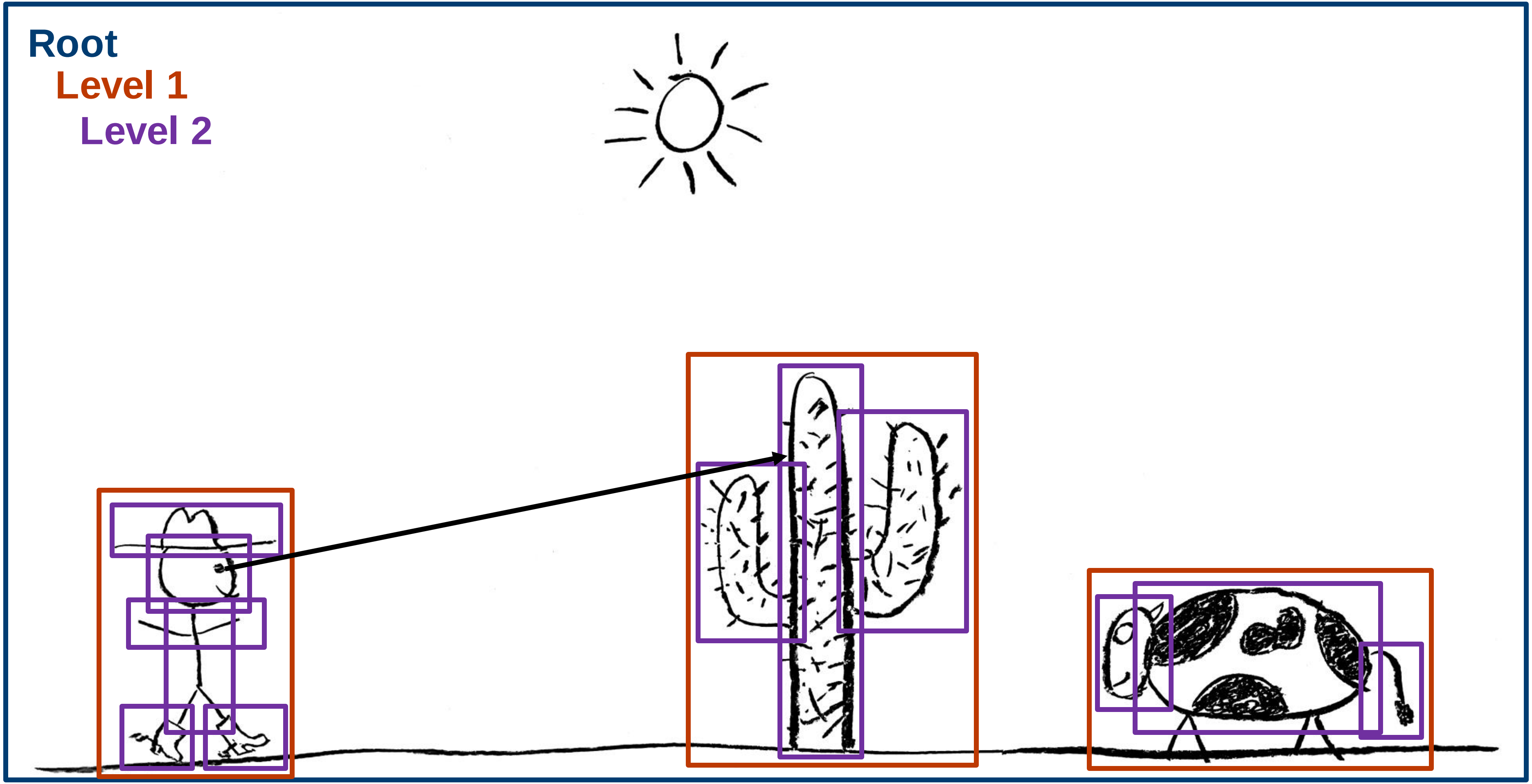
Implementing acceleration structures as a BVH

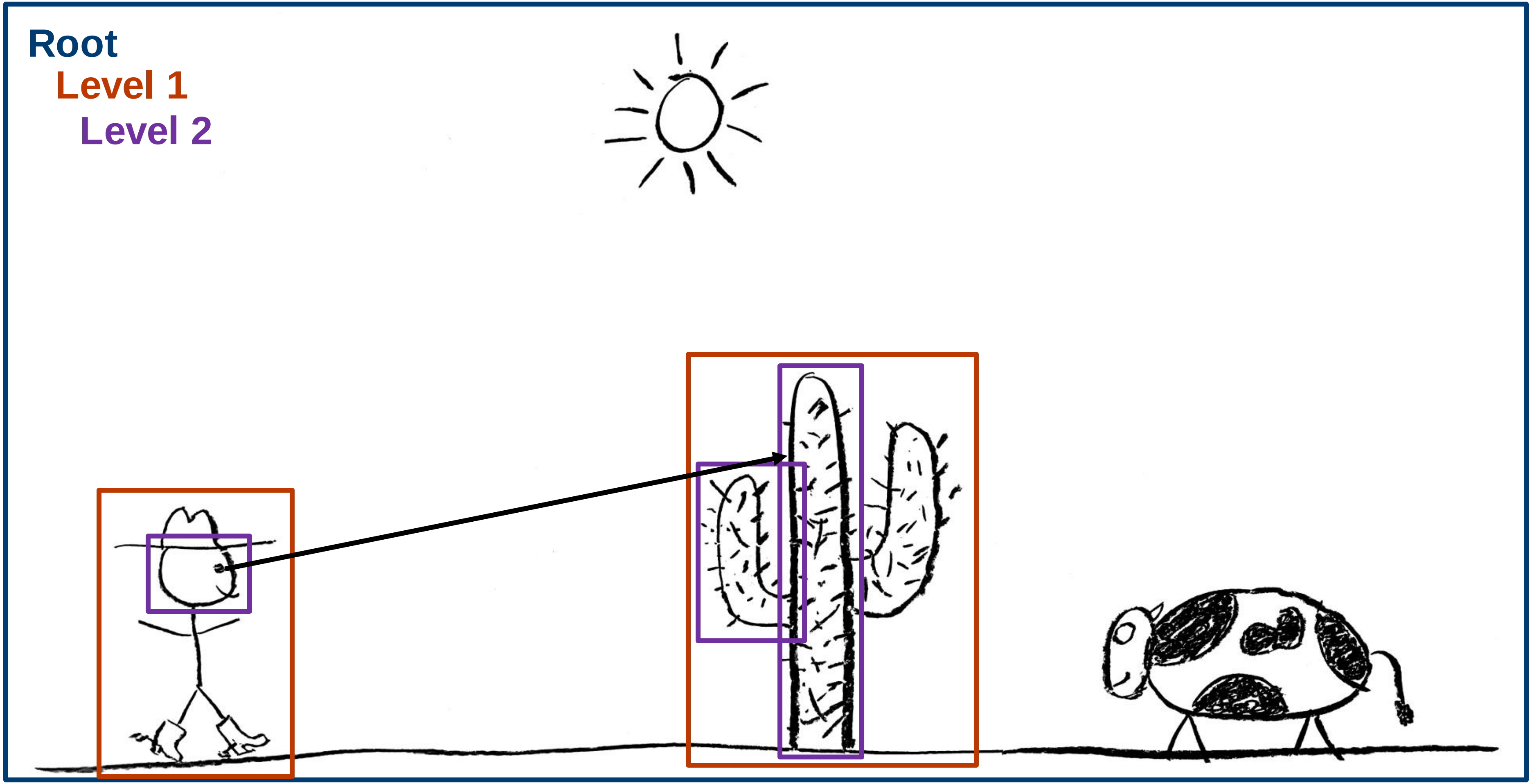
One possible AS implementaiton is a **B**ounding **V**olume **H**ierarchy (BVH)

- An N-ary tree data structure
- The leaves of the tree are primitives
- Each node of the tree has a bounding volume
  - Nodes are sorted to try and make the bounding volumes as small as possible
- When a ray is traced, the bounding volumes are used to quickly discard as much geometry as possible




Root
Level 1
Level 2


Root
Level 1
Level 2


Root
Level 1
Level 2

Ray-tracing shaders

Ray-tracing shaders

VK_KHR_ray_tracing adds six new shader stages:

- Ray generation: Invoked directly by `vkCmdTraceRaysKHR()`
- Any-hit: Invoked any time a primitive hit is detected
- Closest-hit: Invoked after ray traversal completes for the hit with lowest T
- Miss: Invoked after ray traversal completes if no hits were detected
- Intersection: Invoked when an AABB primitive is hit to determine actual intersections
- Callable: Can be invoked manually from any ray-tracing shader stage

Ray generation shaders

Ray generation shaders are the root of the shader "call tree"

- Invoked directly by `vkCmdTraceRaysKHR()`
- Look mostly like a compute shader:
  - Only one level of 3D dispatch grid (no local/global distinction)
  - No shared memory or barriers
  - Inputs: `gl_LaunchIDEXT` and `gl_LaunchSizeEXT`
- Typically call `traceRaysEXT()` to fire primary rays

Any-hit shaders

Any-hit shaders are invoked for each hit

- Ordering if invocations along the ray is not guaranteed
- Get information about the ray and the primitive via 19 built-ins including
  - `gl_LaunchIDEXT` and `gl_LaunchSizeEXT`
  - `gl_GeometryIndexEXT` and `gl_PrimitiveID`
  - `gl_HitTEXT`
- Can modify ray traversal
  - `ignoreIntersectionEXT()` ignores this hit
  - `terminateRayEXT()` terminates ray traversal early

Closest-hit shaders

The closest-hit shader is invoked at most once at the end of traversal

- Has the same 19 built-ins as any-hit shaders
- Hit information is for the hit with lowest T
- Typically where most "fragment" work such as texturing is done

Miss shaders

The miss shader is invoked at most once at the end of traversal

- Only invoked if no hits were accepted
  - No hits were reported or
  - Every any-hit shader called `ignoreIntersectionEXT()`
- Only has built-ins for launch and ray information (no hit)
- Could be used for, say, returning the sky color

Intersection shaders

Intersection shaders are invoked for every AABBs hit

- Used for procedural geometry (such as perfect spheres)
- Has most of the same built-ins as any-hit shaders
  - No actual hit information
- Reports hits via `reportIntersectionKHR()`
- If no intersections are supported, it's considered a miss

Callable shaders

Callable shaders are invoked manually by `executeCallable()`

- Only has the two launch built-ins
  - `gl_LaunchIDEXT` and `gl_LaunchSizeEXT`
- Can be used for whatever the client wants

Lost yet? If not, we'll fix that. ;-)

The ray-tracing call stack

- Ray-tracing involves a call stack
  - Any RT shader can call `traceRayEXT()` or `executeCallableEXT()`
  - Interseciton shaders can invoke any-hit via `reportIntersectionEXT()`
- Data is passed up and down the stack via special I/O variables
  - A `rayPayloadEXT` variable can be passed to `traceRayEXT()`
  - A matching `rayPayloadInEXT` variable can be declared in the called shader
  - Payloads are read/write in all shaders
  - For `executeCallable()` it's the same but called `callableDataEXT`
- Yes, it's a real stack; recursion is allowed

How do shader calls work in hardware?

This slide intentionally left blank :-)

Still following? Don't worry, that won't last long. ;-)

Shader binding tables

How do we solve the problem of switching shaders?

- Forward rendering passes typically use many pipelines
  - Typically used to handle different materials
- Ray-tracing pipelines can have arbitrarily many shaders of different stages
- Pipelines export "groups" of shaders:
  - A single ray generation shader
  - Any-hit + closest-hit for triangles
  - Interseciton + any-hit + closest-hit for AABBs
  - A single callable shader

Shader binding tables

- Each shader group has a 32B "handle"
  - Contains information required for dispatching those shaders
  - Handles fetched with `vkGetRayTracingShaderGroupHandlesKHR()`
- The client places those handles in a buffer
- The SBT buffers and strides are provided to `vkCmdTraceRaysKHR()`
  - All handles must come from the currently bound pipeline

Shader binding tables

```
void vkCmdTraceRaysKHR(
    VkCommandBuffer                         commandBuffer,
    const VkStridedBufferRegionKHR*         pRaygenShaderBindingTable,
    const VkStridedBufferRegionKHR*         pMissShaderBindingTable,
    const VkStridedBufferRegionKHR*         pHitShaderBindingTable,
    const VkStridedBufferRegionKHR*         pCallableShaderBindingTable,
    uint32_t                                width,
    uint32_t                                height,
    uint32_t                                depth);
```

Shader binding tables

Shaders are executed from the provided SBTs

- The ray generation shader is always the first one in the table
- For miss shaders, `traceRayEXT()` takes an SBT index
- For any-hit, closest-hit, and intersection shaders, the index is calculated:

```
instanceShaderBindingTableRecordOffset +
geometryIndex × sbtRecordStride +
sbtRecordOffset
```

- For callable shaders, `executeCallableEXT()` takes an SBT index

And that's about it! Simple, right?

Congratulations, you survived!

Questions?