# Vulkan for graphics research

## Tutorial Resources

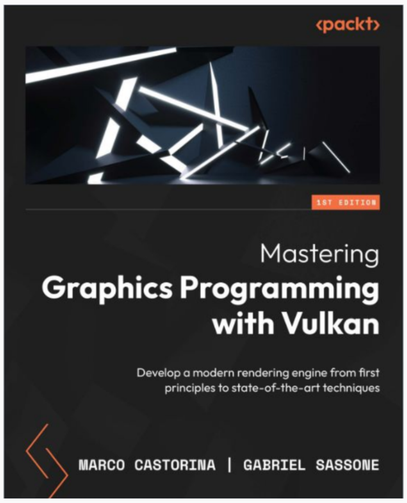
https://github.com/PacktPublishing/Mastering-Graphics-Programming-with-Vulkan

## Presenters

Marco Castorina first got familiar with Vulkan while working as a driver developer at Samsung. Later he developed a 2D and 3D renderer in Vulkan from scratch for a leading media-server company. He recently joined the games graphics performance team at AMD. In his spare time, he keeps up to date with the latest techniques in real-time graphics.

Gabriel Sassone is a rendering enthusiast currently working as a Principal Rendering Engineer at Multiplayer Group. Previously working for Avalanche Studios, where his first contact with Vulkan happened, where they developed the Vulkan layer for the proprietary Apex Engine and its Google Stadia Port. He previously worked at ReadyAtDawn, Codemasters, FrameStudios, and some non-gaming tech companies. His spare time is filled with music and rendering, gaming, and outdoor activities.

Co-authors of “Mastering Graphics Programming with Vulkan”



‹packt›
1ST EDITION
Mastering
Graphics Programming with Vulkan
Develop a modern rendering engine from first principles to state-of-the-art techniques
MARCO CASTORINA | GABRIEL SASSONE

## Topics

**Part 1 - Introduction to Vulkan**

Device and Queues

Memory Management and Resources

Render Passes, Descriptors, Pipelines

Surface and Swapchain

Command Buffers and Multi threading

Synchronization

**Part 2 - Advanced Features**

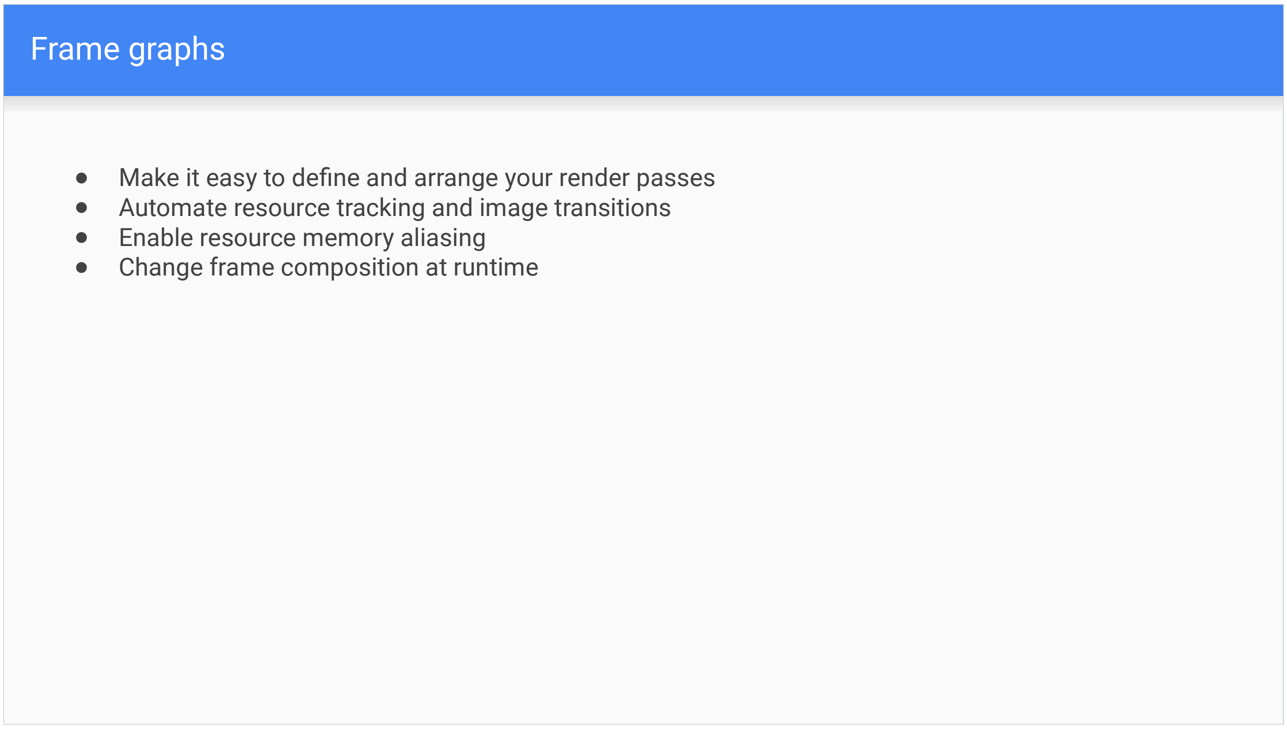
Frame Graph

Mesh Shaders

Ray Tracing

As before, we are going to break down each part into two 40’ sessions, so that we have time for questions in between. As a quick recap, in Part 1 we gave a brief overview of all the main components of the Vulkan API and to use them to submit work to the GPU.
In this second part we are going to cover some features that might or might not be aware of:

- Frame graphs are not part of the Vulkan API. It’s a framework that can remove some of the manual resource tracking that you have to do and that we have shown in the first part
- Mesh shaders are now officially part of the Vulkan API and provide an alternative to the traditional geometry pipeline. They allow, amongst other things, to perform culling on the GPU
- Ray tracing is also part of the official Vulkan API. The API has a lot of moving parts and we are going to provide an introduction to it

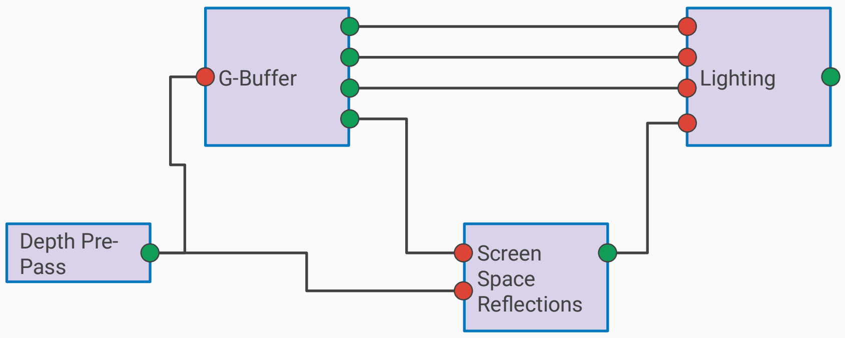


## Frame graphs

- Make it easy to define and arrange your render passes
- Automate resource tracking and image transitions
- Enable resource memory aliasing
- Change frame composition at runtime

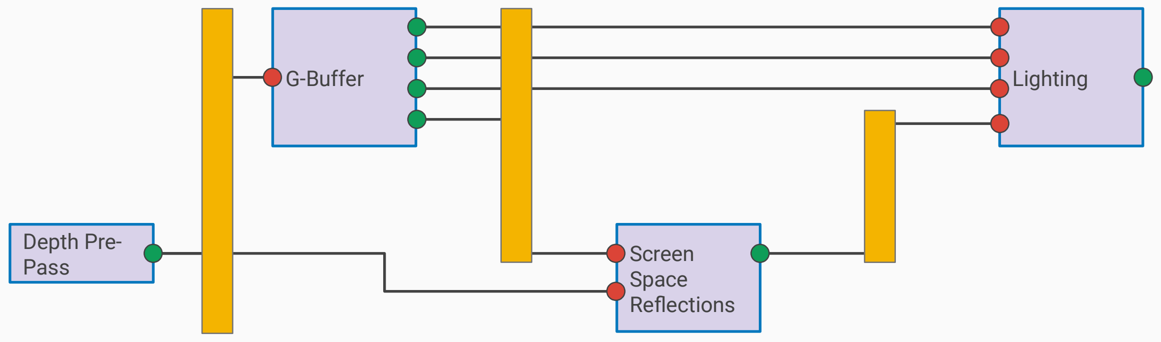
Frame graphs have been implemented in most rendering engines. They simplify adding and organizing the render passes in your frame and, more importantly, they can perform resource tracking and transitions for you.
Because a frame graph has knowledge of the whole frame, it can also alias memory between resources to reduce the overall memory used by the application.
Finally, it’s possible to change the active passes at run-time to make it easier to compare techniques.

## Frame graphs



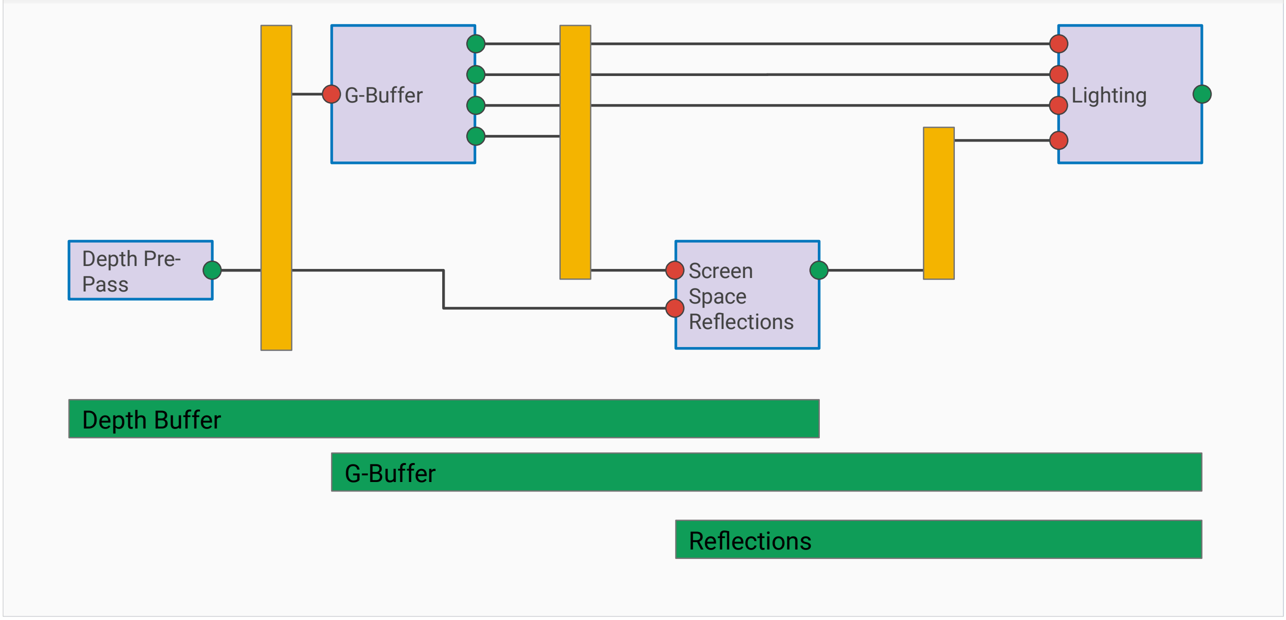
As the name implies, a frame graph organizes the render passes into a graph. This is done by defining the inputs and outputs for each node. Here we see a simple example of a frame graph with four passes.
The great thing about a frame graph is that you don’t have to worry about defining the passes (nodes) in order. Because we have inputs and outputs, we can build a directed acyclic graph (DAG) from the nodes and then perform a topological sort. You will have to make sure your graph doesn’t contain any cycles, otherwise it will be invalid.

## Frame graphs



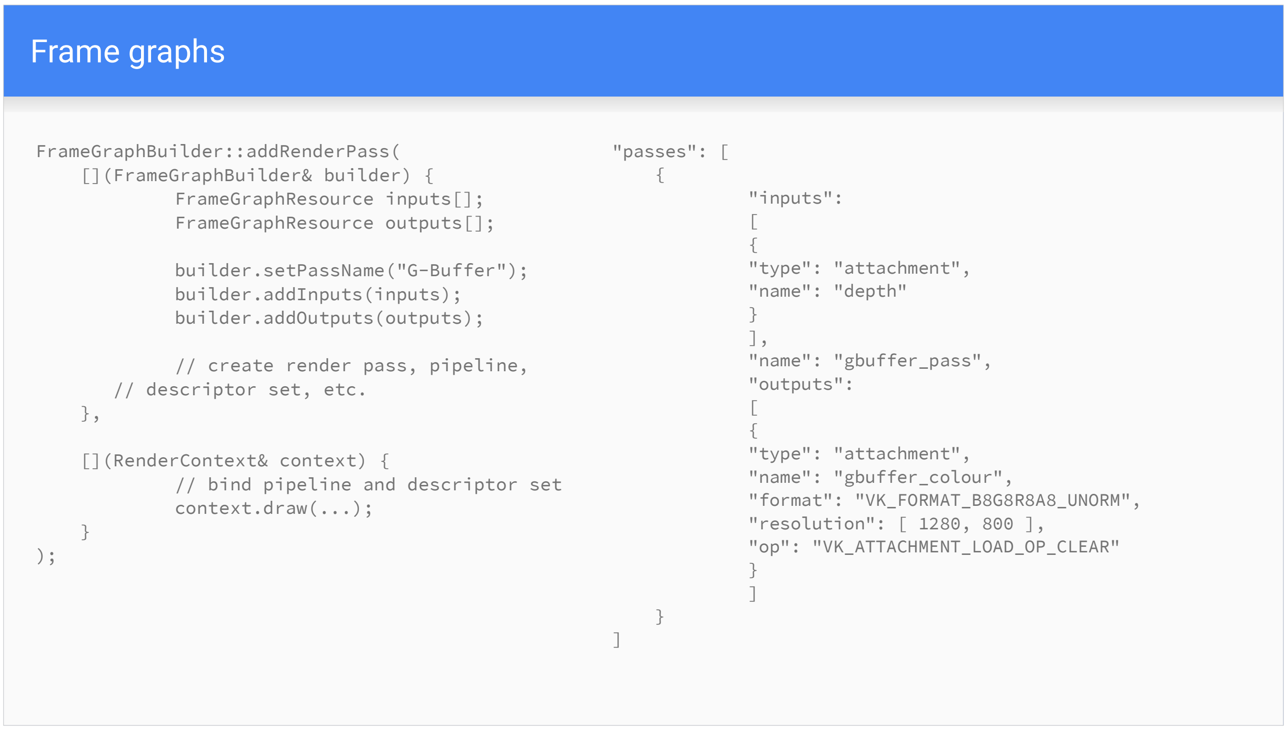
Each pass produces one or more outputs. Because we know how each resource is going to be used next, we can automatically insert the image transitions where needed. We also track which resources have already been transitioned so that we can avoid redundant barriers.
It’s not the case in our example, but we can also determine if some passes can run in parallel and organize our command buffers accordingly.

## Frame graphs



The other important feature that frame graphs provide is memory aliasing. Memory aliasing happens when two resources can share the same underlying memory.
In this example the memory for the depth buffer can be reused after we are done with the screen space reflections pass.

It’s important to note that we might not be able to alias some resources: for instance resources that need to persist across frames. These resources are usually marked (i.e. as external resources) to tell the frame graph that the user is responsible to manage them.



## Frame graphs

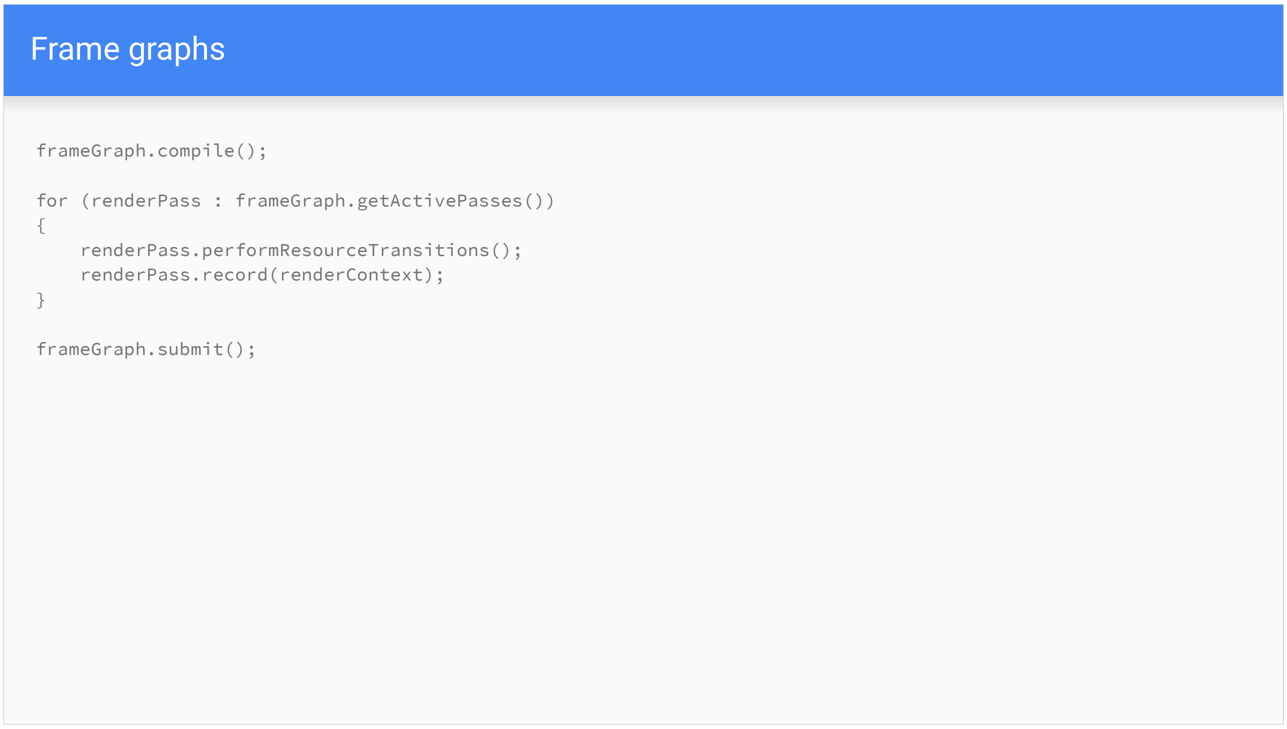
```
FrameGraphBuilder::addRenderPass(
    [](FrameGraphBuilder& builder) {
            FrameGraphResource inputs[];
            FrameGraphResource outputs[];

            builder.setPassName("G-Buffer");
            builder.addInputs(inputs);
            builder.addOutputs(outputs);

            // create render pass, pipeline,
      // descriptor set, etc.
    },

    [](RenderContext& context) {
            // bind pipeline and descriptor set
            context.draw(...);
    }
);
```

```
"passes": [
    {
            "inputs":
            [
            {
            "type": "attachment",
            "name": "depth"
            }
            ],
            "name": "gbuffer_pass",
            "outputs":
            [
            {
            "type": "attachment",
            "name": "gbuffer_colour",
            "format": "VK_FORMAT_B8G8R8A8_UNORM",
            "resolution": [ 1280, 800 ],
            "op": "VK_ATTACHMENT_LOAD_OP_CLEAR"
            }
            ]
    }
]
```

Now that we know what frame graphs are and how they work, we can look at how to implement them. There are two ways that this can be done:
- In code: defining your graph in code means your logic and graph definition are in one place. The downside is that you need a coder to make changes to the graph and you need to re-compile your code after the changes
- Data driven: in a data driven implementation, the graph is defined in a human readable format (or through a visual tool). This allows non-coders to modify the graph, however you still to implement new passes in code. Defining your graph with data could also allow you to pre-process your data to compile only the shader that are in use by the graph, etc.

Both approaches produce the same results in the end, choose the one that works best for you.

Here on the left we have an example of a pseudo-code implementation, and data driven one on the right.



## Frame graphs

```
frameGraph.compile();

for (renderPass : frameGraph.getActivePasses())
{
    renderPass.performResourceTransitions();
    renderPass.record(renderContext);
}

frameGraph.submit();
```

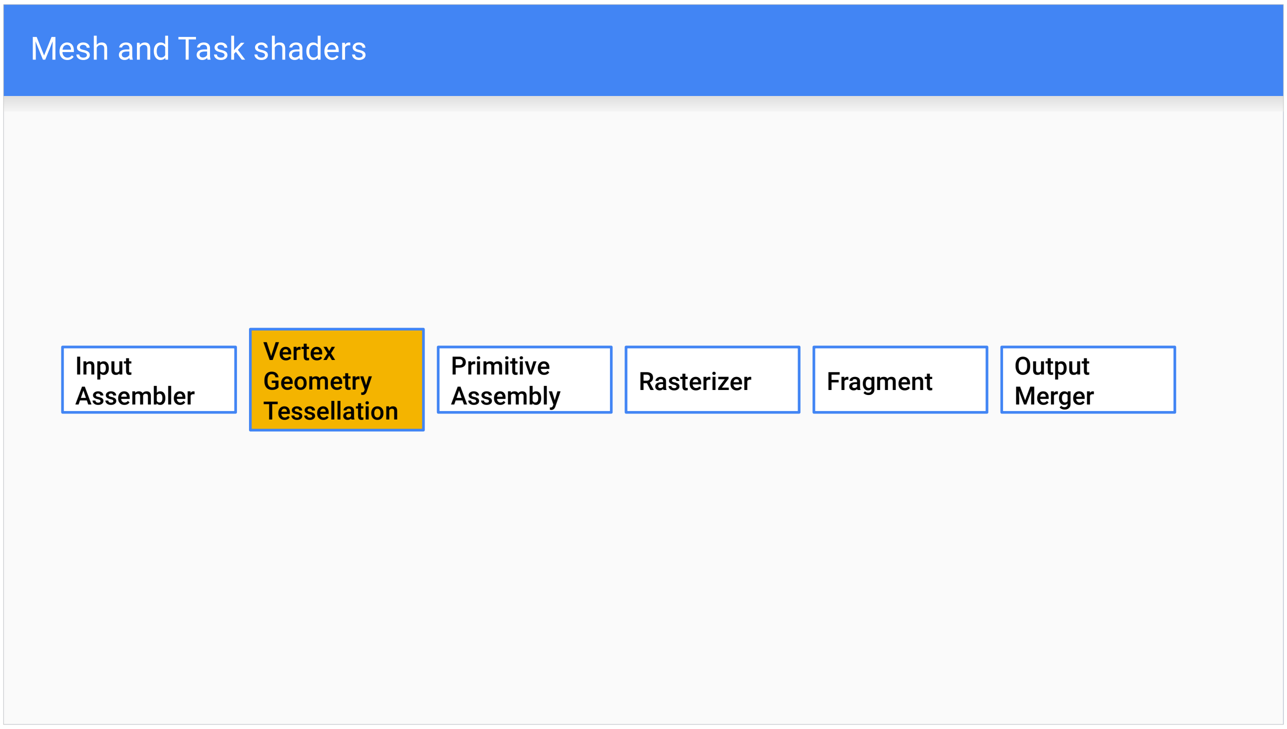
In pseudo code, this is what your render loop might look like now that we have implemented a frame graph.

- The compile step performs the topological sort and memory analysis to determine which render passes are active and their order
- Then we loop through all the active render passes. For each render pass we transition each resource to the correct layout and then we record the commands for the active render pass
- The last step is to submit the recorded command to the device

In this example the command buffer recording is done in a serial fashion. If you have a large number of passes, this step could be multi-threaded to improve the recording time.

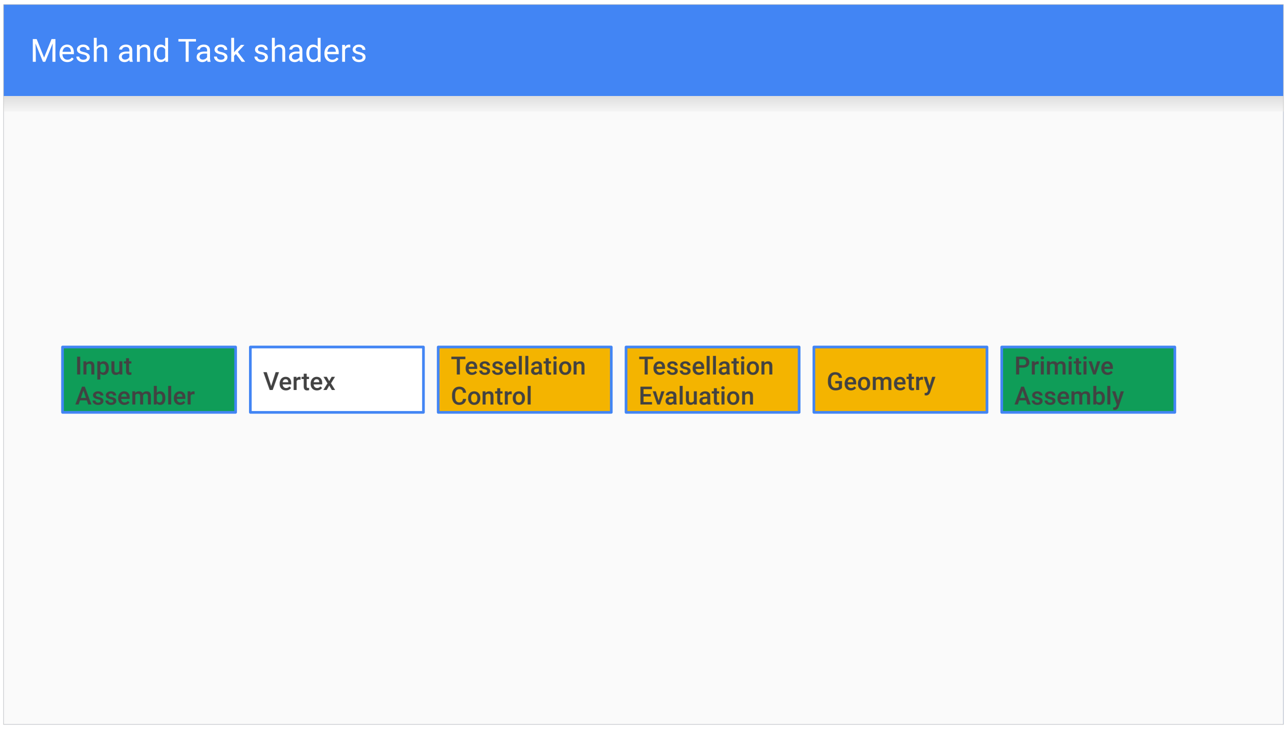
Nodes should also have knowledge about which queue they are using, so that the the framegraph is aware of any extra synchronization that needs to happen to ensure correct execution.

We have glossed over some implementation details, but the concept really is this simple. We are going to cover mesh shades next.



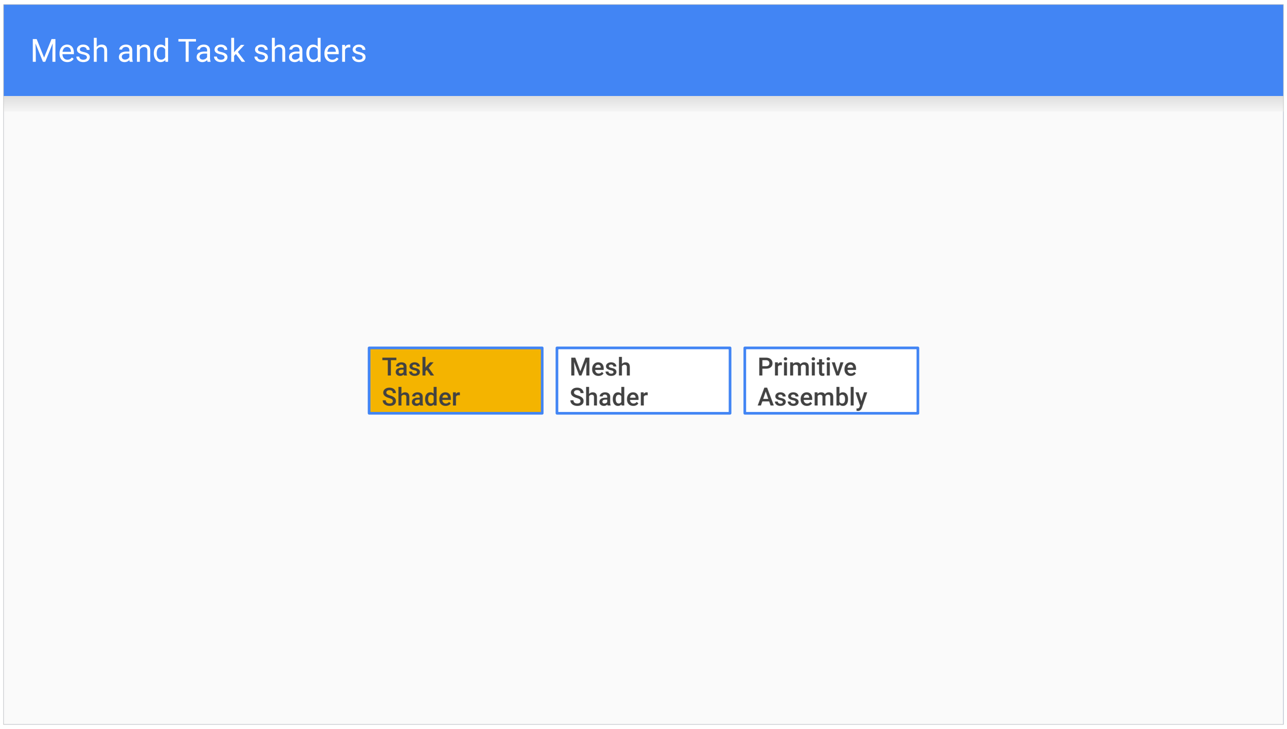
## Mesh and Task shaders

In the first part of the talk, we gave an overview of the GPU graphics pipeline. We are now going to focus on the geometry part of the pipeline



## Mesh and Task shaders

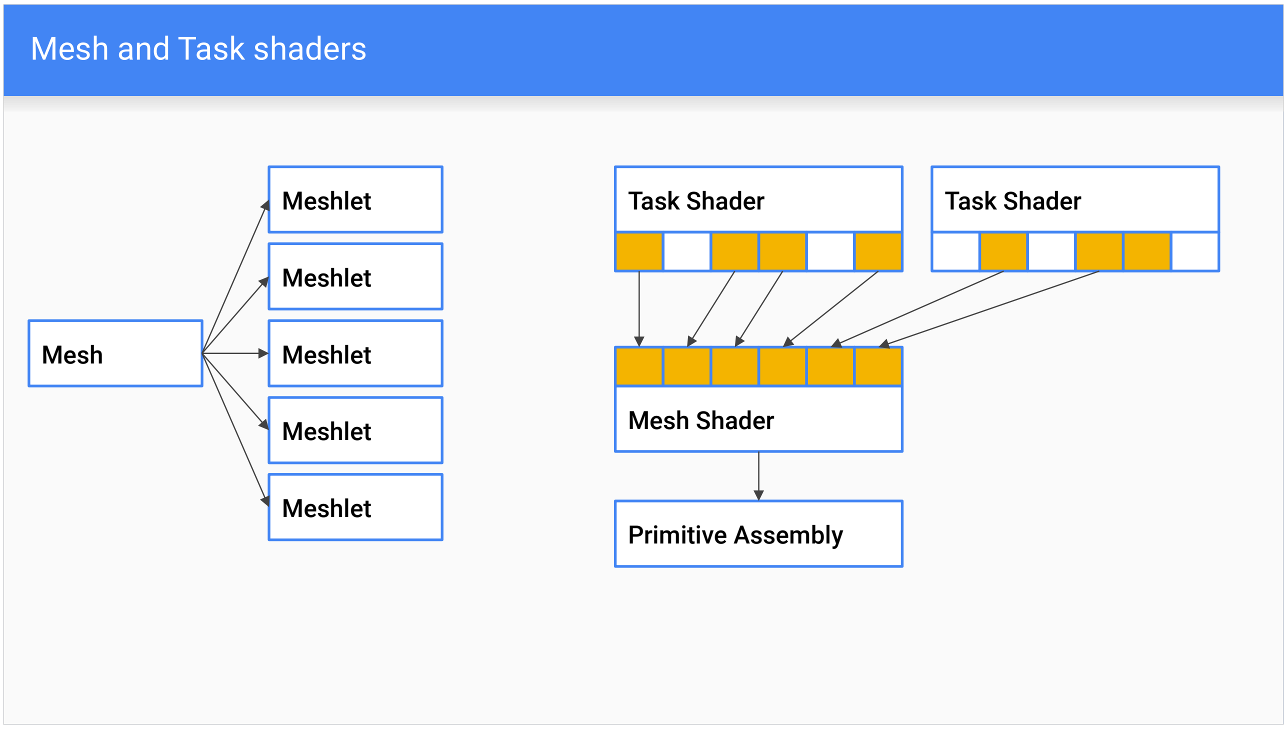
In a traditional graphics pipeline, we have multiple stages that can transform a vertex or produce new primitives. The stages in yellow are optional, while the stages in green are fixed.
Tessellation and Geometry shaders can be used to create new primitives (i.e. point to triangle) but they have very limited capability. They can’t, for instance, discard a vertex and, depending on your use case, they could have poor performance.
An option that has become common is to perform some of this processing in compute shaders. However, as we have seen in part 1, this requires careful handling of resource synchronization to make sure they are accessed correctly.



## Mesh and Task shaders

Task Shader → Mesh Shader → Primitive Assembly

With the introduction of Mesh shaders the geometry pipeline becomes a lot simpler, and mesh shaders open up more possibilities to what we can do with the incoming geometry.
Task shaders are not mandatory, but, as we'll see in our examples, they allow the implementation of a lot of features that are not possible using mesh shaders alone.

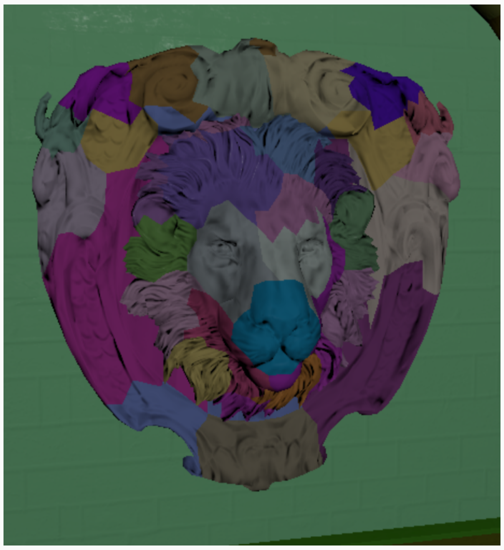


Before we dive into the code, we give a brief overview of how task and mesh shaders work.

- We start by pre-processing our mesh and divide it into what we call meshlets. These are small groups of triangles (usually 64 to 128).
- Each task shader invocation processes a number of meshlets (usually 32 or 64) and we determine which meshlets we are going to process further (the yellow ones on the slide). You could, for instance, cull meshlets outside the frustum or meshlets that are back-facing the camera. You can also perform occlusion culling at this stage.
- These meshlets are then coalesced for efficiency and are processed by the mesh shader.
- The mesh shader then processes all the triangles for each meshlets (as you would in a vertex shader) to determine their final position in NDC. However, it's also possible to refine the meshlet further by doing, for instance, triangle culling.
- The mesh shader is also responsible for writing the indices of the active triangles.

## Mesh and Task shaders

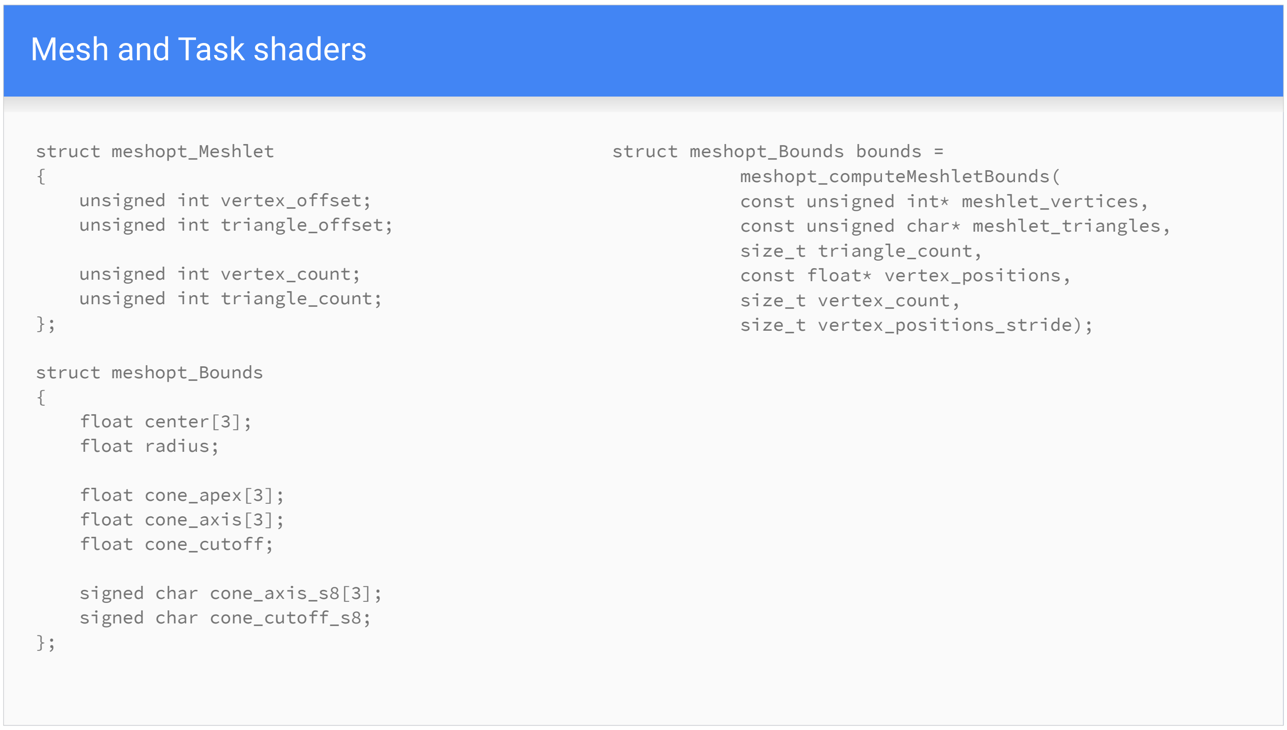
```
max_meshlets = meshopt_buildMeshletsBound(
    mesh_index_count,
    max_vertices_per_meshlet,
    max_triangles_per_meshlet);

meshlet_count = meshopt_buildMeshlets(
      struct meshopt_Meshlet* meshlets,
      unsigned int*,
      unsigned char* meshlet_triangles,
      const unsigned int* indices,
      size_t index_count,
      const float* vertex_positions,
      size_t vertex_count,
      size_t vertex_positions_stride,
      size_t max_vertices_per_meshlet,
      size_t max_triangles_per_meshlet,
      float cone_weight)
```



Now that we have provided an overview of how mesh and task shaders work, it’s time to look at how to implement them.
As we mentioned, the first step is to process your mesh to obtain meshlets. We use a library called mehsoptimizer in this example, but there are other available (or you could write your own!)

- The first API call computes the maximum number of meshlets we could have given the number of indices in the mesh, the maximum number of vertices per meshlet and the maximum number of triangles per meshlet. Depending on your mesh and how much vertex reuse it’s possible, the algorithm might not always be able to reach the specified limit
- The next API call is the one that builds the meshlets. It returns an array of meshopt_Meshlet (which we’ll look at in a moment), a vertex and index array for all meshlets (meshlet_vertices and meshlet_triangles). We provide the indices and vertices for our mesh, the same maximum values we specified in the previous API call and a cone weight. We’ll talk about the cone weight in the next slide

Here on the right, you can see the meshlets that compose the lion mesh in the Sponza model.



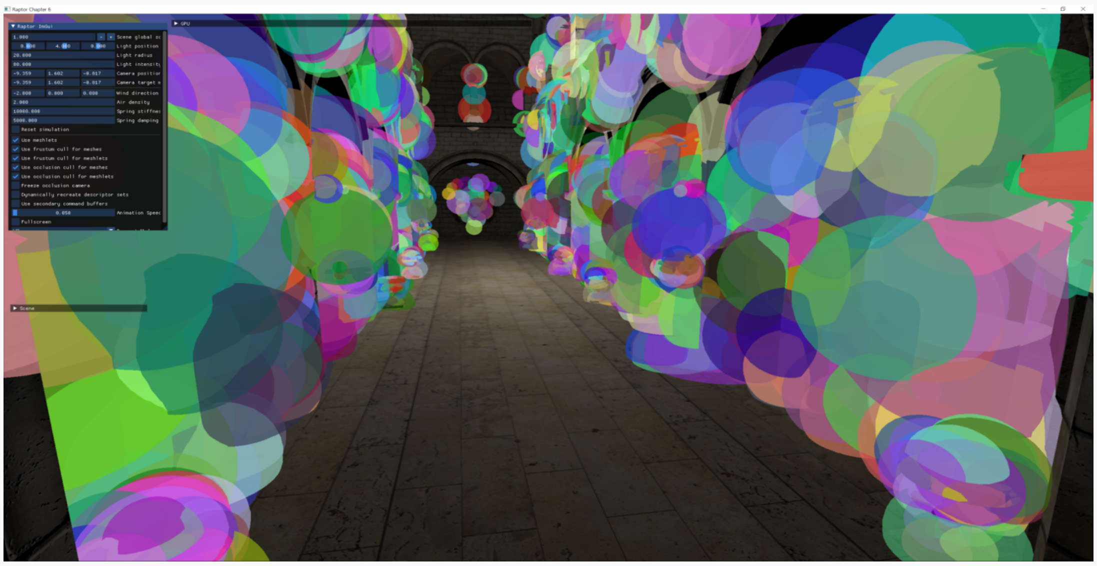
## Mesh and Task shaders

```
struct meshopt_Meshlet
{
    unsigned int vertex_offset;
    unsigned int triangle_offset;

    unsigned int vertex_count;
    unsigned int triangle_count;
};

struct meshopt_Bounds
{
    float center[3];
    float radius;

    float cone_apex[3];
    float cone_axis[3];
    float cone_cutoff;

    signed char cone_axis_s8[3];
    signed char cone_cutoff_s8;
};
```

```
struct meshopt_Bounds bounds =
            meshopt_computeMeshletBounds(
            const unsigned int* meshlet_vertices,
            const unsigned char* meshlet_triangles,
            size_t triangle_count,
            const float* vertex_positions,
            size_t vertex_count,
            size_t vertex_positions_stride);
```

As we mentioned in the previous slide, the API call to build the meshlets returns an array of vertices and triangles for all the meshlets. The meshopt_Meshlet structure tells us the vertex and index offset and count for each meshlet.
We could stop here and use the data we have so far to render our meshlets without using a task shader.
One of the main reasons to use a task shader however is to perform culling on the GPU. To perform this operation we need some more details.
For each meshlet, we can compute its bounds by calling meshopt_computeMeshletBounds. We already have all the data we need to call this function, which will return the bounds for this meshlet.
The bounds are computed as a sphere with a center and radius. We then have a cone that encodes the direction of the meshlet. The cone accuracy is determined by the cone_weight we have seen in the previous slide. The cone will be used to perform backface culling.

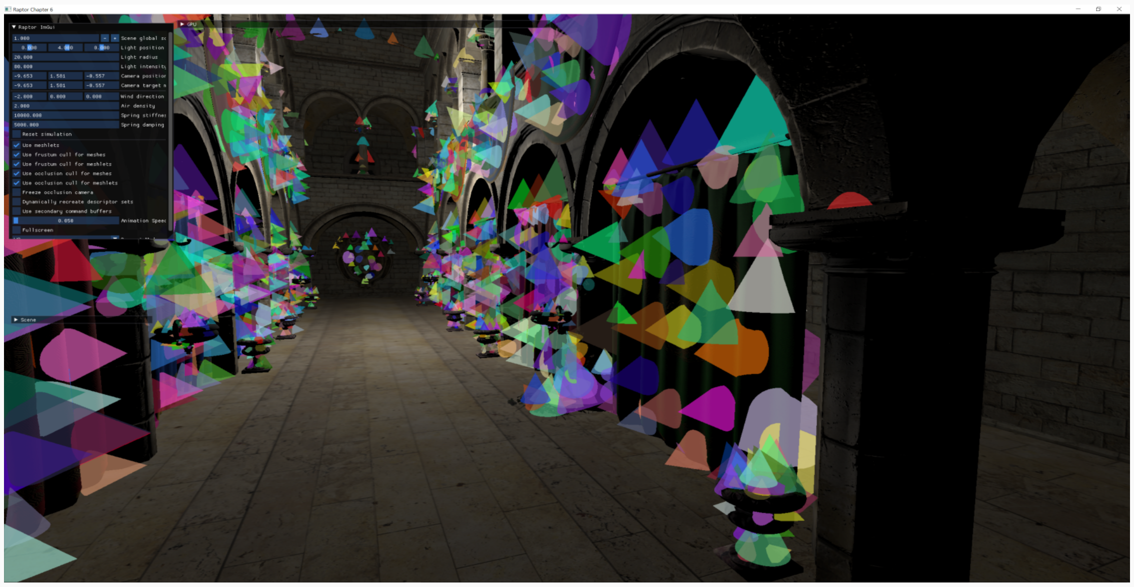
## Mesh and Task shaders



```
1.000                            - +  Scene global sc
0.000       4.000       0.000         Light position
20.000                                Light radius
80.000                                Light intensity
-9.359      1.602       -0.017        Camera position
-9.359      1.602       -0.017        Camera target m
-2.000      0.000       0.000         Wind direction
2.000                                 Air density
10000.000                             Spring stiffnes
5000.000                              Spring damping
[ ] Reset simulation
[✓] Use meshlets
[✓] Use frustum cull for meshes
[✓] Use frustum cull for meshlets
[✓] Use occlusion cull for meshes
[✓] Use occlusion cull for meshlets
[ ] Freeze occlusion camera
[ ] Dynamically recreate descriptor sets
[ ] Use secondary command buffers
0.050                                 Animation Speed
[ ] Fullscreen
```

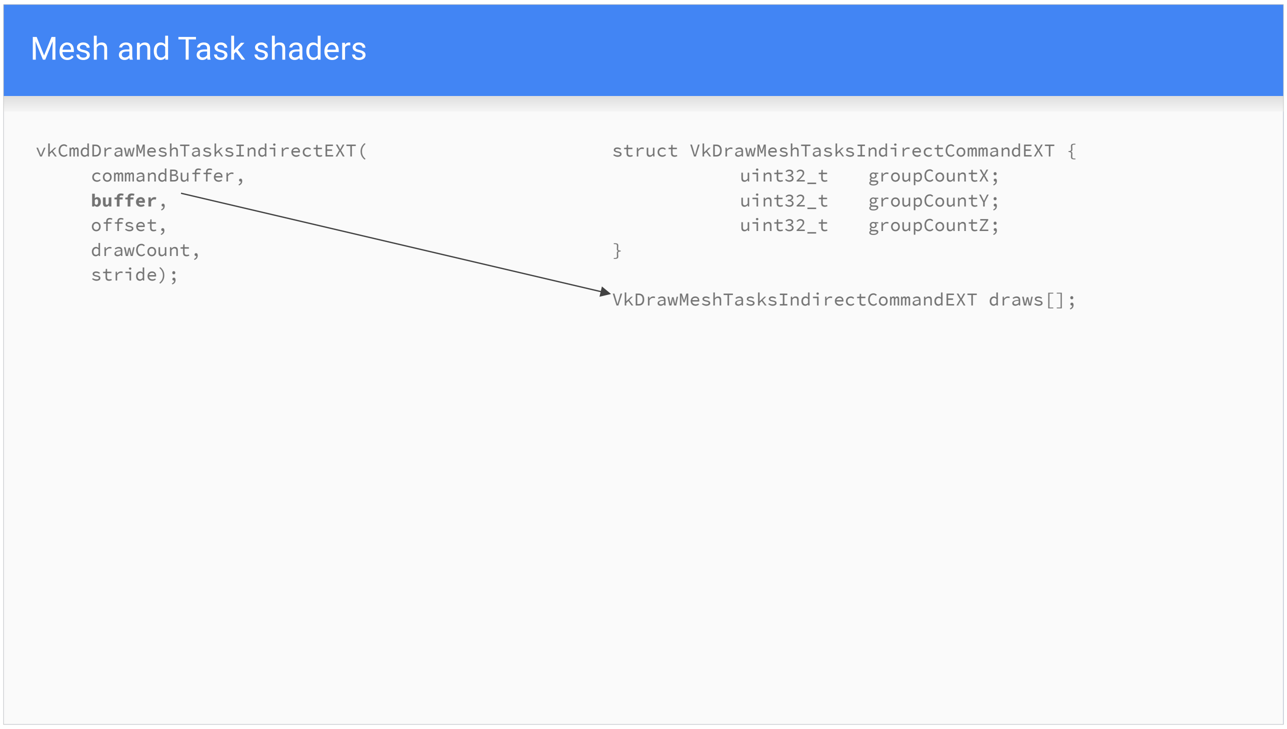
Before we look at the implementation, here is an example of the bounding spheres for each meshlet. As you can see it's quite coarse, but it still allows us to cull many meshlets.

## Mesh and Task shaders



```
▼ Raptor ImGui
1.000                      - +  Scene global sc
0.000     4.000     0.000       Light position
20.000                          Light radius
80.000                          Light intensity
-9.653    1.581     -0.557      Camera position
-9.653    1.581     -0.557      Camera target n
-2.000    0.000     0.000       Wind direction
2.000                           Air density
10000.000                       Spring stiffnes
5000.000                        Spring damping
[ ] Reset simulation
[✓] Use meshlets
[✓] Use frustum cull for meshes
[✓] Use frustum cull for meshlets
[✓] Use occlusion cull for meshes
[✓] Use occlusion cull for meshlets
[ ] Freeze occlusion camera
[ ] Dynamically recreate descriptor sets
[ ] Use secondary command buffers
0.050                           Animation Speed
[ ] Fullscreen
```

Likewise here's a visualization of the meshlet cones. In both pictures we have omitted the bounding spheres and cones for the floor and the walls as otherwise it would have been hard to visualize the smaller meshlets.

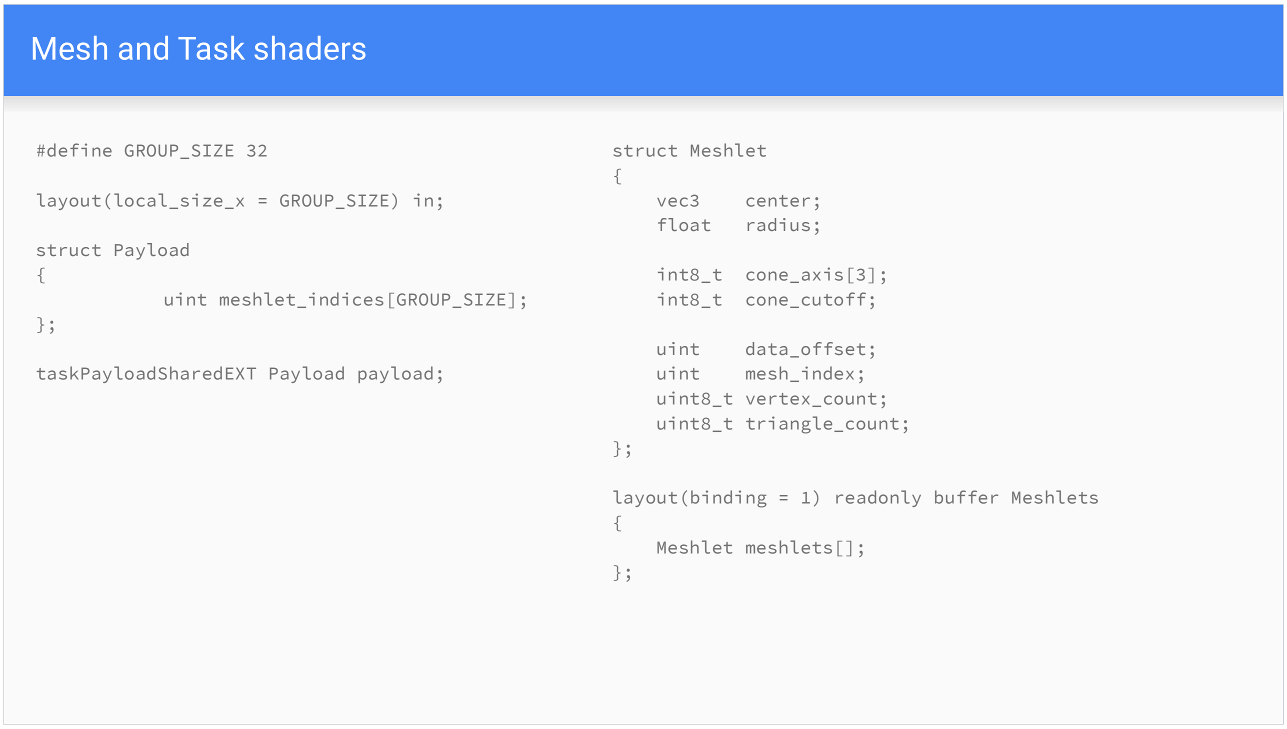


## Mesh and Task shaders

```
vkCmdDrawMeshTasksIndirectEXT(
     commandBuffer,
     buffer,
     offset,
     drawCount,
     stride);
```

```
struct VkDrawMeshTasksIndirectCommandEXT {
            uint32_t    groupCountX;
            uint32_t    groupCountY;
            uint32_t    groupCountZ;
}

VkDrawMeshTasksIndirectCommandEXT draws[];
```

We can now proceed with the implementation details. We will skip the pipeline creation because it follows the same steps we have shown in part 1 (shader module, pipeline layout, etc.)
Invoking a mesh shader is quite simple, the same way you would invoke a compute shader. Really, that's all there is to it. If you have created a task shader, it will be invoked first, otherwise the mesh shader will be invoked directly.
If you haven't used indirect draws before, they allow you to specify the list of draws into a buffer that will be read by the GPU. The buffer is an array of VkDrawMeshTasksIndirectCommandEXT.
This approach allows us to draw all of our meshlets with a single command. The buffer can be filled from the CPU side or through a compute shader.



## Mesh and Task shaders

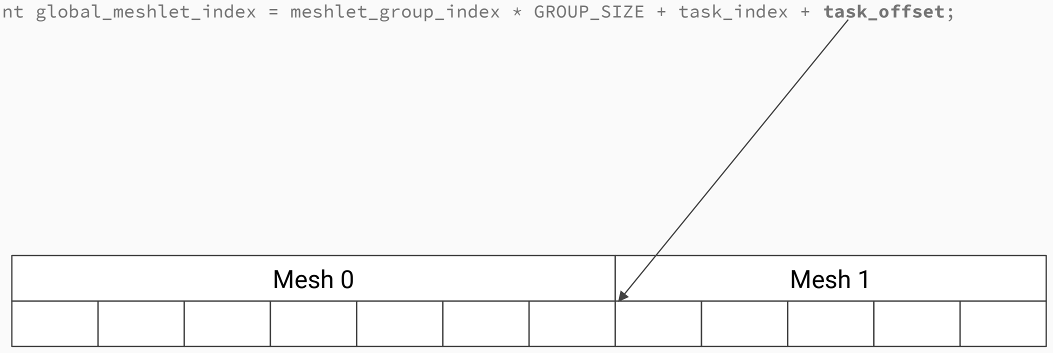
```
#define GROUP_SIZE 32

layout(local_size_x = GROUP_SIZE) in;

struct Payload
{
            uint meshlet_indices[GROUP_SIZE];
};

taskPayloadSharedEXT Payload payload;
```

```
struct Meshlet
{
    vec3    center;
    float   radius;

    int8_t  cone_axis[3];
    int8_t  cone_cutoff;

    uint    data_offset;
    uint    mesh_index;
    uint8_t vertex_count;
    uint8_t triangle_count;
};

layout(binding = 1) readonly buffer Meshlets
{
    Meshlet meshlets[];
};
```

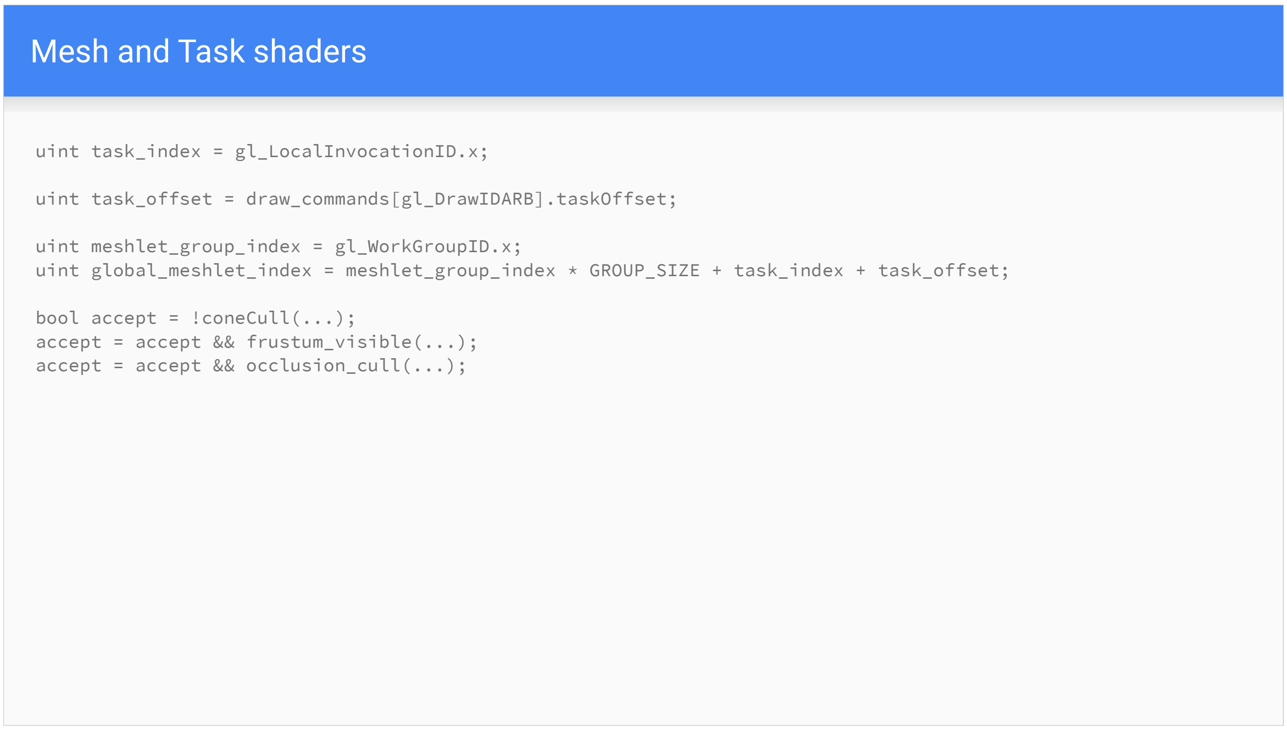
We now explain the task shader implementation. First we define the thread group size for our task shader. It works in the same way as a standard compute shader. Next, we have to define our shader payload. This payload is used to share data between the the task shader and mesh shader.
In our case, we only care about the meshlet indices that have not been culled, which we'll store in an array. The size of the array is the same as our thread group size.
We also have a buffer that contains all the meshlets we have created in the previous step. It contains all the fields needed to retrieve the meshlet data.

## Mesh and Task shaders

```
uint task_index = gl_LocalInvocationID.x;

uint task_offset = draw_commands[gl_DrawIDARB].taskOffset;

uint meshlet_group_index = gl_WorkGroupID.x;
uint global_meshlet_index = meshlet_group_index * GROUP_SIZE + task_index + task_offset;
```



The implementation starts with computing the index for meshlet being process by this thread.
The meshlets are stored in a contiguous array for all meshes. When we compute the meshlet index we need to take into account the offset for the meshlet we are processing.
Notice the use of the gl_DrawIDARB builtin variable. Recall from slide 19 that we are submitting all of our draws at once, and we can determine which draw we are processing using this variable.



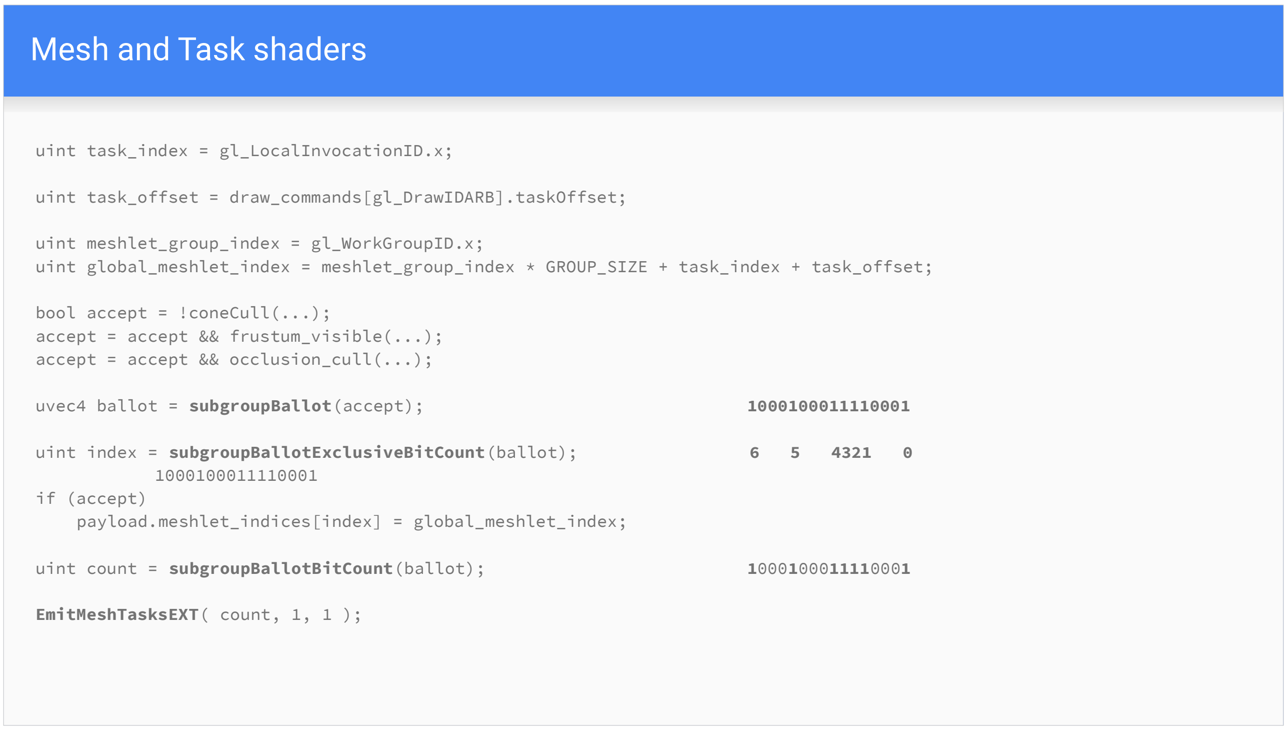
## Mesh and Task shaders

```
uint task_index = gl_LocalInvocationID.x;

uint task_offset = draw_commands[gl_DrawIDARB].taskOffset;

uint meshlet_group_index = gl_WorkGroupID.x;
uint global_meshlet_index = meshlet_group_index * GROUP_SIZE + task_index + task_offset;

bool accept = !coneCull(...);
accept = accept && frustum_visible(...);
accept = accept && occlusion_cull(...);
```

The next step is to determine whether we should process this meshlet further. We perform three tests:

- back face culling, using the cone we pre-computed for the meshlet
- frustum culling determines if the meshlet is outside the camera frustum
- finally we determine if the meshlet is not occluded by other geometry

We’ll go into details for each of these steps in a few slides.



## Mesh and Task shaders

```
uint task_index = gl_LocalInvocationID.x;

uint task_offset = draw_commands[gl_DrawIDARB].taskOffset;

uint meshlet_group_index = gl_WorkGroupID.x;
uint global_meshlet_index = meshlet_group_index * GROUP_SIZE + task_index + task_offset;

bool accept = !coneCull(...);
accept = accept && frustum_visible(...);
accept = accept && occlusion_cull(...);

uvec4 ballot = subgroupBallot(accept);                                1000100011110001

uint index = subgroupBallotExclusiveBitCount(ballot);                 6   5   4321   0
            1000100011110001
if (accept)
    payload.meshlet_indices[index] = global_meshlet_index;

uint count = subgroupBallotBitCount(ballot);                          1000100011110001

EmitMeshTasksEXT( count, 1, 1 );
```

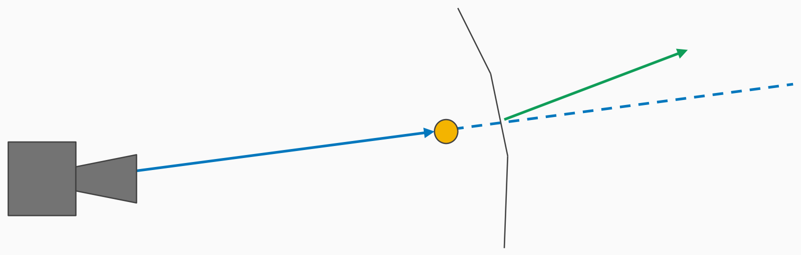
Once we determined the visibility for this meshlet, we need to count how many meshlets have passed the test across the thread group.

- subgroupBallot computes a bitfield across all threads. In our example we have 16 threads, a 1 is inserted for all the threads were accept is true
- subgroupBallotExclusiveBitCount computes the bit count up the active thread. Basically this computes the index only for the threads where accept was true, which we use to write out the index of the meshlet. This index will be used by the mesh shader for further processing
- subgroupBallotBitCount counts all the active bits. We use this value to tell the next stage how many elements to process through EmitMeshTasksEXT

This concludes the logic for the task shader.

## Mesh and Task shaders

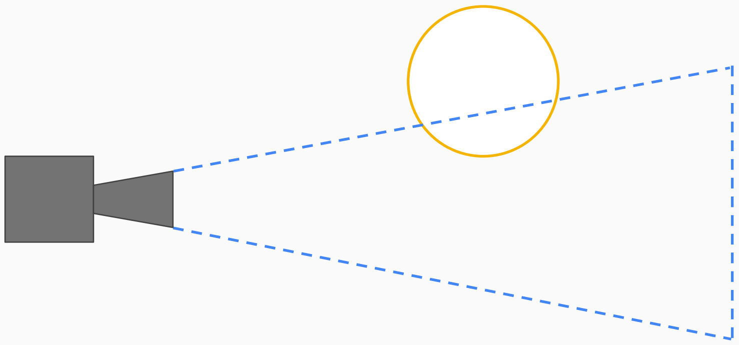
```
// as described in meshoptimizer.h
bool coneCull(vec3 center,
               float radius,
              vec3 cone_axis,
              float cone_cutoff,
              vec3 camera_position)
{
            return dot(center - camera_position, cone_axis) <
             cone_cutoff * length(center - camera_position) + radius;
}
```



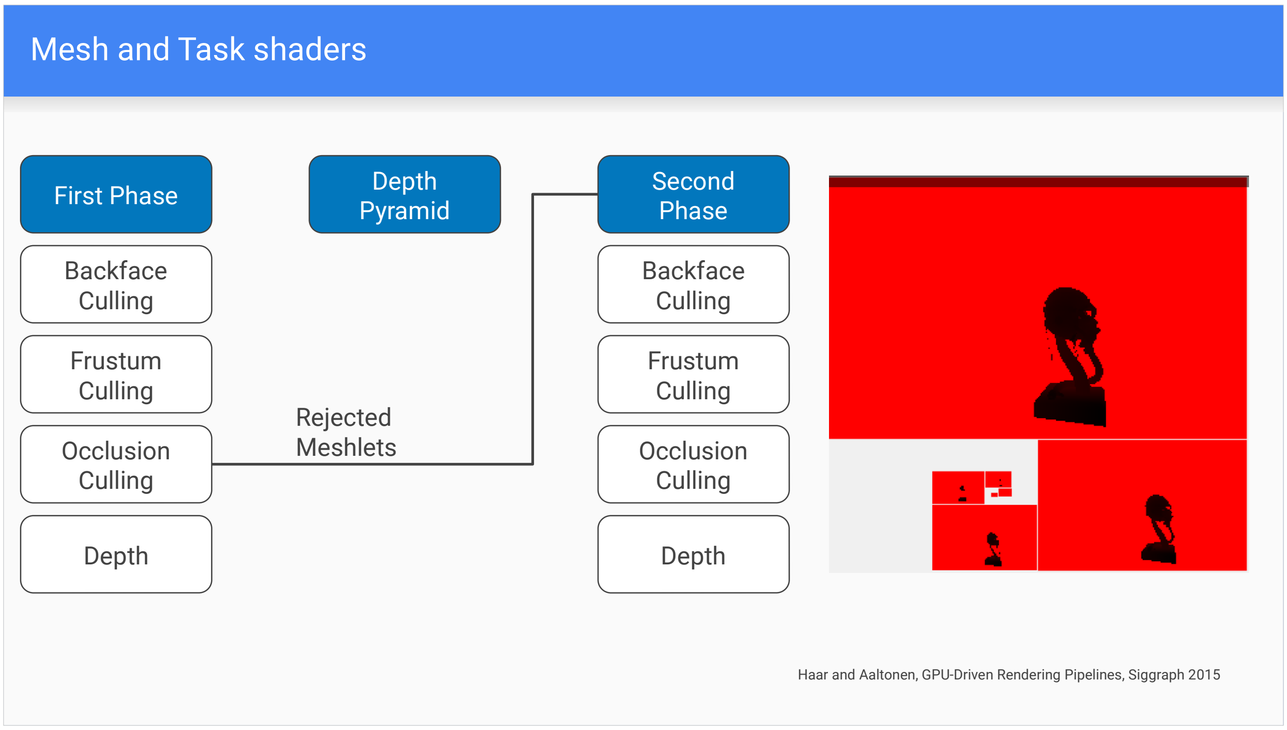
Before moving to the implementation of the mesh shader we are going to into more details of each culling step. We start with the back face culling.
We compute the angle between the camera and the center of the meshlet. If the angle is below a certain threshold, we consider the whole meshlet as backfacing and we cull it.

## Mesh and Task shaders

```
vec4 view_center = world_to_camera * world_center;

bool frustum_visible = true;
for ( uint i = 0; i < 6; ++i ) {
    frustum_visible = frustum_visible && (dot( frustum_planes[i], view_center) > -radius);
}
```



The next step if frustum culling. We bounding sphere for the meshlet against the 6 camera frustum planes. If it’s outside the frustum, we cull the meshlet.



## Mesh and Task shaders

Haar and Aaltonen, GPU-Driven Rendering Pipelines, Siggraph 2015

The final step is occlusion culling. Before looking at the code, we'll briefly explain how the algorithm works.
There are two phases:

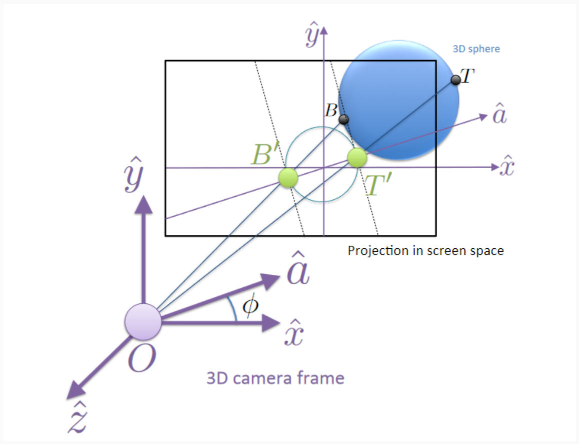
- first phase: we re-project each vertex using the transform matrix from the previous frame and we perform occlusion culling using a depth pyramid from the previous frame. The idea is that most of the geometry will be similar between frames
- we then update the depth pyramid with the geometry we just rendered
- second phase: we re-render the geometry we culled in the first phase to take into account any disocclusions that might have occurred because of object or camera movement

Here you can see what the depth pyramid looks like. It's implemented as a downsample using the max depth (or min depth if you're using reverse-z) as we want to be conservative.

You can find more details for this algorithm in this great Siggraph presentation.

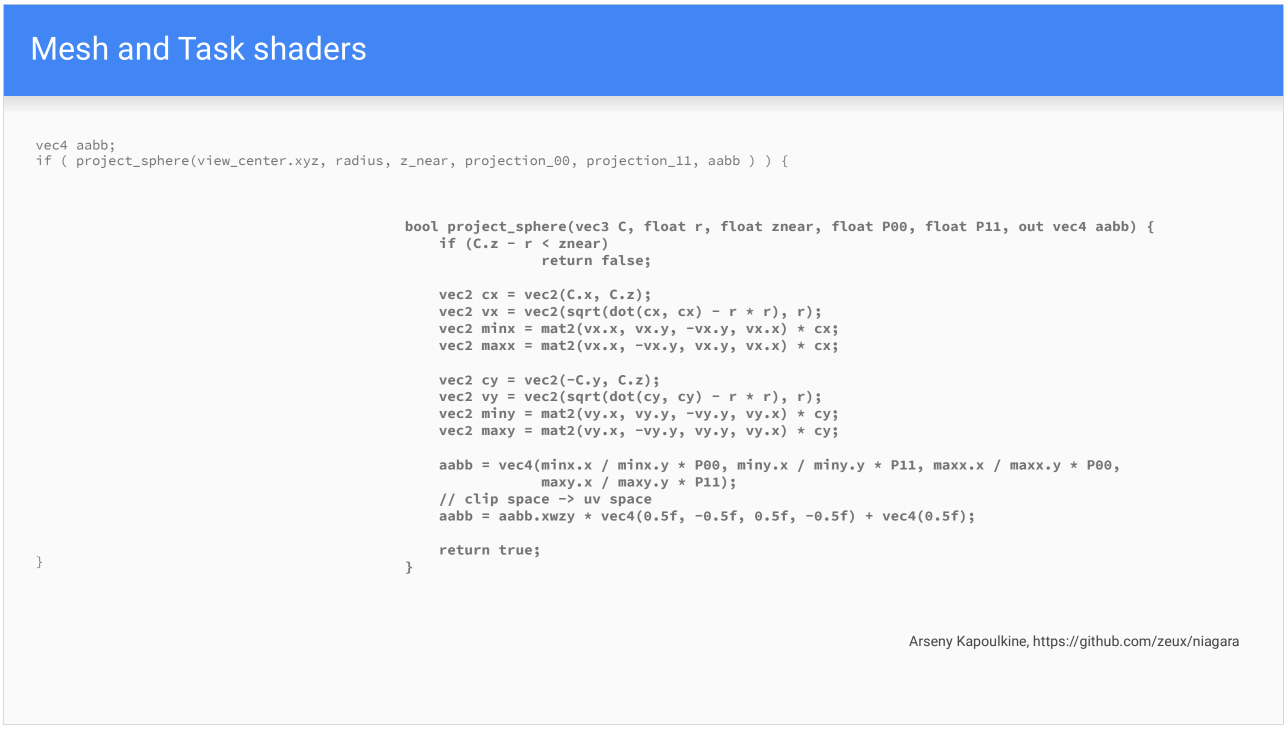
## Mesh and Task shaders

```
vec4 aabb;
if ( project_sphere(view_center.xyz, radius, z_near, projection_00, projection_11, aabb ) ) {

}
```



Michael Mara and Morgan McGuire, 2D Polyhedral Bounds of a Clipped, Perspective-Projected 3D Sphere, *Journal of Computer Graphics Techniques (JCGT)*, 2013

Recall that we computed a bounding sphere for each meshlet, which we are now going to use to determine whether a meshlet is occluded or not.
We first project the sphere in screen space, and we use the technique described in this paper. The projection of a 3D sphere to screen space has to take into account the camera position and projection.
The main idea of the paper is to solve the problem for each axis (we are only interested in x and y) and find the projection of the sphere on each axis.
Once we have these points (B’ and T’ in the picture) for x and y, we can determine the bounding box for the sphere in screen space.
The algorithm generalizes to more than two axis, but that’s enough for our purposes.



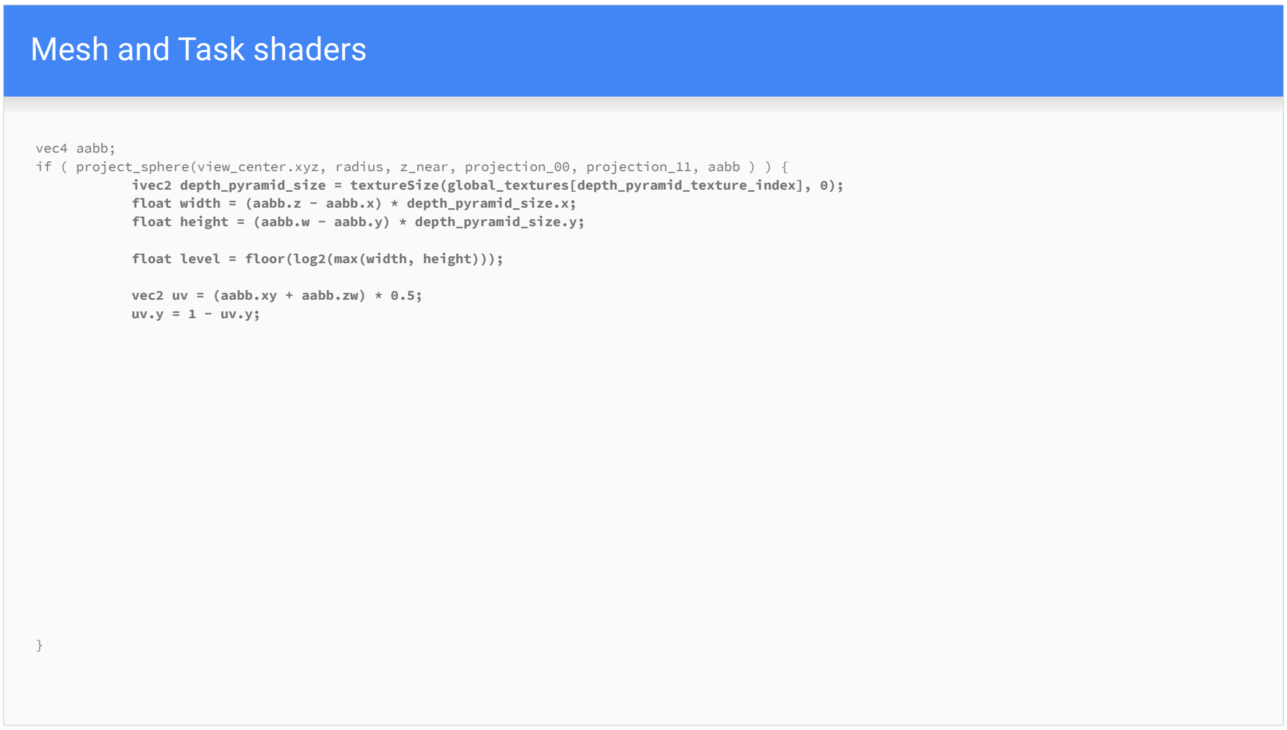
## Mesh and Task shaders

```
vec4 aabb;
if ( project_sphere(view_center.xyz, radius, z_near, projection_00, projection_11, aabb ) ) {

}
```

```
bool project_sphere(vec3 C, float r, float znear, float P00, float P11, out vec4 aabb) {
    if (C.z - r < znear)
                return false;

    vec2 cx = vec2(C.x, C.z);
    vec2 vx = vec2(sqrt(dot(cx, cx) - r * r), r);
    vec2 minx = mat2(vx.x, vx.y, -vx.y, vx.x) * cx;
    vec2 maxx = mat2(vx.x, -vx.y, vx.y, vx.x) * cx;

    vec2 cy = vec2(-C.y, C.z);
    vec2 vy = vec2(sqrt(dot(cy, cy) - r * r), r);
    vec2 miny = mat2(vy.x, vy.y, -vy.y, vy.x) * cy;
    vec2 maxy = mat2(vy.x, -vy.y, vy.y, vy.x) * cy;

    aabb = vec4(minx.x / minx.y * P00, miny.x / miny.y * P11, maxx.x / maxx.y * P00,
                maxy.x / maxy.y * P11);
    // clip space -> uv space
    aabb = aabb.xwzy * vec4(0.5f, -0.5f, 0.5f, -0.5f) + vec4(0.5f);

    return true;
}
```

Arseny Kapoulkine, https://github.com/zeux/niagara

The paper has code samples in C++ and glsl, but we opted to use this optimal implementation from the Niagara project.
We are sharing this for reference, we won't go into details of the actual implementation.
The author has also a series of videos detailing the implementation of this code (linked in the Github repo). We highly recommend watching the whole series as it contains many details for using mesh shaders and implementing occlusion culling.



## Mesh and Task shaders

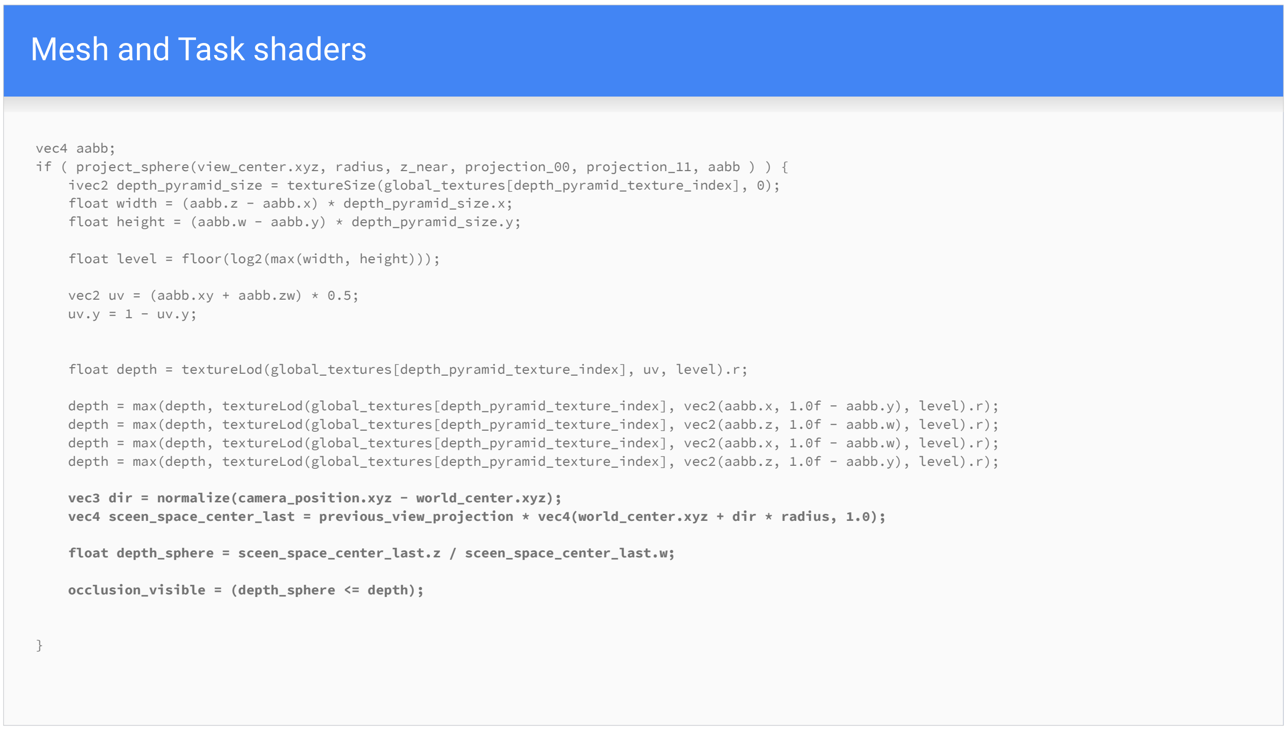
```
vec4 aabb;
if ( project_sphere(view_center.xyz, radius, z_near, projection_00, projection_11, aabb ) ) {
        ivec2 depth_pyramid_size = textureSize(global_textures[depth_pyramid_texture_index], 0);
        float width = (aabb.z - aabb.x) * depth_pyramid_size.x;
        float height = (aabb.w - aabb.y) * depth_pyramid_size.y;

        float level = floor(log2(max(width, height)));

        vec2 uv = (aabb.xy + aabb.zw) * 0.5;
        uv.y = 1 - uv.y;









}
```

If the sphere is visible on screen, we use its bounding box dimensions in screen space to compute which level of the pyramid we have to access.
This basically reduces the size of the bounding box to a single fragment, which we can do given that our depth pyramid is conservative.
We also compute the UV coordinates for our pyramid texture.
One important thing to note is that we are using a point sampler, we don't want interpolation across fragments.

## Mesh and Task shaders

```
vec4 aabb;
if ( project_sphere(view_center.xyz, radius, z_near, projection_00, projection_11, aabb ) ) {
    ivec2 depth_pyramid_size = textureSize(global_textures[depth_pyramid_texture_index], 0);
    float width = (aabb.z - aabb.x) * depth_pyramid_size.x;
    float height = (aabb.w - aabb.y) * depth_pyramid_size.y;

    float level = floor(log2(max(width, height)));

    vec2 uv = (aabb.xy + aabb.zw) * 0.5;
    uv.y = 1 - uv.y;


    float depth = textureLod(global_textures[depth_pyramid_texture_index], uv, level).r;

    depth = max(depth, textureLod(global_textures[depth_pyramid_texture_index], vec2(aabb.x, 1.0f - aabb.y), level).r);
    depth = max(depth, textureLod(global_textures[depth_pyramid_texture_index], vec2(aabb.z, 1.0f - aabb.w), level).r);
    depth = max(depth, textureLod(global_textures[depth_pyramid_texture_index], vec2(aabb.x, 1.0f - aabb.w), level).r);
    depth = max(depth, textureLod(global_textures[depth_pyramid_texture_index], vec2(aabb.z, 1.0f - aabb.y), level).r);









}
```

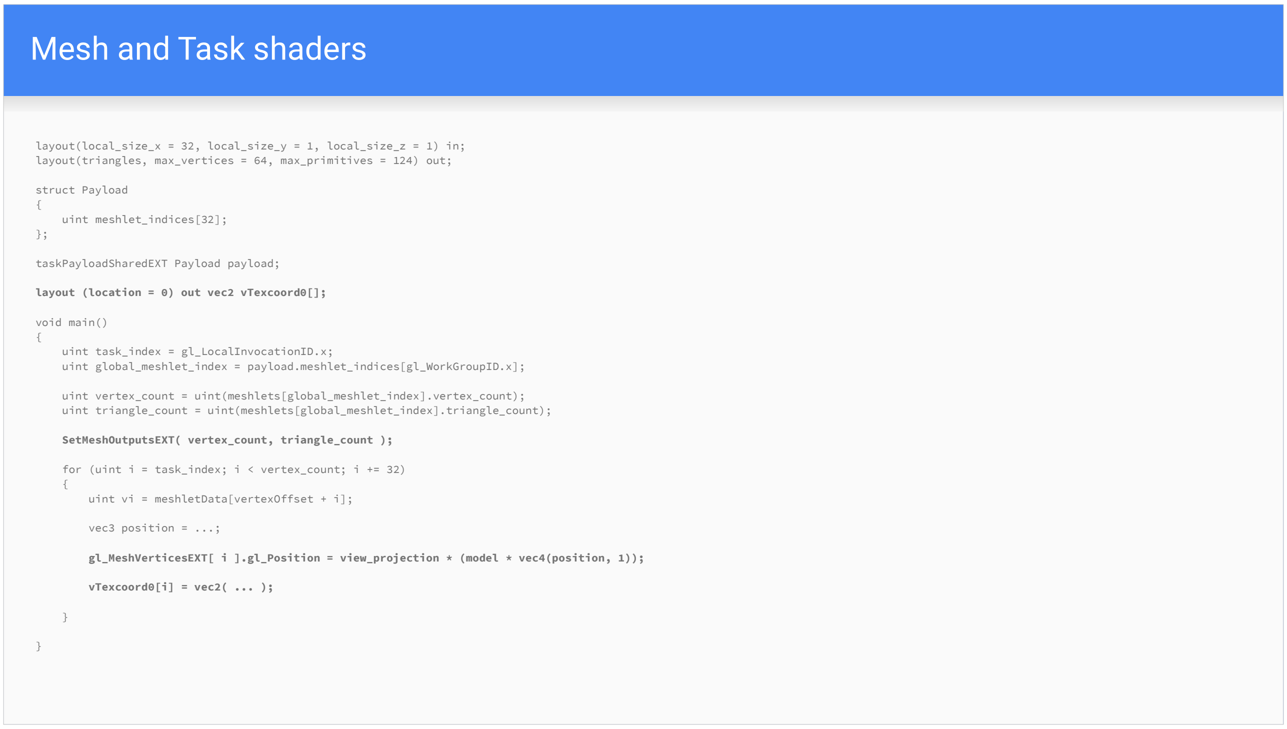
Next we read the depth at the computed UV coordinates and also the four corners of the bounding box. We keep the max depth, again to be conservative.
We sample the four corners to make sure we take into account the whole area for the projected sphere.



## Mesh and Task shaders

```
vec4 aabb;
if ( project_sphere(view_center.xyz, radius, z_near, projection_00, projection_11, aabb ) ) {
    ivec2 depth_pyramid_size = textureSize(global_textures[depth_pyramid_texture_index], 0);
    float width = (aabb.z - aabb.x) * depth_pyramid_size.x;
    float height = (aabb.w - aabb.y) * depth_pyramid_size.y;

    float level = floor(log2(max(width, height)));

    vec2 uv = (aabb.xy + aabb.zw) * 0.5;
    uv.y = 1 - uv.y;


    float depth = textureLod(global_textures[depth_pyramid_texture_index], uv, level).r;

    depth = max(depth, textureLod(global_textures[depth_pyramid_texture_index], vec2(aabb.x, 1.0f - aabb.y), level).r);
    depth = max(depth, textureLod(global_textures[depth_pyramid_texture_index], vec2(aabb.z, 1.0f - aabb.w), level).r);
    depth = max(depth, textureLod(global_textures[depth_pyramid_texture_index], vec2(aabb.x, 1.0f - aabb.w), level).r);
    depth = max(depth, textureLod(global_textures[depth_pyramid_texture_index], vec2(aabb.z, 1.0f - aabb.y), level).r);

    vec3 dir = normalize(camera_position.xyz - world_center.xyz);
    vec4 sceen_space_center_last = previous_view_projection * vec4(world_center.xyz + dir * radius, 1.0);

    float depth_sphere = sceen_space_center_last.z / sceen_space_center_last.w;

    occlusion_visible = (depth_sphere <= depth);


}
```

Finally we compute the minimum depth of the sphere in screen space and compare it with the depth we just retrieved.
If the depth is less that the depth stored in the pyramid, the meshlet is visible.
Otherwise we consider this meshlet as occluded.
This concludes our task shader implementation.



## Mesh and Task shaders

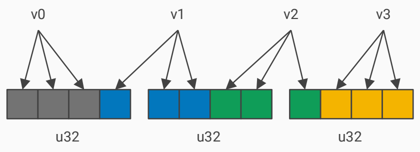
```
layout(local_size_x = 32, local_size_y = 1, local_size_z = 1) in;
layout(triangles, max_vertices = 64, max_primitives = 124) out;

struct Payload
{
    uint meshlet_indices[32];
};

taskPayloadSharedEXT Payload payload;

layout (location = 0) out vec2 vTexcoord0[];

void main()
{
    uint task_index = gl_LocalInvocationID.x;
    uint global_meshlet_index = payload.meshlet_indices[gl_WorkGroupID.x];

    uint vertex_count = uint(meshlets[global_meshlet_index].vertex_count);
    uint triangle_count = uint(meshlets[global_meshlet_index].triangle_count);

    SetMeshOutputsEXT( vertex_count, triangle_count );

    for (uint i = task_index; i < vertex_count; i += 32)
    {
        uint vi = meshletData[vertexOffset + i];

        vec3 position = ...;

        gl_MeshVerticesEXT[ i ].gl_Position = view_projection * (model * vec4(position, 1));

        vTexcoord0[i] = vec2( ... );

    }

}
```

We now take a quick look at the mesh shader implementation. This should look more familiar as it behaves almost like a vertex shader.
We have highlighted the main differences:

- We are not processing a single vertex, but all the vertices of the meshlet. For this reason, we have to output to an array of gl_MeshVerticesEXT
- For custom attributes, we also define an array, vTexcoord0 in this case. This should be familiar if you have used geometry or tessellation shaders before

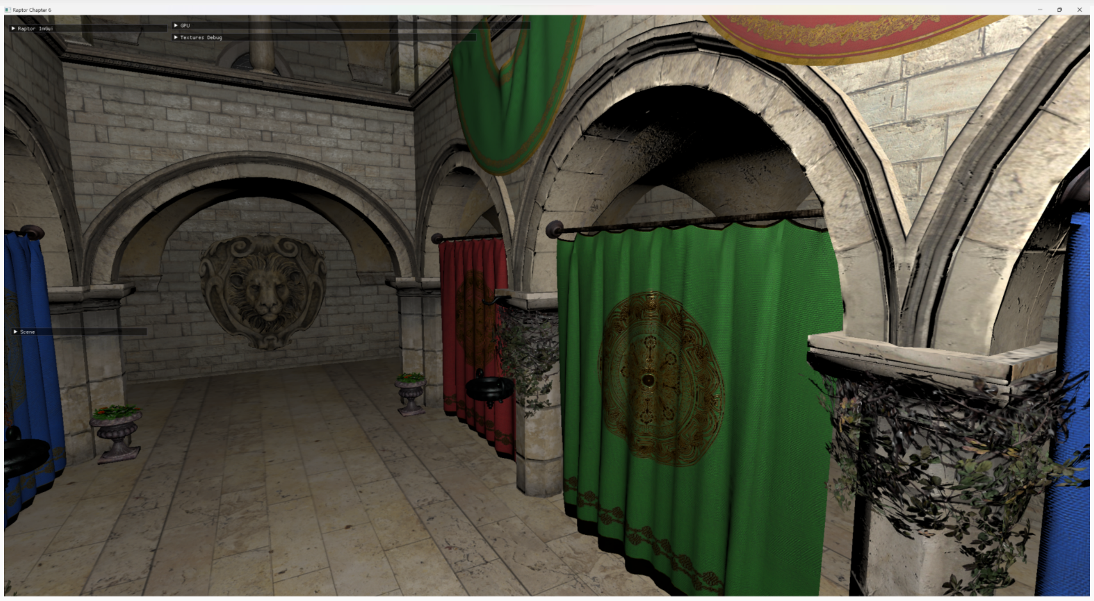
## Mesh and Task shaders

```
void main()
{
    ...
    uvec3 index_entry_offset[4] =
    {
        uvec3( 0, 0, 0 ),
        uvec3( 0, 1, 1 ),
        uvec3( 1, 1, 2 ),
        uvec3( 2, 2 ,2 ),
    };

    uvec3 index_entry_shift[4] =
    {
        uvec3(  0,  8, 16 ),
        uvec3( 24,  0,  8 ),
        uvec3( 16, 24,  0 ),
        uvec3(  8, 16, 24 ),
    };

    for (uint i = task_index; i < triangle_count; i += 32)
    {
        uint index_entry = i % 4;
        uint new_index = indexOffset + ( i / 4 ) * 3;
        gl_PrimitiveTriangleIndicesEXT[ i ] = uvec3(
            ( meshletData[ new_index + index_entry_offset[ index_entry ].x ] >> index_entry_shift[ index_entry ].x ) & 0xff,
            ( meshletData[ new_index + index_entry_offset[ index_entry ].y ] >> index_entry_shift[ index_entry ].y ) & 0xff,
            ( meshletData[ new_index + index_entry_offset[ index_entry ].z ] >> index_entry_shift[ index_entry ].z ) & 0xff
        );
    }
}
```



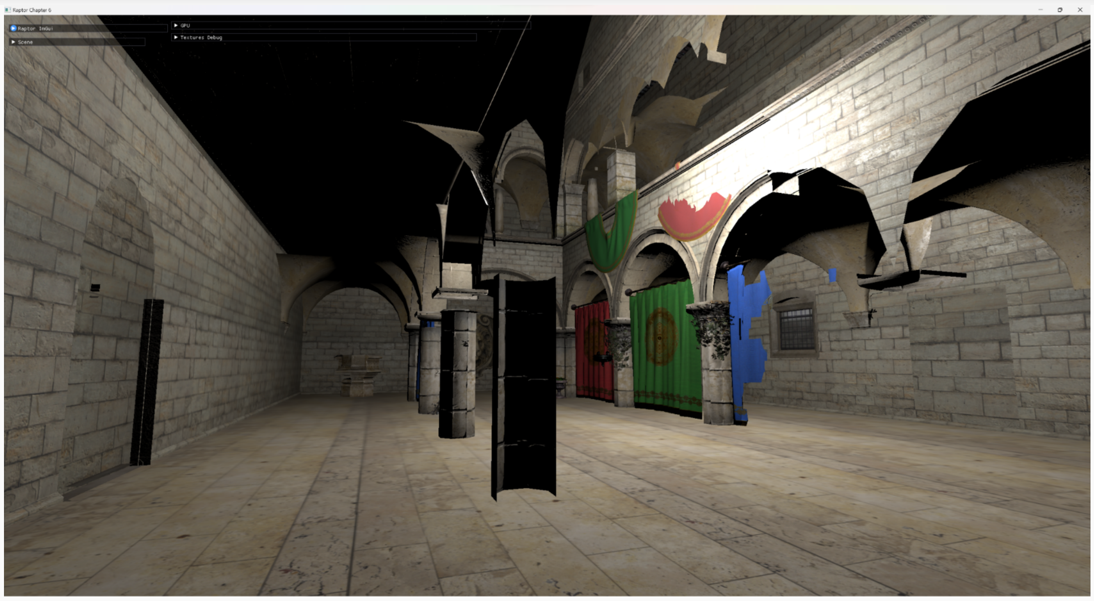
As we mentioned before, when using mesh shaders we are also responsible for writing out the indices for our vertices.
Our indices are stored together with the meshlet data, which is an array of unsigned integers. Each entry contains 4 indices, but we need three for each vertex.
We derived this algorithm to avoid branching. An alternative would be to store indices in a separate stream and interpret it as a uint8_t.
Yet another option is to use buffer device address, we'll explain more about this extension in the ray tracing section.

## Mesh and Task shaders



Now, at then end of all this you might ask: is it worth it? Here’s a screenshot from our framework (the same we used for the book).

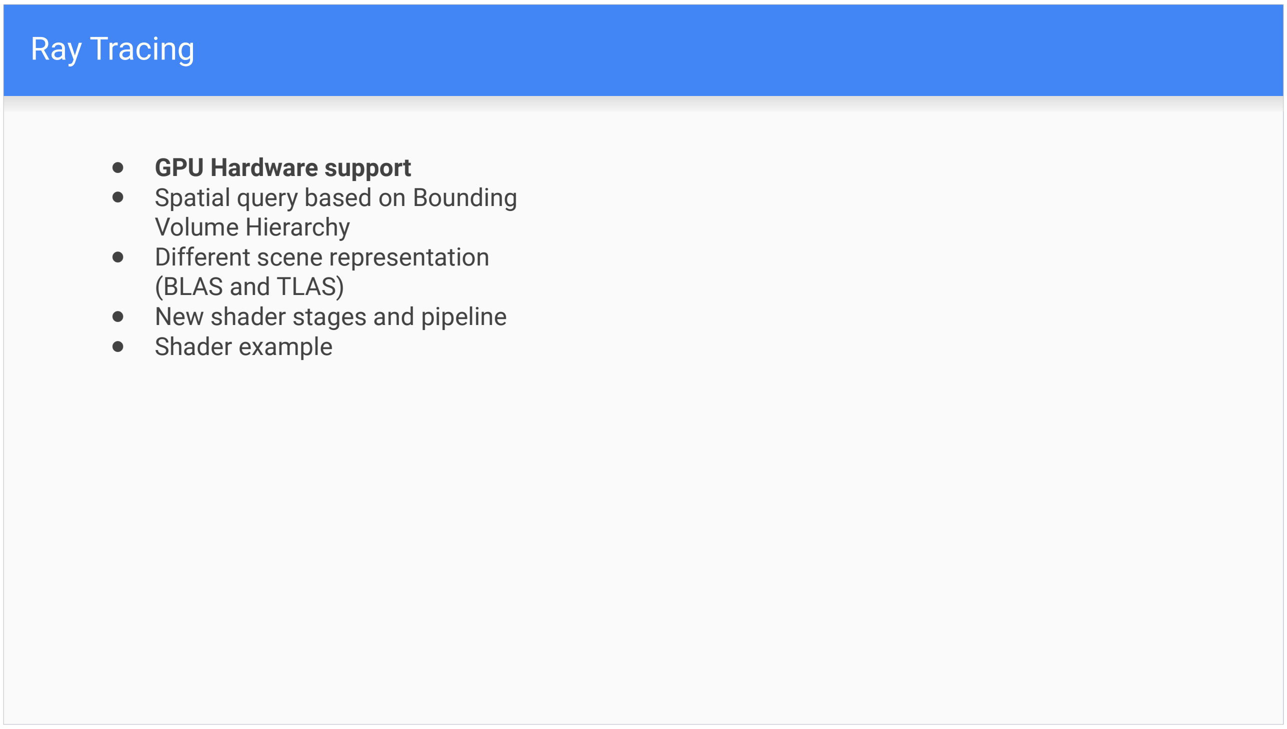
## Mesh and Task shaders



And here’s another screenshot after we froze the camera. As you can see, this algorithm can remove a lot of geometry, which leads to performance improvements. The model we are using here is also not optimized (the floor and some of the walls are a single triangle) and we could reduce the processing further.
Occlusion culling is not a solved problem and we encourage you to explore this area further.

## Mesh and Task shaders

Questions?



## Ray Tracing

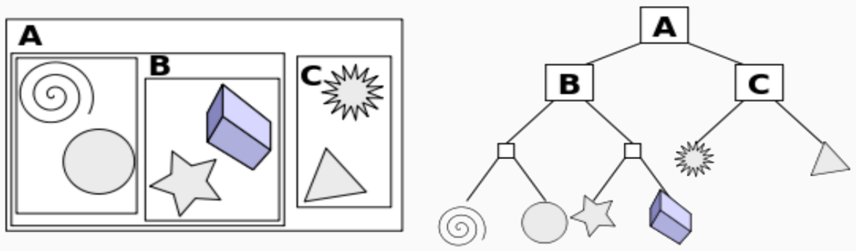
- **GPU Hardware support**
- Spatial query based on Bounding Volume Hierarchy
- Different scene representation (BLAS and TLAS)
- New shader stages and pipeline
- Shader example

Ray Tracing is a way of producing visual images that is used since the dawn of computer graphics, but always needed too much processing power to be used in realtime.
The core of Ray Tracing is a way to know which surface is hit given a ray: this can be used to calculate lighting information, simulating light bounces, or other phenomena (like sound propagation, physics simulation, …).
In 2018 NVidia launched the first GPU to have so called “RT cores” to accelerate the spatial queries used in Ray Tracing.

## Ray Tracing

- GPU Hardware support
- **Spatial query based on Bounding Volume Hierarchy**
- Different scene representation (BLAS and TLAS)
- New shader stages and pipeline
- Shader example

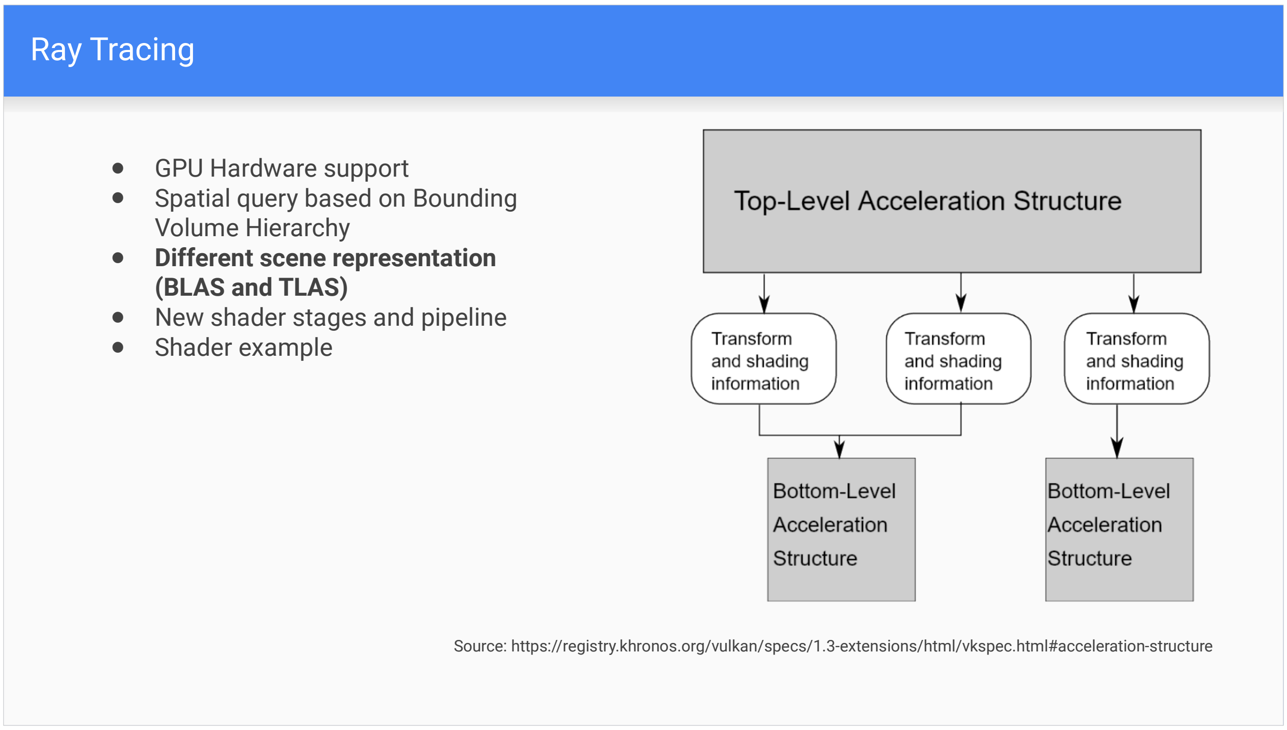
Bounding Volume Hierarchy



Source: https://en.wikipedia.org/wiki/Bounding_volume_hierarchy

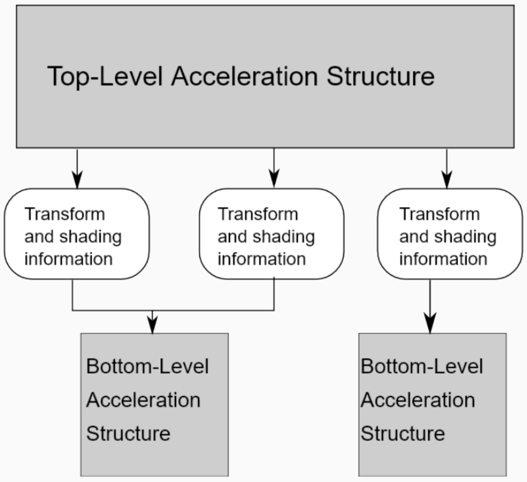
In order to perform queries we need to use a Bounding Volume Hierarchy of the scene.
This will be used to accelerate the ray-surface queries from within the shaders.
In Vulkan (and in DirectX) we cannot choose the BVH representation, thus we have an API to give Vulkan the possibility to create its internal BVH representation.

When building the BVH through the API, you can choose whether to prefer faster build times or faster trace time. Fast build times might lead to suboptimal BVHs that take longer to traverse. For faster trace times, the BVH build might take longer. This is something you have to experiment with to select the right setting for your application. It also depends on whether you need to build the BVH once (i.e. static geometry) or whether it needs to be rebuilt at runtime.



## Ray Tracing

- GPU Hardware support
- Spatial query based on Bounding Volume Hierarchy
- **Different scene representation (BLAS and TLAS)**
- New shader stages and pipeline
- Shader example



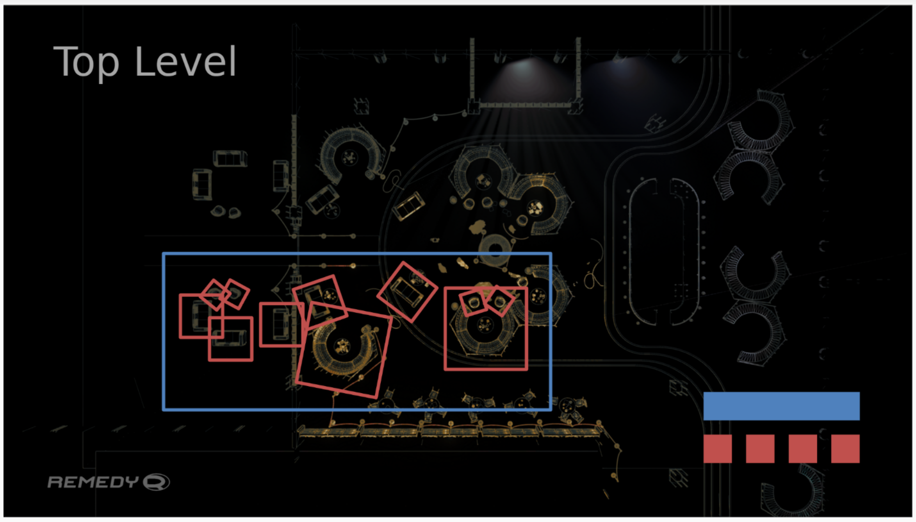
Top-Level Acceleration Structure
Transform and shading information
Transform and shading information
Transform and shading information
Bottom-Level Acceleration Structure
Bottom-Level Acceleration Structure

Source: https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html#acceleration-structure

Vulkan uses a representation based on Bottom Level Acceleration Structure (BLAS) and Top Level Acceleration Structure (TLAS).
BLAS contains the geometric data (triangles) and the TLAS the transforms and shading information.
Think of the BLAS as the mesh, and the TLAS as a scene graph, where each node of the TLAS defining a transform.

Next we will see how we can build the BLAS and the TLAS. There will be a lot of code on the next few slides, but don't worry about reading it now. It's there for reference when you revisit these slides.

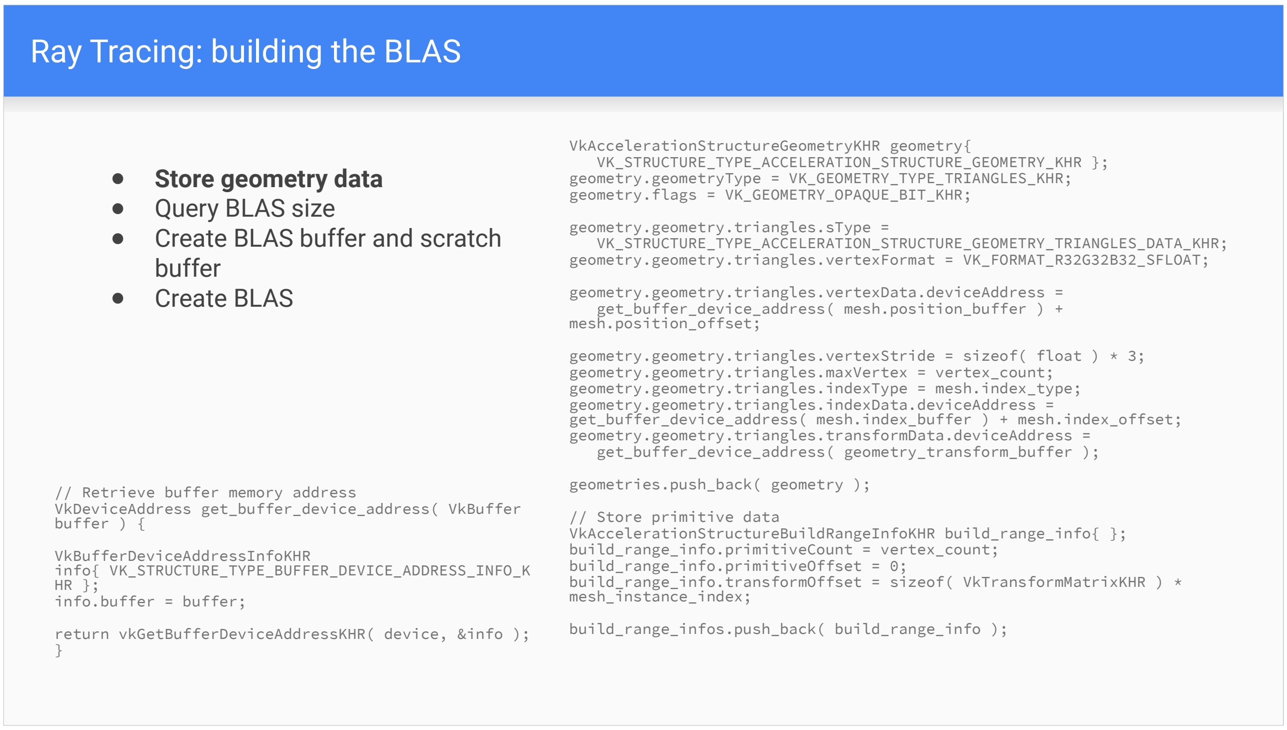
## Ray Tracing



The Latest Graphics Technology in Remedy's Northlight Engine, Tatu Aalto, GDC, 2018

In our example we will build a TLAS with one instance, but the goal of a BLAS is to define geometry once and re-use it multiple times with different transforms.
Here for instance, each sofa is a single BLAS instanced multiple times. The same applies to the chairs and other geometry. Each instance is then grouped into a TLAS.

We are going to show how to use the Vulkan API to create TLAS and BLAS next.



## Ray Tracing: building the BLAS

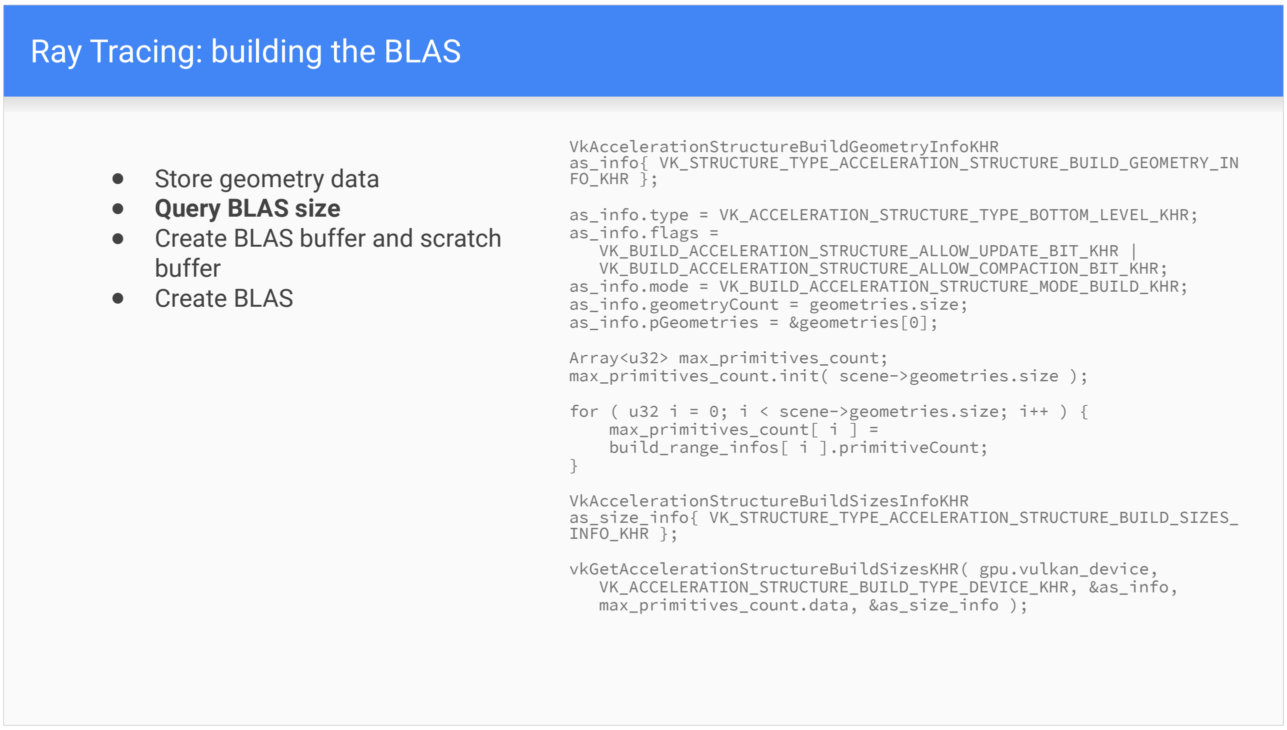
- **Store geometry data**
- Query BLAS size
- Create BLAS buffer and scratch buffer
- Create BLAS

```
// Retrieve buffer memory address
VkDeviceAddress get_buffer_device_address( VkBuffer
buffer ) {

VkBufferDeviceAddressInfoKHR
info{ VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO_K
HR };
info.buffer = buffer;

return vkGetBufferDeviceAddressKHR( device, &info );
}
```

```
VkAccelerationStructureGeometryKHR geometry{
   VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_KHR };
geometry.geometryType = VK_GEOMETRY_TYPE_TRIANGLES_KHR;
geometry.flags = VK_GEOMETRY_OPAQUE_BIT_KHR;

geometry.geometry.triangles.sType =
   VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_TRIANGLES_DATA_KHR;
geometry.geometry.triangles.vertexFormat = VK_FORMAT_R32G32B32_SFLOAT;

geometry.geometry.triangles.vertexData.deviceAddress =
   get_buffer_device_address( mesh.position_buffer ) +
mesh.position_offset;

geometry.geometry.triangles.vertexStride = sizeof( float ) * 3;
geometry.geometry.triangles.maxVertex = vertex_count;
geometry.geometry.triangles.indexType = mesh.index_type;
geometry.geometry.triangles.indexData.deviceAddress =
get_buffer_device_address( mesh.index_buffer ) + mesh.index_offset;
geometry.geometry.triangles.transformData.deviceAddress =
   get_buffer_device_address( geometry_transform_buffer );

geometries.push_back( geometry );

// Store primitive data
VkAccelerationStructureBuildRangeInfoKHR build_range_info{ };
build_range_info.primitiveCount = vertex_count;
build_range_info.primitiveOffset = 0;
build_range_info.transformOffset = sizeof( VkTransformMatrixKHR ) *
mesh_instance_index;

build_range_infos.push_back( build_range_info );
```

First we need to define the geometric data of the BLAS. For each mesh we provide vertex and index data as we would do for traditional draws.
To simplify things, we are treating the scene as static, thus we provide an array of transforms as well, this time one for each mesh instance.
Use the helper method (get_buffer_device_address) to get the actual memory address of the vulkan buffers.
Primitive count and local transforms are defined with the VkAccelerationStructureBuildRangeInfoKHR structure.

We store an array of geometries and build range informations for the next steps.

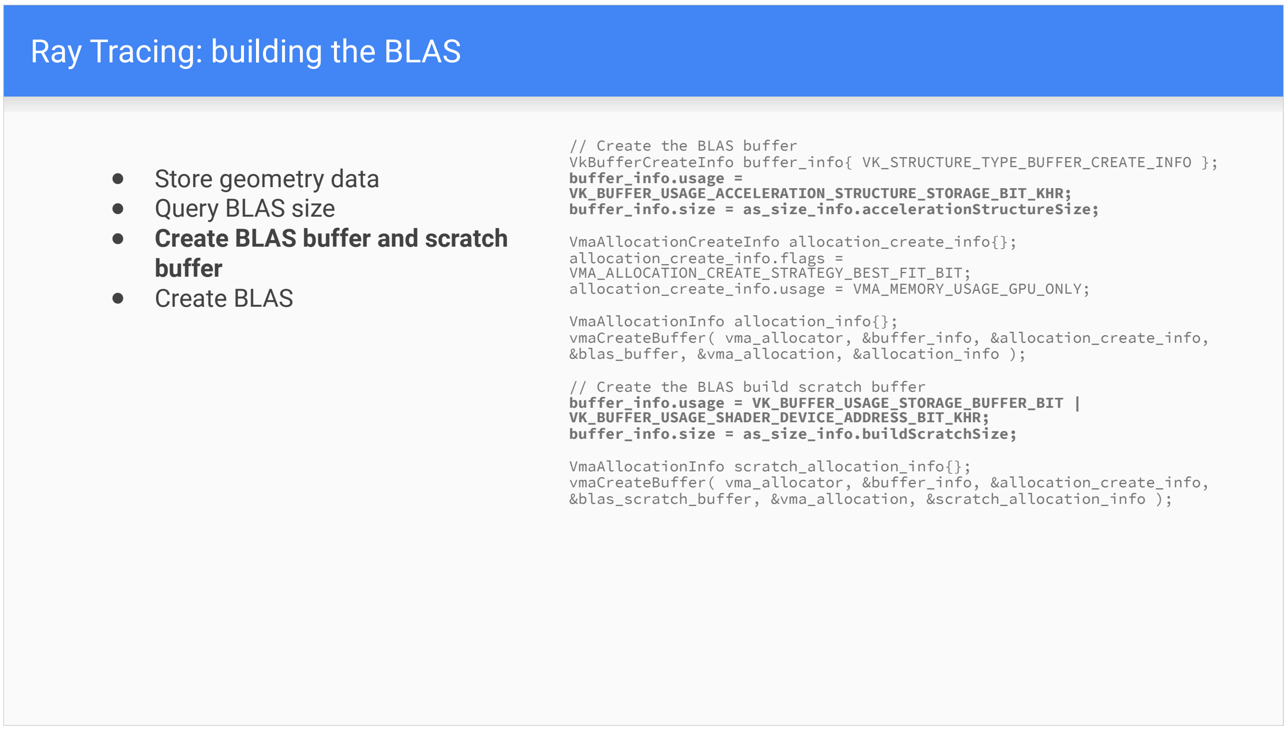


## Ray Tracing: building the BLAS

- Store geometry data
- **Query BLAS size**
- Create BLAS buffer and scratch buffer
- Create BLAS

```
VkAccelerationStructureBuildGeometryInfoKHR
as_info{ VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_GEOMETRY_IN
FO_KHR };

as_info.type = VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR;
as_info.flags =
   VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_UPDATE_BIT_KHR |
   VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_COMPACTION_BIT_KHR;
as_info.mode = VK_BUILD_ACCELERATION_STRUCTURE_MODE_BUILD_KHR;
as_info.geometryCount = geometries.size;
as_info.pGeometries = &geometries[0];

Array<u32> max_primitives_count;
max_primitives_count.init( scene->geometries.size );

for ( u32 i = 0; i < scene->geometries.size; i++ ) {
    max_primitives_count[ i ] =
    build_range_infos[ i ].primitiveCount;
}

VkAccelerationStructureBuildSizesInfoKHR
as_size_info{ VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_SIZES_
INFO_KHR };

vkGetAccelerationStructureBuildSizesKHR( gpu.vulkan_device,
   VK_ACCELERATION_STRUCTURE_BUILD_TYPE_DEVICE_KHR, &as_info,
   max_primitives_count.data, &as_size_info );
```

To query the BLAS size, we build a structure that uses the geometry data we cached previously and an array of primitive counts for each mesh.
We can call vkGetAccelerationStructureBuildSizesKHR method to fill another structure containing the necessary memory.

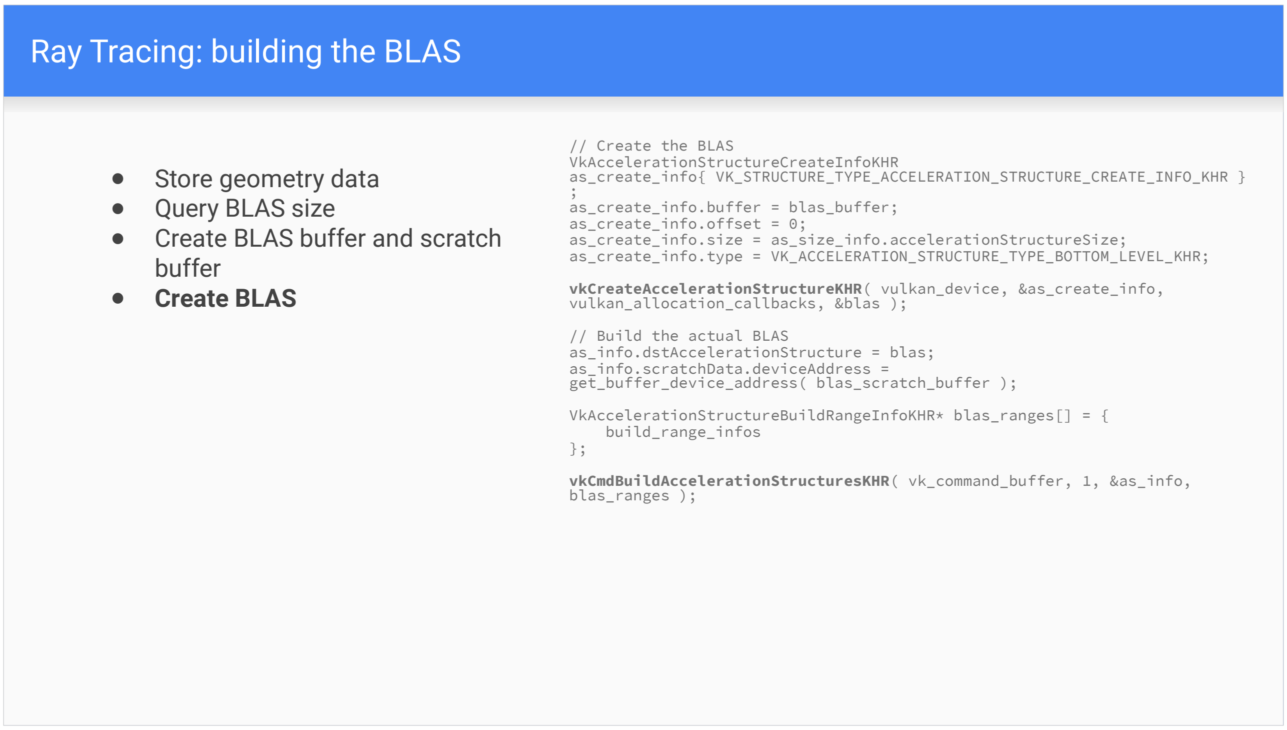


## Ray Tracing: building the BLAS

- Store geometry data
- Query BLAS size
- **Create BLAS buffer and scratch buffer**
- Create BLAS

```
// Create the BLAS buffer
VkBufferCreateInfo buffer_info{ VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO };
buffer_info.usage =
VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_STORAGE_BIT_KHR;
buffer_info.size = as_size_info.accelerationStructureSize;

VmaAllocationCreateInfo allocation_create_info{};
allocation_create_info.flags =
VMA_ALLOCATION_CREATE_STRATEGY_BEST_FIT_BIT;
allocation_create_info.usage = VMA_MEMORY_USAGE_GPU_ONLY;

VmaAllocationInfo allocation_info{};
vmaCreateBuffer( vma_allocator, &buffer_info, &allocation_create_info,
&blas_buffer, &vma_allocation, &allocation_info );

// Create the BLAS build scratch buffer
buffer_info.usage = VK_BUFFER_USAGE_STORAGE_BUFFER_BIT |
VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT_KHR;
buffer_info.size = as_size_info.buildScratchSize;

VmaAllocationInfo scratch_allocation_info{};
vmaCreateBuffer( vma_allocator, &buffer_info, &allocation_create_info,
&blas_scratch_buffer, &vma_allocation, &scratch_allocation_info );
```

The BLAS needs two buffers to be created: one to store the actual BLAS, the other used as scratch memory to build the BLAS.
We can create those two buffers with the sizes coming from the previous query, and using the flags outlined in the code.



## Ray Tracing: building the BLAS

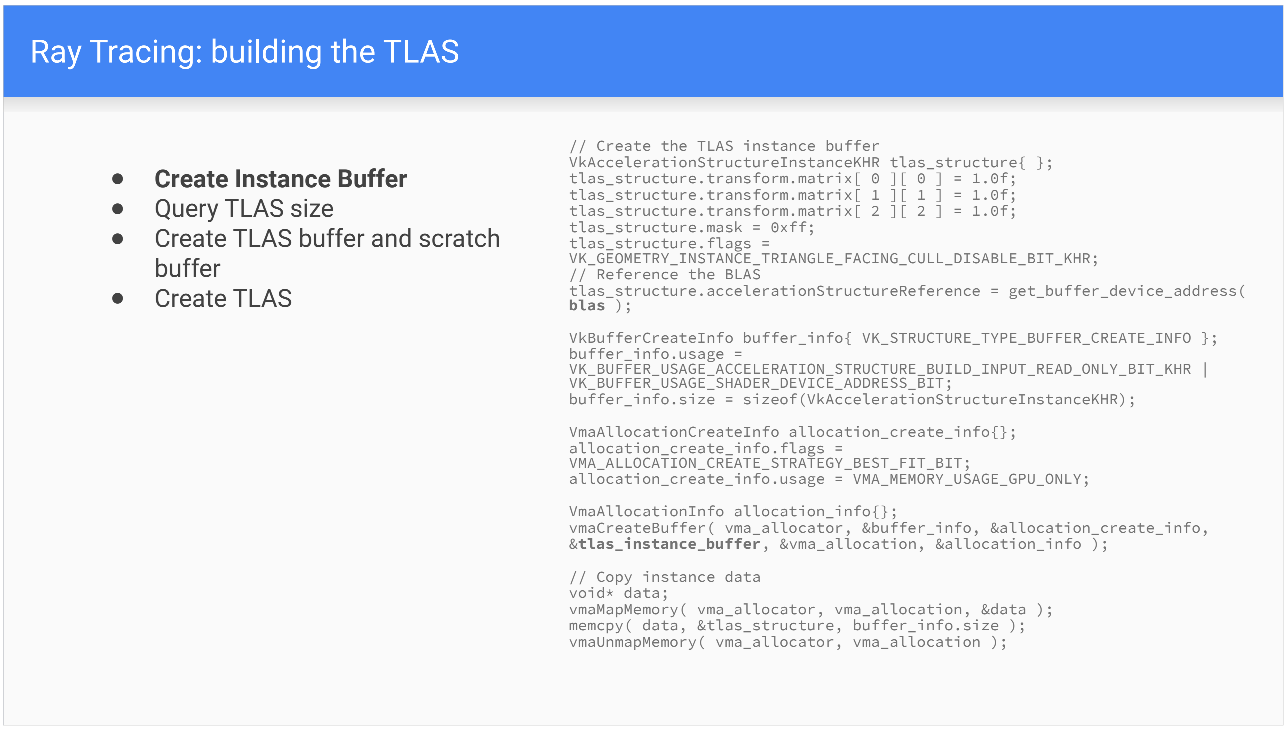
- Store geometry data
- Query BLAS size
- Create BLAS buffer and scratch buffer
- **Create BLAS**

```
// Create the BLAS
VkAccelerationStructureCreateInfoKHR
as_create_info{ VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO_KHR }
;
as_create_info.buffer = blas_buffer;
as_create_info.offset = 0;
as_create_info.size = as_size_info.accelerationStructureSize;
as_create_info.type = VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR;

vkCreateAccelerationStructureKHR( vulkan_device, &as_create_info,
vulkan_allocation_callbacks, &blas );

// Build the actual BLAS
as_info.dstAccelerationStructure = blas;
as_info.scratchData.deviceAddress =
get_buffer_device_address( blas_scratch_buffer );

VkAccelerationStructureBuildRangeInfoKHR* blas_ranges[] = {
    build_range_infos
};

vkCmdBuildAccelerationStructuresKHR( vk_command_buffer, 1, &as_info,
blas_ranges );
```

With all those informations, we can finally build the BLAS.
First we define the BLAS structure, then we actually build it.

It’s important to note that we can build multiple BLAS at the same time, but we can’t build a BLAS and TLAS at the same time. This is because we need to make sure the BLAS build has completed.

Next we can create our TLAS!



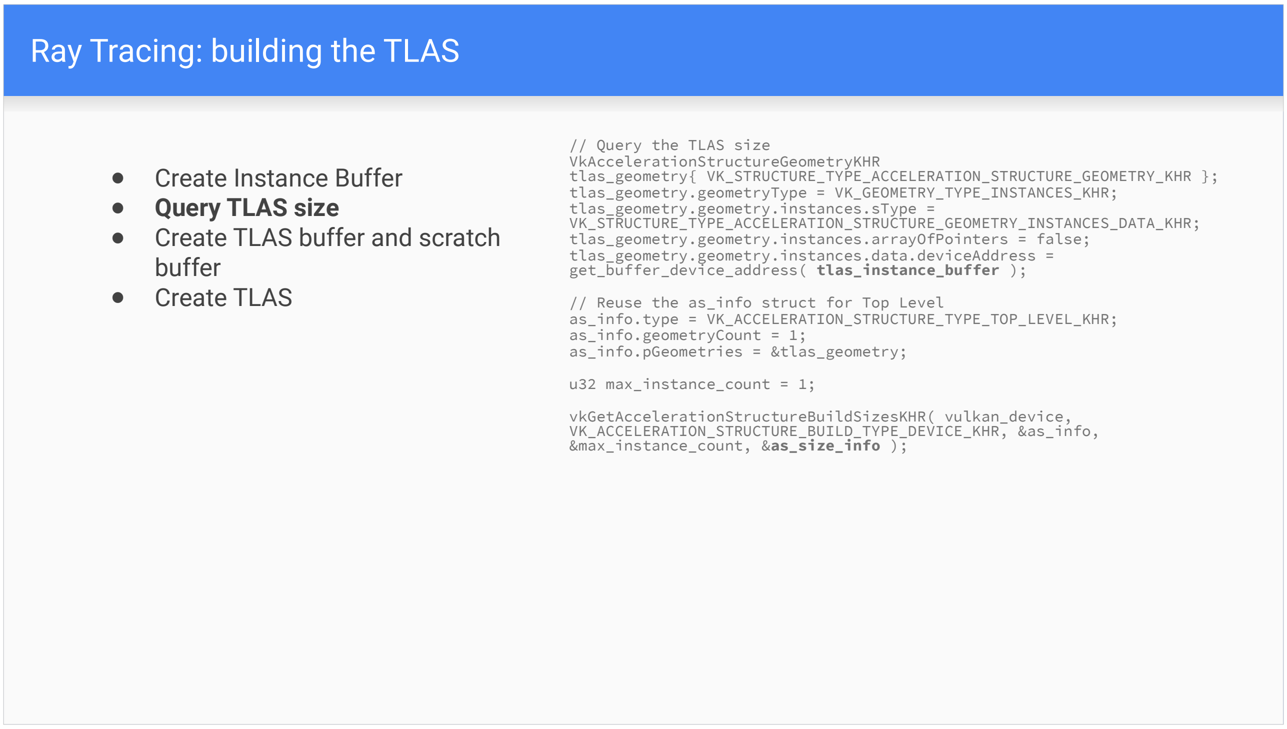
## Ray Tracing: building the TLAS

- **Create Instance Buffer**
- Query TLAS size
- Create TLAS buffer and scratch buffer
- Create TLAS

```
// Create the TLAS instance buffer
VkAccelerationStructureInstanceKHR tlas_structure{ };
tlas_structure.transform.matrix[ 0 ][ 0 ] = 1.0f;
tlas_structure.transform.matrix[ 1 ][ 1 ] = 1.0f;
tlas_structure.transform.matrix[ 2 ][ 2 ] = 1.0f;
tlas_structure.mask = 0xff;
tlas_structure.flags =
VK_GEOMETRY_INSTANCE_TRIANGLE_FACING_CULL_DISABLE_BIT_KHR;
// Reference the BLAS
tlas_structure.accelerationStructureReference = get_buffer_device_address(
blas );

VkBufferCreateInfo buffer_info{ VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO };
buffer_info.usage =
VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR |
VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT;
buffer_info.size = sizeof(VkAccelerationStructureInstanceKHR);

VmaAllocationCreateInfo allocation_create_info{};
allocation_create_info.flags =
VMA_ALLOCATION_CREATE_STRATEGY_BEST_FIT_BIT;
allocation_create_info.usage = VMA_MEMORY_USAGE_GPU_ONLY;

VmaAllocationInfo allocation_info{};
vmaCreateBuffer( vma_allocator, &buffer_info, &allocation_create_info,
&tlas_instance_buffer, &vma_allocation, &allocation_info );

// Copy instance data
void* data;
vmaMapMemory( vma_allocator, vma_allocation, &data );
memcpy( data, &tlas_structure, buffer_info.size );
vmaUnmapMemory( vma_allocator, vma_allocation );
```

For the TLAS creation, we will simplify things and assume we are dealing with a static scene, so we can focus on the important parts of the code.

First we create an instance buffer where we copy the Vulkan Instance struct.

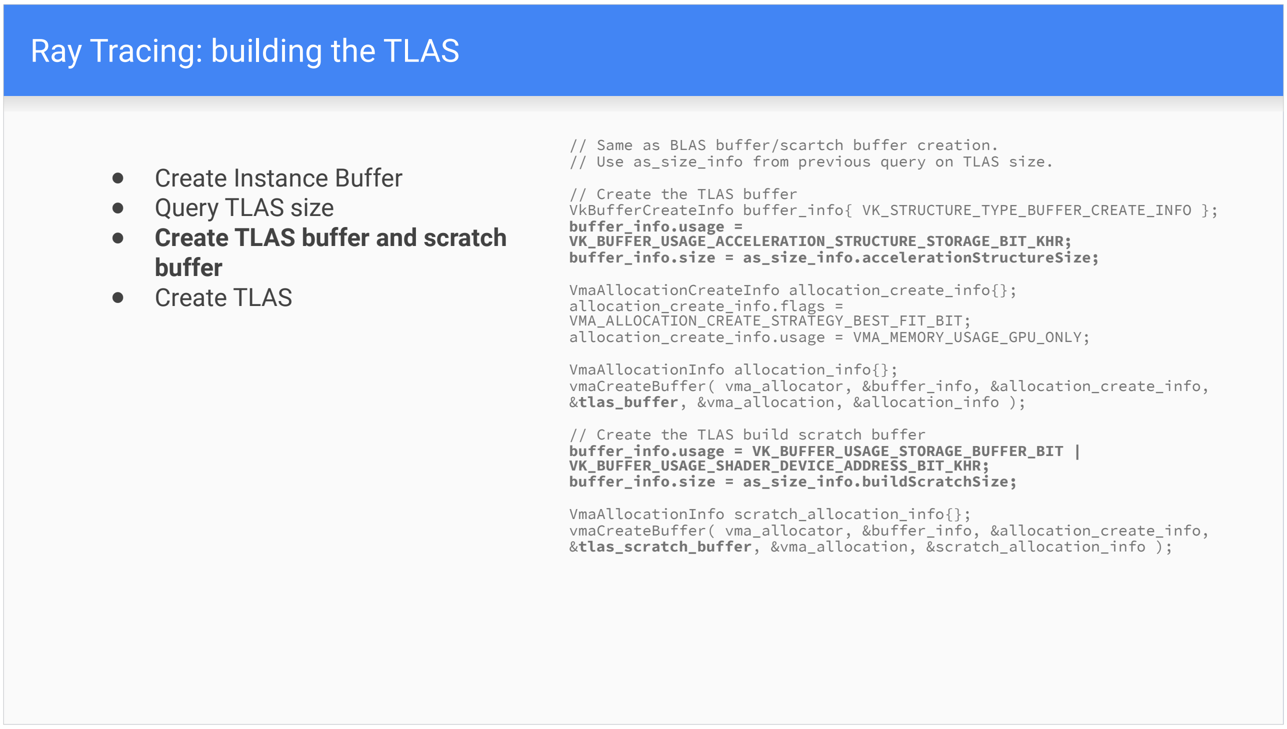


## Ray Tracing: building the TLAS

- Create Instance Buffer
- **Query TLAS size**
- Create TLAS buffer and scratch buffer
- Create TLAS

```
// Query the TLAS size
VkAccelerationStructureGeometryKHR
tlas_geometry{ VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_KHR };
tlas_geometry.geometryType = VK_GEOMETRY_TYPE_INSTANCES_KHR;
tlas_geometry.geometry.instances.sType =
VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_INSTANCES_DATA_KHR;
tlas_geometry.geometry.instances.arrayOfPointers = false;
tlas_geometry.geometry.instances.data.deviceAddress =
get_buffer_device_address( tlas_instance_buffer );

// Reuse the as_info struct for Top Level
as_info.type = VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR;
as_info.geometryCount = 1;
as_info.pGeometries = &tlas_geometry;

u32 max_instance_count = 1;

vkGetAccelerationStructureBuildSizesKHR( vulkan_device,
VK_ACCELERATION_STRUCTURE_BUILD_TYPE_DEVICE_KHR, &as_info,
&max_instance_count, &as_size_info );
```

Based on the instance buffer, we can query the TLAS size.
As previously stated, we have only 1 instance containing all the geometries in the scene within the BLAS.

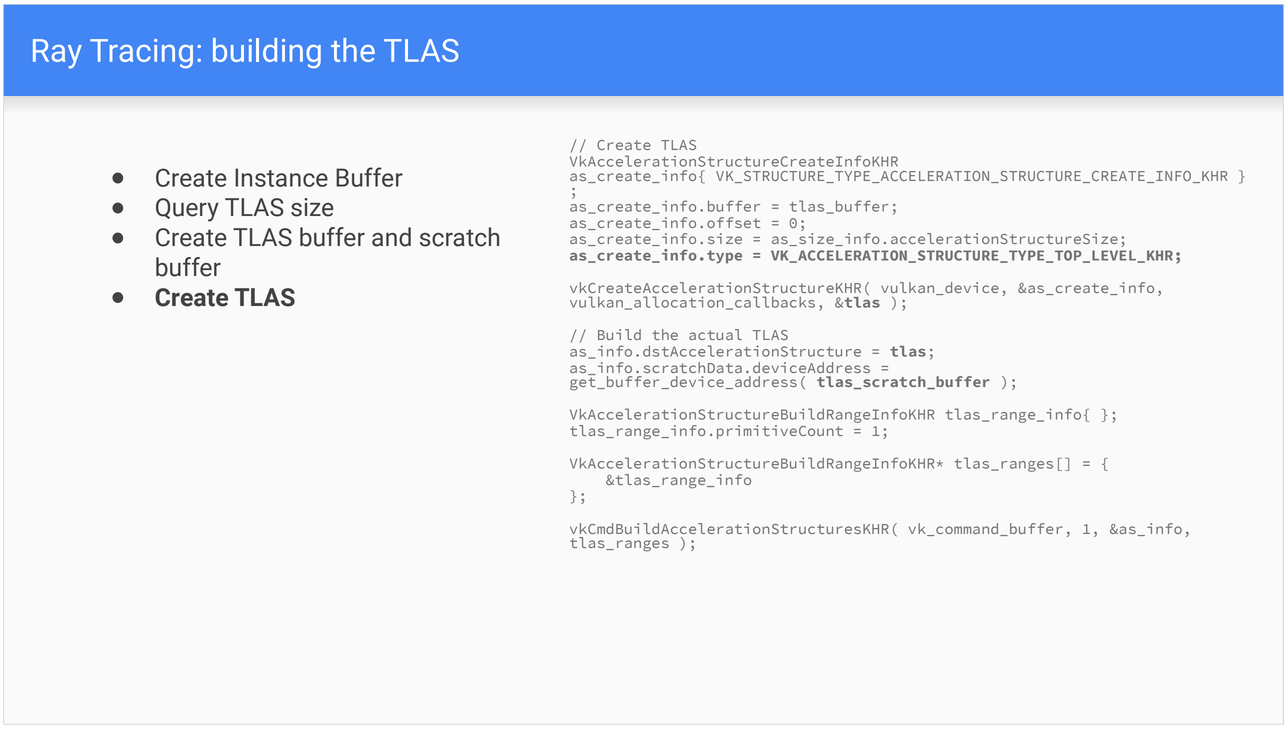


## Ray Tracing: building the TLAS

- Create Instance Buffer
- Query TLAS size
- **Create TLAS buffer and scratch buffer**
- Create TLAS

```
// Same as BLAS buffer/scartch buffer creation.
// Use as_size_info from previous query on TLAS size.

// Create the TLAS buffer
VkBufferCreateInfo buffer_info{ VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO };
buffer_info.usage =
VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_STORAGE_BIT_KHR;
buffer_info.size = as_size_info.accelerationStructureSize;

VmaAllocationCreateInfo allocation_create_info{};
allocation_create_info.flags =
VMA_ALLOCATION_CREATE_STRATEGY_BEST_FIT_BIT;
allocation_create_info.usage = VMA_MEMORY_USAGE_GPU_ONLY;

VmaAllocationInfo allocation_info{};
vmaCreateBuffer( vma_allocator, &buffer_info, &allocation_create_info,
&tlas_buffer, &vma_allocation, &allocation_info );

// Create the TLAS build scratch buffer
buffer_info.usage = VK_BUFFER_USAGE_STORAGE_BUFFER_BIT |
VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT_KHR;
buffer_info.size = as_size_info.buildScratchSize;

VmaAllocationInfo scratch_allocation_info{};
vmaCreateBuffer( vma_allocator, &buffer_info, &allocation_create_info,
&tlas_scratch_buffer, &vma_allocation, &scratch_allocation_info );
```

This code is the same as the creation of the BLAS buffer and scratch buffers, but using the result from the previous query to know the sizes.



## Ray Tracing: building the TLAS

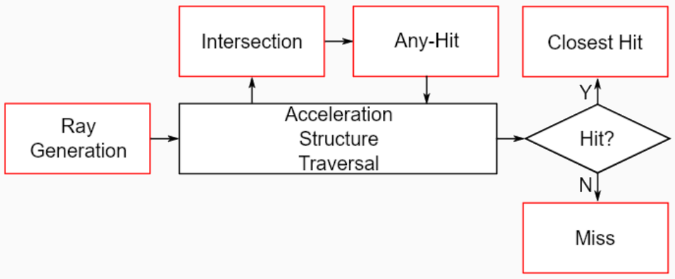
- Create Instance Buffer
- Query TLAS size
- Create TLAS buffer and scratch buffer
- **Create TLAS**

```
// Create TLAS
VkAccelerationStructureCreateInfoKHR
as_create_info{ VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO_KHR }
;
as_create_info.buffer = tlas_buffer;
as_create_info.offset = 0;
as_create_info.size = as_size_info.accelerationStructureSize;
as_create_info.type = VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR;

vkCreateAccelerationStructureKHR( vulkan_device, &as_create_info,
vulkan_allocation_callbacks, &tlas );

// Build the actual TLAS
as_info.dstAccelerationStructure = tlas;
as_info.scratchData.deviceAddress =
get_buffer_device_address( tlas_scratch_buffer );

VkAccelerationStructureBuildRangeInfoKHR tlas_range_info{ };
tlas_range_info.primitiveCount = 1;

VkAccelerationStructureBuildRangeInfoKHR* tlas_ranges[] = {
    &tlas_range_info
};

vkCmdBuildAccelerationStructuresKHR( vk_command_buffer, 1, &as_info,
tlas_ranges );
```

We can finally build the TLAS. The procedure is almost identical to the BLAS creation: first we create the TLAS object, then we build it.
In this case we have only one instance, used for all the static scene.

## Ray Tracing: shader stages

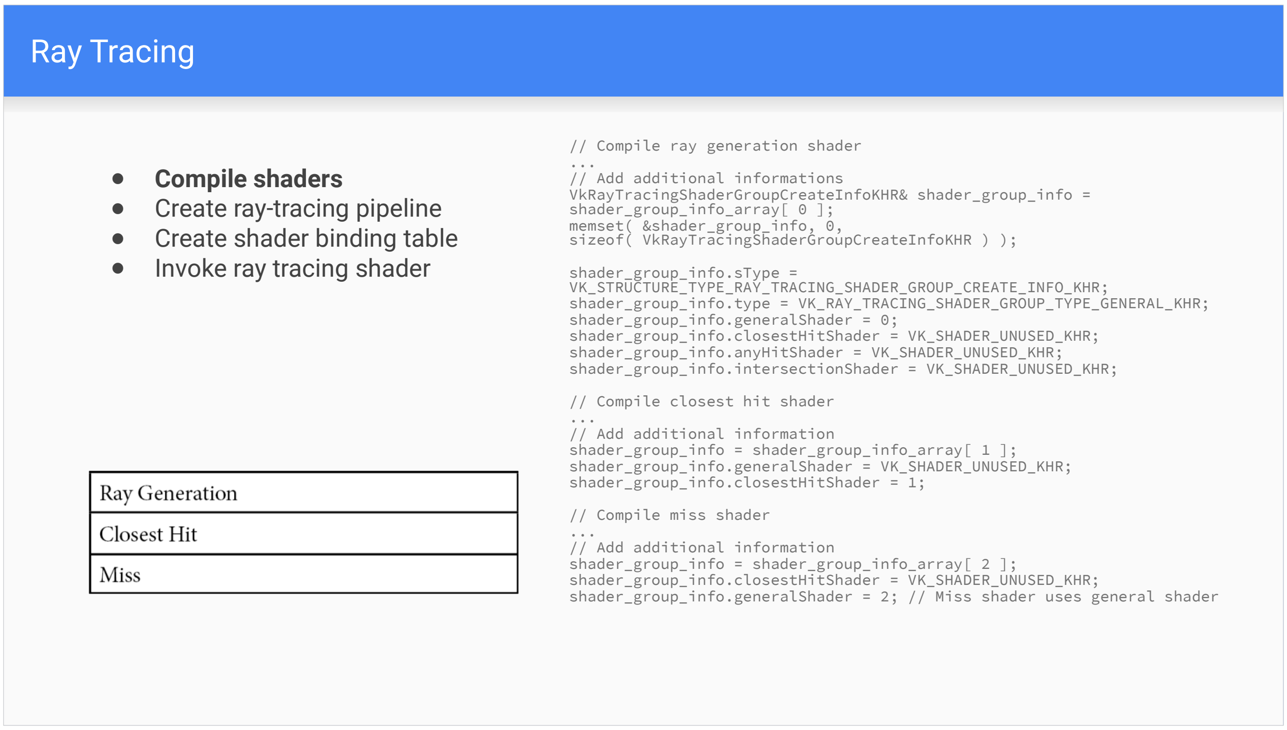
- GPU Hardware support
- Spatial query based on Bounding Volume Hierarchy
- Different scene representation (BLAS and TLAS)
- **New shader stages and pipeline**
- Shader example



Source: https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html#ray-tracing

Now that we have covered how to create the data used by ray-tracing, we can move to the new shader stages required which are used to traverse the scene and perform computation based on the result of the traversal.

- In the Ray Generation stage, we setup a ray and what kind of data should it store (called payload) and then launch the query.
- Then there is the intersection shader, that gives us the possibility to customize the geometry primitive used. In Vulkan is either a triangle or an AABB. This type of shaderse is used mainly if you want to specify your own intersection function, i.e. for an SDF
- The Any Hit shader will run after the Intersection stage and determine whether we need to process this ray further.
- Closes Hit shaders will run if the traversal has found an intersection
- Miss shaders will run if no hit has been found
- There is also a special shader, callable, that defines which shaders can be called from one of those shaders.



## Ray Tracing

- **Compile shaders**
- Create ray-tracing pipeline
- Create shader binding table
- Invoke ray tracing shader

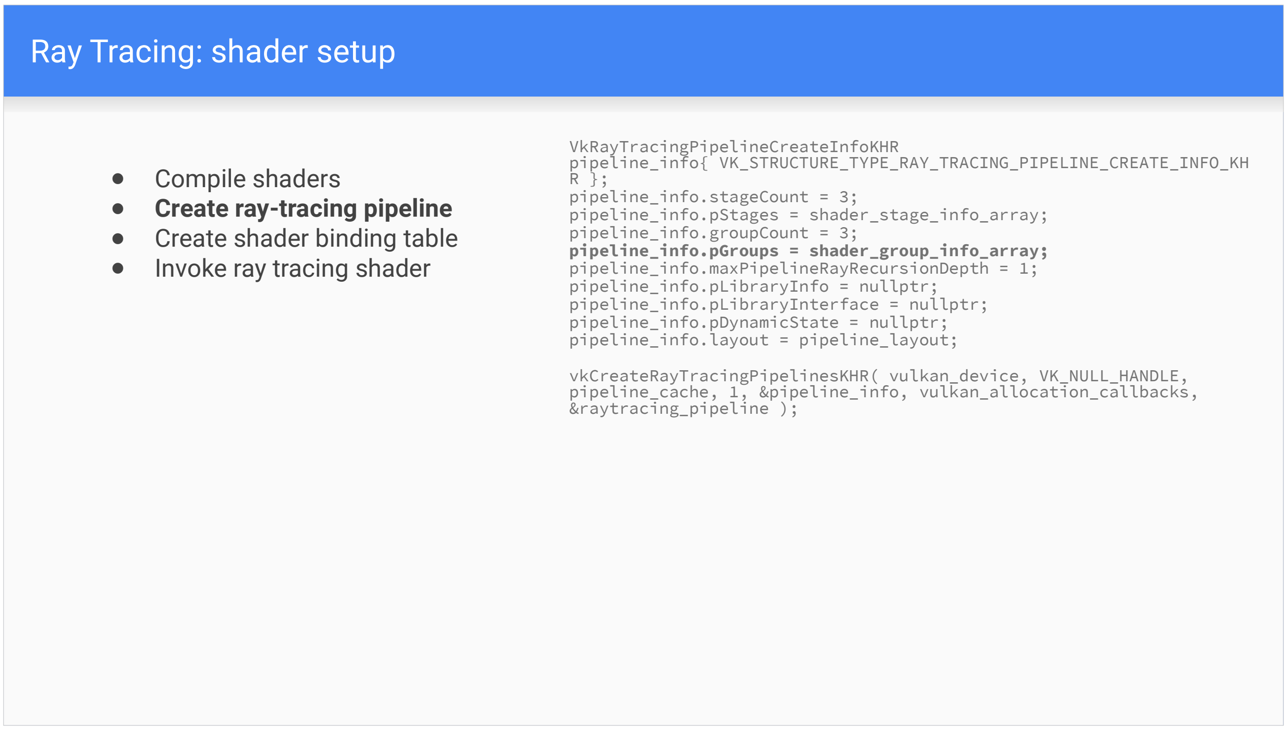
```
// Compile ray generation shader
...
// Add additional informations
VkRayTracingShaderGroupCreateInfoKHR& shader_group_info =
shader_group_info_array[ 0 ];
memset( &shader_group_info, 0,
sizeof( VkRayTracingShaderGroupCreateInfoKHR ) );

shader_group_info.sType =
VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_KHR;
shader_group_info.type = VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_KHR;
shader_group_info.generalShader = 0;
shader_group_info.closestHitShader = VK_SHADER_UNUSED_KHR;
shader_group_info.anyHitShader = VK_SHADER_UNUSED_KHR;
shader_group_info.intersectionShader = VK_SHADER_UNUSED_KHR;

// Compile closest hit shader
...
// Add additional information
shader_group_info = shader_group_info_array[ 1 ];
shader_group_info.generalShader = VK_SHADER_UNUSED_KHR;
shader_group_info.closestHitShader = 1;

// Compile miss shader
...
// Add additional information
shader_group_info = shader_group_info_array[ 2 ];
shader_group_info.closestHitShader = VK_SHADER_UNUSED_KHR;
shader_group_info.generalShader = 2; // Miss shader uses general shader
```

| |
|---|
| Ray Generation |
| Closest Hit |
| Miss |

For each shader stage, we compile the shaders as we would do normally and cache additional data  to create shader groups.

This is a departure compared to traditional pipelines and it provides a lot of flexibility as it allows us to invoke different shaders depending on the circumstances.
For example, different closest-hit shaders can run depending on the material of the geometry that we hit, and this information is embedded in the instance data.

The image on the bottom-left illustrates the table produced by the example code.
Production pipelines can have hundreds of shaders, depending on the complexity of the scene.



## Ray Tracing: shader setup

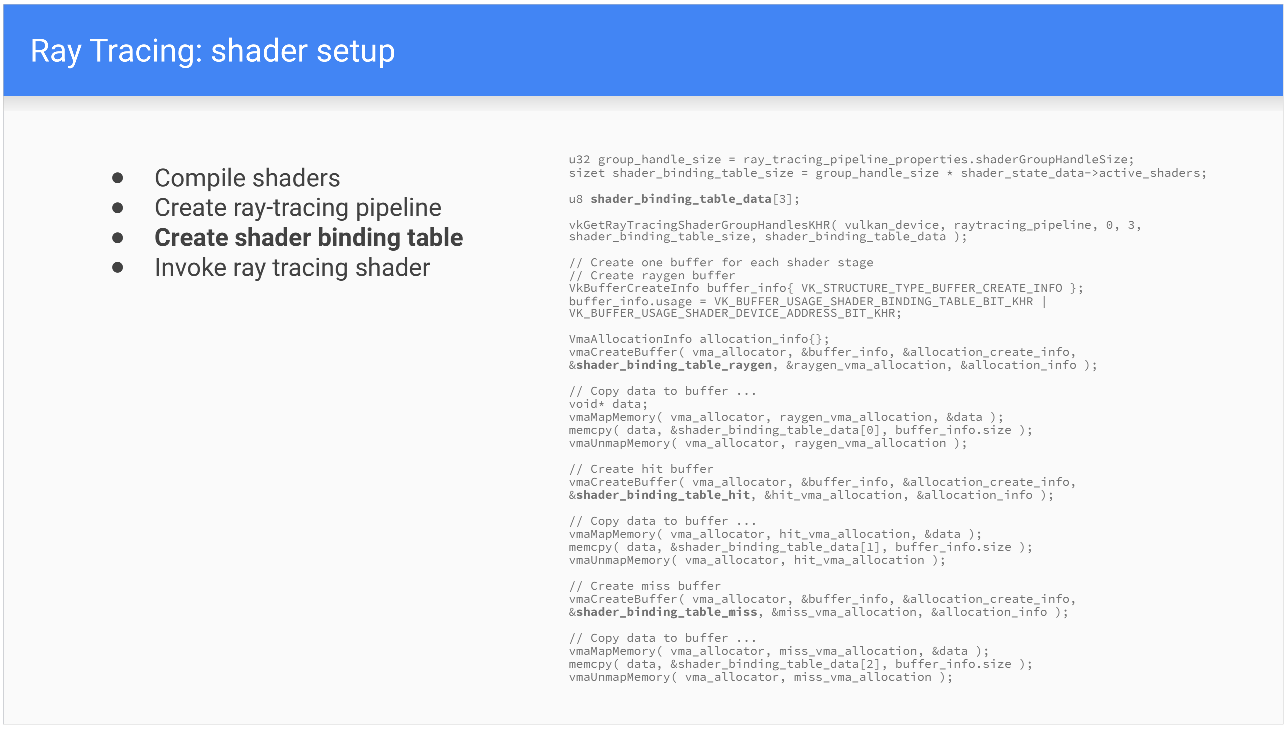
- Compile shaders
- **Create ray-tracing pipeline**
- Create shader binding table
- Invoke ray tracing shader

```
VkRayTracingPipelineCreateInfoKHR
pipeline_info{ VK_STRUCTURE_TYPE_RAY_TRACING_PIPELINE_CREATE_INFO_KH
R };
pipeline_info.stageCount = 3;
pipeline_info.pStages = shader_stage_info_array;
pipeline_info.groupCount = 3;
pipeline_info.pGroups = shader_group_info_array;
pipeline_info.maxPipelineRayRecursionDepth = 1;
pipeline_info.pLibraryInfo = nullptr;
pipeline_info.pLibraryInterface = nullptr;
pipeline_info.pDynamicState = nullptr;
pipeline_info.layout = pipeline_layout;

vkCreateRayTracingPipelinesKHR( vulkan_device, VK_NULL_HANDLE,
pipeline_cache, 1, &pipeline_info, vulkan_allocation_callbacks,
&raytracing_pipeline );
```

We now create the ray-tracing pipeline, using the additional group info calculated in the previous step.

It’s possible for ray-tracing shaders to be invoked recursively (i.e. multi-bounce lighting). We need to specify in advance the maximum level of recursion as the driver and compiler need to account for maximum size of the stack.

Another aspect to note here is that ray-tracing shaders can be compiled into libraries that can re-used across multiple pipelines. This can improve the creation time of a new pipeline.

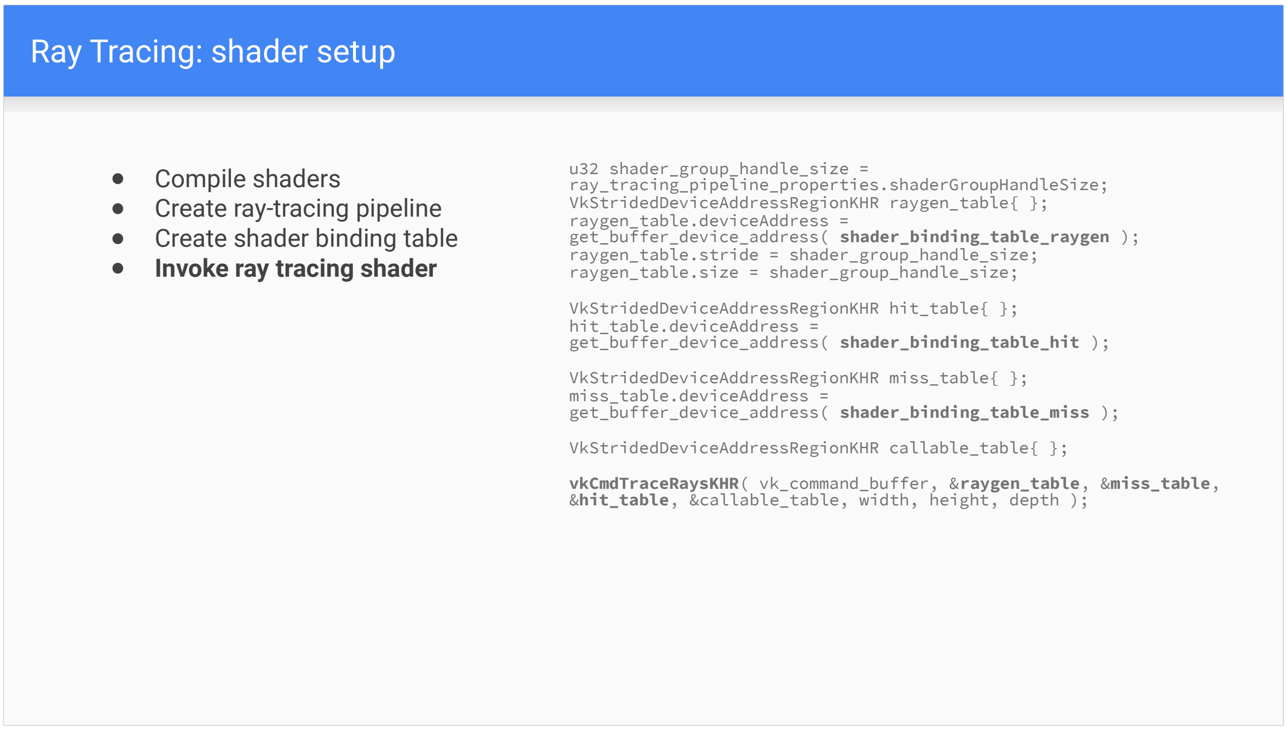


## Ray Tracing: shader setup

- Compile shaders
- Create ray-tracing pipeline
- **Create shader binding table**
- Invoke ray tracing shader

```
u32 group_handle_size = ray_tracing_pipeline_properties.shaderGroupHandleSize;
sizet shader_binding_table_size = group_handle_size * shader_state_data->active_shaders;

u8 shader_binding_table_data[3];

vkGetRayTracingShaderGroupHandlesKHR( vulkan_device, raytracing_pipeline, 0, 3,
shader_binding_table_size, shader_binding_table_data );

// Create one buffer for each shader stage
// Create raygen buffer
VkBufferCreateInfo buffer_info{ VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO };
buffer_info.usage = VK_BUFFER_USAGE_SHADER_BINDING_TABLE_BIT_KHR |
VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT_KHR;

VmaAllocationInfo allocation_info{};
vmaCreateBuffer( vma_allocator, &buffer_info, &allocation_create_info,
&shader_binding_table_raygen, &raygen_vma_allocation, &allocation_info );

// Copy data to buffer ...
void* data;
vmaMapMemory( vma_allocator, raygen_vma_allocation, &data );
memcpy( data, &shader_binding_table_data[0], buffer_info.size );
vmaUnmapMemory( vma_allocator, raygen_vma_allocation );

// Create hit buffer
vmaCreateBuffer( vma_allocator, &buffer_info, &allocation_create_info,
&shader_binding_table_hit, &hit_vma_allocation, &allocation_info );

// Copy data to buffer ...
vmaMapMemory( vma_allocator, hit_vma_allocation, &data );
memcpy( data, &shader_binding_table_data[1], buffer_info.size );
vmaUnmapMemory( vma_allocator, hit_vma_allocation );

// Create miss buffer
vmaCreateBuffer( vma_allocator, &buffer_info, &allocation_create_info,
&shader_binding_table_miss, &miss_vma_allocation, &allocation_info );

// Copy data to buffer ...
vmaMapMemory( vma_allocator, miss_vma_allocation, &data );
memcpy( data, &shader_binding_table_data[2], buffer_info.size );
vmaUnmapMemory( vma_allocator, miss_vma_allocation );
```

We retrieve the shader group handles and create a buffer for each shader stage (ray generation, hit, miss).

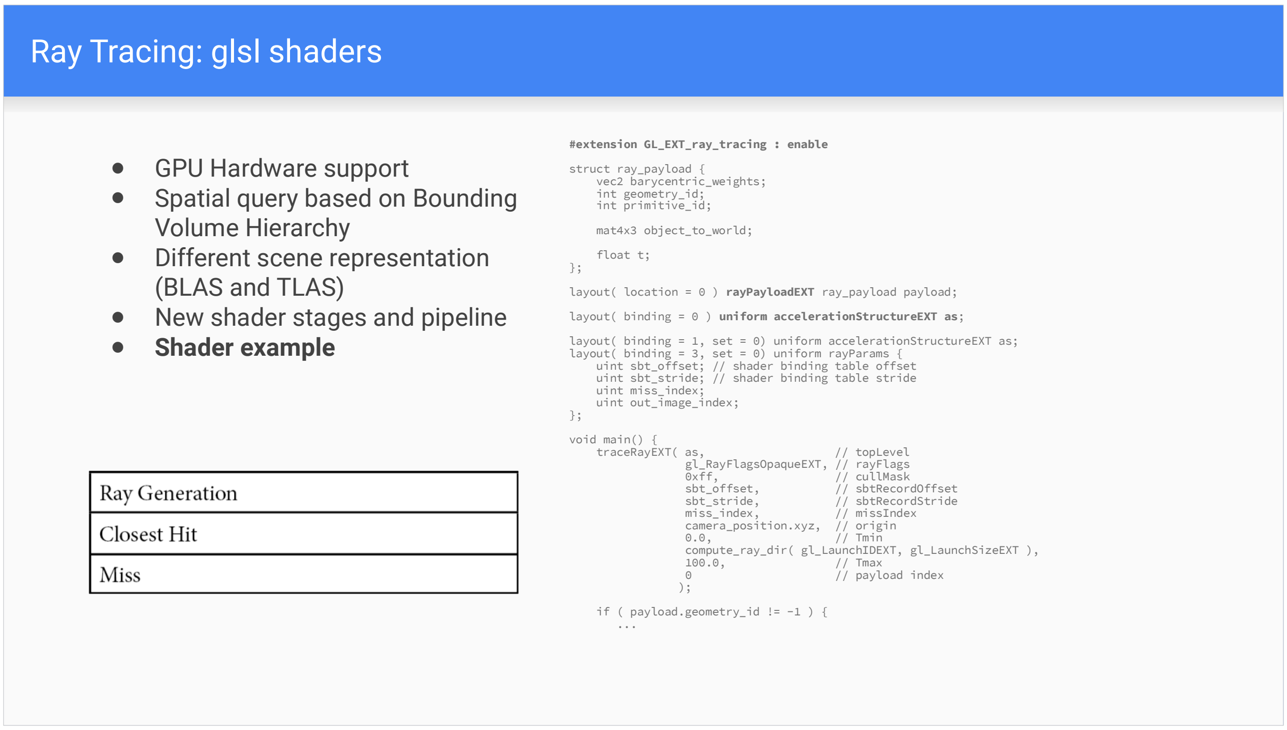


## Ray Tracing: shader setup

- Compile shaders
- Create ray-tracing pipeline
- Create shader binding table
- **Invoke ray tracing shader**

```
u32 shader_group_handle_size =
ray_tracing_pipeline_properties.shaderGroupHandleSize;
VkStridedDeviceAddressRegionKHR raygen_table{ };
raygen_table.deviceAddress =
get_buffer_device_address( shader_binding_table_raygen );
raygen_table.stride = shader_group_handle_size;
raygen_table.size = shader_group_handle_size;

VkStridedDeviceAddressRegionKHR hit_table{ };
hit_table.deviceAddress =
get_buffer_device_address( shader_binding_table_hit );

VkStridedDeviceAddressRegionKHR miss_table{ };
miss_table.deviceAddress =
get_buffer_device_address( shader_binding_table_miss );

VkStridedDeviceAddressRegionKHR callable_table{ };

vkCmdTraceRaysKHR( vk_command_buffer, &raygen_table, &miss_table,
&hit_table, &callable_table, width, height, depth );
```

As an example to invoke a ray tracing shader, we need to retrieve the device addresses of the buffers we just created, which returns a VkStridedDeviceAddressRegionKHR struct.
You can think as ray tracing shaders as a special version of compute shaders, in that you can specify the dispatch size.
Let’s say that we want to perform a raytracing of the scene and put it in a fullscreen texture. We will call vkCmdTraceRaysKHR and see the different shader table for each stage.
Notice that we have to specify an empty callable_table even if we make no use of callable shaders in this example.



## Ray Tracing: glsl shaders

- GPU Hardware support
- Spatial query based on Bounding Volume Hierarchy
- Different scene representation (BLAS and TLAS)
- New shader stages and pipeline
- **Shader example**

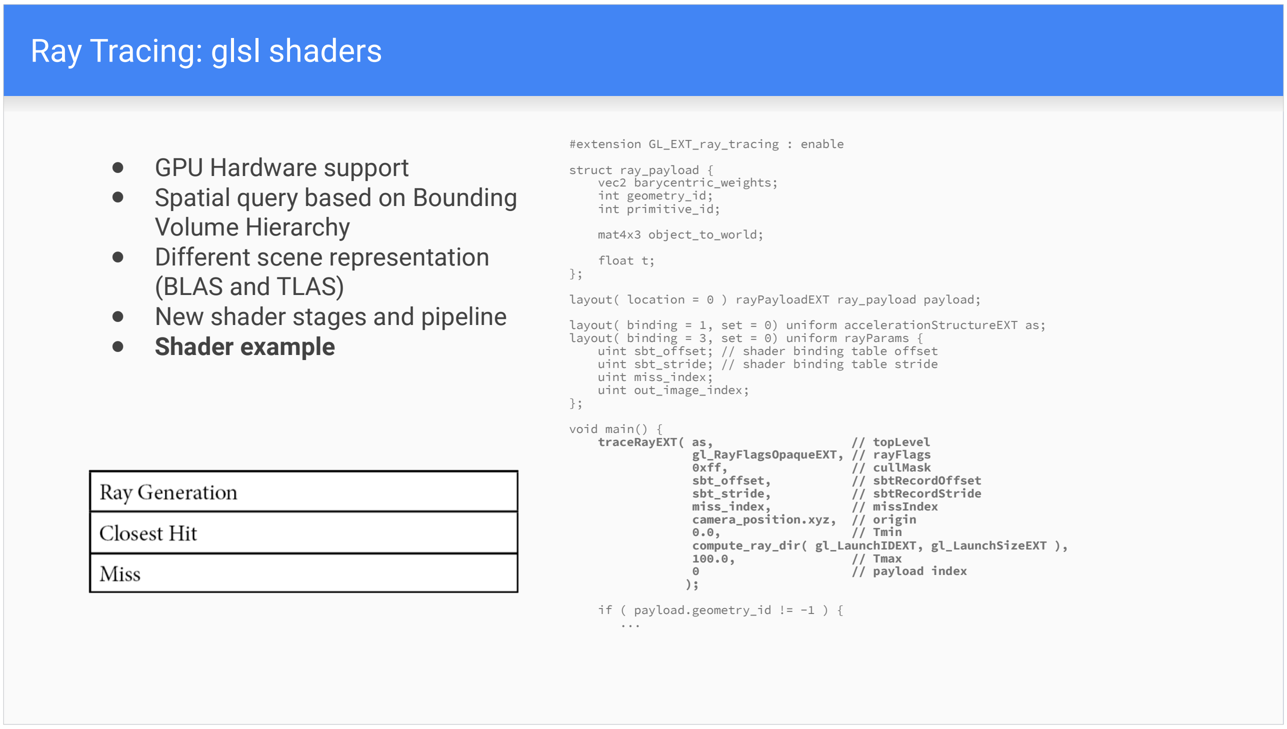
| |
|---|
| Ray Generation |
| Closest Hit |
| Miss |

```
#extension GL_EXT_ray_tracing : enable

struct ray_payload {
    vec2 barycentric_weights;
    int geometry_id;
    int primitive_id;

    mat4x3 object_to_world;

    float t;
};

layout( location = 0 ) rayPayloadEXT ray_payload payload;

layout( binding = 0 ) uniform accelerationStructureEXT as;

layout( binding = 1, set = 0) uniform accelerationStructureEXT as;
layout( binding = 3, set = 0) uniform rayParams {
    uint sbt_offset; // shader binding table offset
    uint sbt_stride; // shader binding table stride
    uint miss_index;
    uint out_image_index;
};

void main() {
    traceRayEXT( as,                    // topLevel
                 gl_RayFlagsOpaqueEXT, // rayFlags
                 0xff,                 // cullMask
                 sbt_offset,           // sbtRecordOffset
                 sbt_stride,           // sbtRecordStride
                 miss_index,           // missIndex
                 camera_position.xyz,  // origin
                 0.0,                  // Tmin
                 compute_ray_dir( gl_LaunchIDEXT, gl_LaunchSizeEXT ),
                 100.0,                // Tmax
                 0                     // payload index
                );

    if ( payload.geometry_id != -1 ) {
       ...
```

We will first see the ray generation shader, as it is the first shader to be invoked, generating work for the other shaders.
First of all we need to add the extension at the beginning of the shader.
Then we define our payload, a structure that will be used to communicate between the different shaders.
NOTE: when declared in the ray generation shader, use rayPayloadEXT as type, while in the other shaders will be rayPayloadInEXT! Don’t waste precious hours trying to understand why your raytracing shader does not work like me!

We then need to bind the acceleration structure (the TLAS) and a buffer containing at least the shader binding table offset, stride, and miss shader index, to explicitly call it.

Shader binding table offset and size can be queried by passing a VkPhysicalDeviceRayTracingPipelinePropertiesKHR structure to vkGetPhysicalDeviceProperties2.



## Ray Tracing: glsl shaders

- GPU Hardware support
- Spatial query based on Bounding Volume Hierarchy
- Different scene representation (BLAS and TLAS)
- New shader stages and pipeline
- **Shader example**

|  |
|---|
| Ray Generation |
| Closest Hit |
| Miss |

```
#extension GL_EXT_ray_tracing : enable

struct ray_payload {
    vec2 barycentric_weights;
    int geometry_id;
    int primitive_id;

    mat4x3 object_to_world;

    float t;
};

layout( location = 0 ) rayPayloadEXT ray_payload payload;

layout( binding = 1, set = 0) uniform accelerationStructureEXT as;
layout( binding = 3, set = 0) uniform rayParams {
    uint sbt_offset; // shader binding table offset
    uint sbt_stride; // shader binding table stride
    uint miss_index;
    uint out_image_index;
};

void main() {
    traceRayEXT( as,                    // topLevel
                 gl_RayFlagsOpaqueEXT, // rayFlags
                 0xff,                 // cullMask
                 sbt_offset,           // sbtRecordOffset
                 sbt_stride,           // sbtRecordStride
                 miss_index,           // missIndex
                 camera_position.xyz,  // origin
                 0.0,                  // Tmin
                 compute_ray_dir( gl_LaunchIDEXT, gl_LaunchSizeEXT ),
                 100.0,                // Tmax
                 0                     // payload index
                );

    if ( payload.geometry_id != -1 ) {
       ...
```

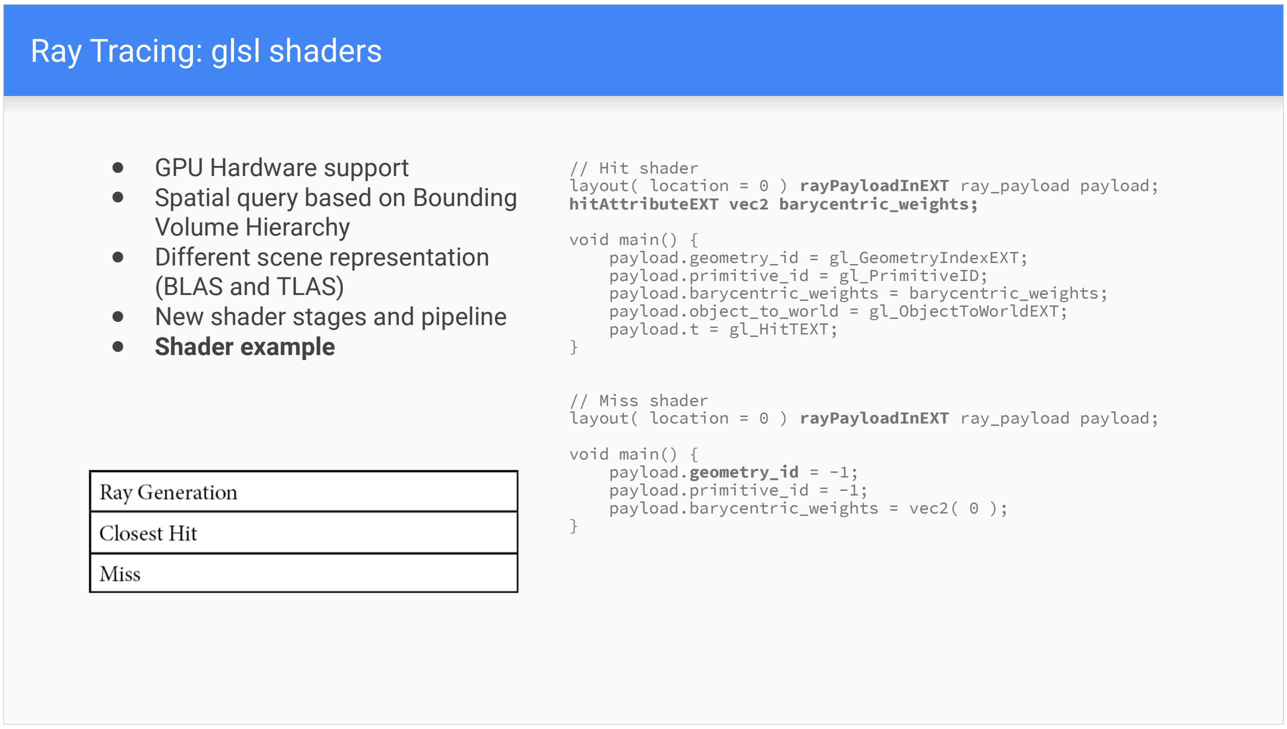
We are now ready to actually trace rays!
This happens by calling the function traceRayExt, with few parameters as outlined in the slide.
Aside from the shader binding table data and miss index, we need to specify the origin of the ray, the direction and maximum length.
The payload index is the location that we define in the shader.

Note also the compute_ray_dir, in which we calculate a direction given a pixel position.
As compute shaders, gl_LaunchIDEXTis the dispatch index, while gl_LaunchSizeEXT is the size. For this fullscreen shader, gl_LaunchIDEXT will be the pixel coordinate, while gl_LaunchSizeEXT will be the screen size.

We calculate the world space position with compute_ray_dir, adapted from https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-generating-camera-rays/generating-camera-rays
:

```
vec3 compute_ray_dir( uvec3 launchID, uvec3 launchSize) {
    float x = ( 2 * ( float( launchID.x ) + 0.5 ) / float( launchSize.x ) - 1.0 );
    float y = ( 1.0 - 2 * ( float( launchID.y ) + 0.5 ) / float( launchSize.y ) );
    vec4 dir = inverse_view_projection * vec4( x, y, 1, 1 );
    dir = normalize( dir );

    return dir.xyz;
}
```

After the function finishes, we can check if we actually hit a geometry.
Who writes the geometry_id value we are reading ?



## Ray Tracing: glsl shaders

- GPU Hardware support
- Spatial query based on Bounding Volume Hierarchy
- Different scene representation (BLAS and TLAS)
- New shader stages and pipeline
- **Shader example**

| |
|---|
| Ray Generation |
| Closest Hit |
| Miss |

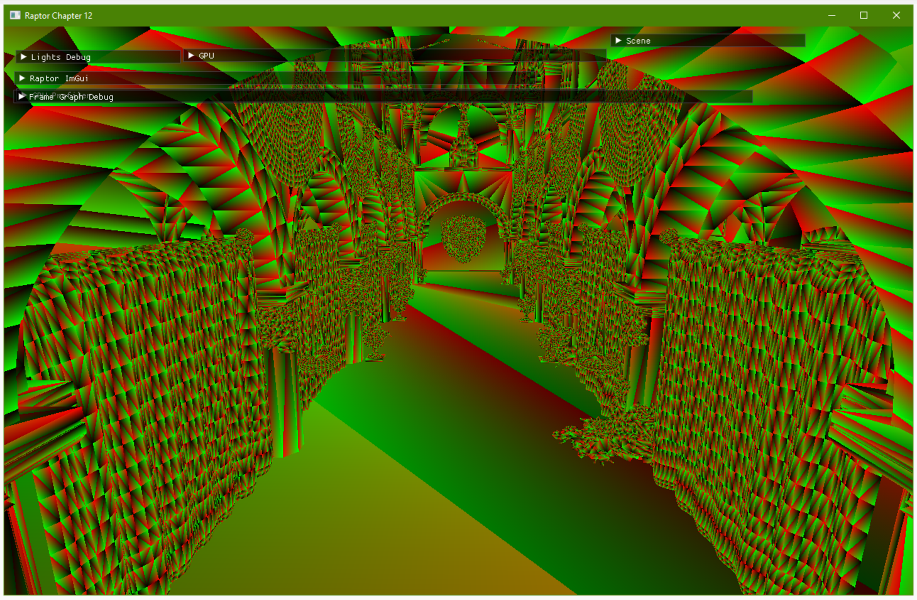
```
// Hit shader
layout( location = 0 ) rayPayloadInEXT ray_payload payload;
hitAttributeEXT vec2 barycentric_weights;

void main() {
    payload.geometry_id = gl_GeometryIndexEXT;
    payload.primitive_id = gl_PrimitiveID;
    payload.barycentric_weights = barycentric_weights;
    payload.object_to_world = gl_ObjectToWorldEXT;
    payload.t = gl_HitTEXT;
}


// Miss shader
layout( location = 0 ) rayPayloadInEXT ray_payload payload;

void main() {
    payload.geometry_id = -1;
    payload.primitive_id = -1;
    payload.barycentric_weights = vec2( 0 );
}
```

Hit and miss shader will do!
These are very basic shaders, but you can get a gist of what is happening.

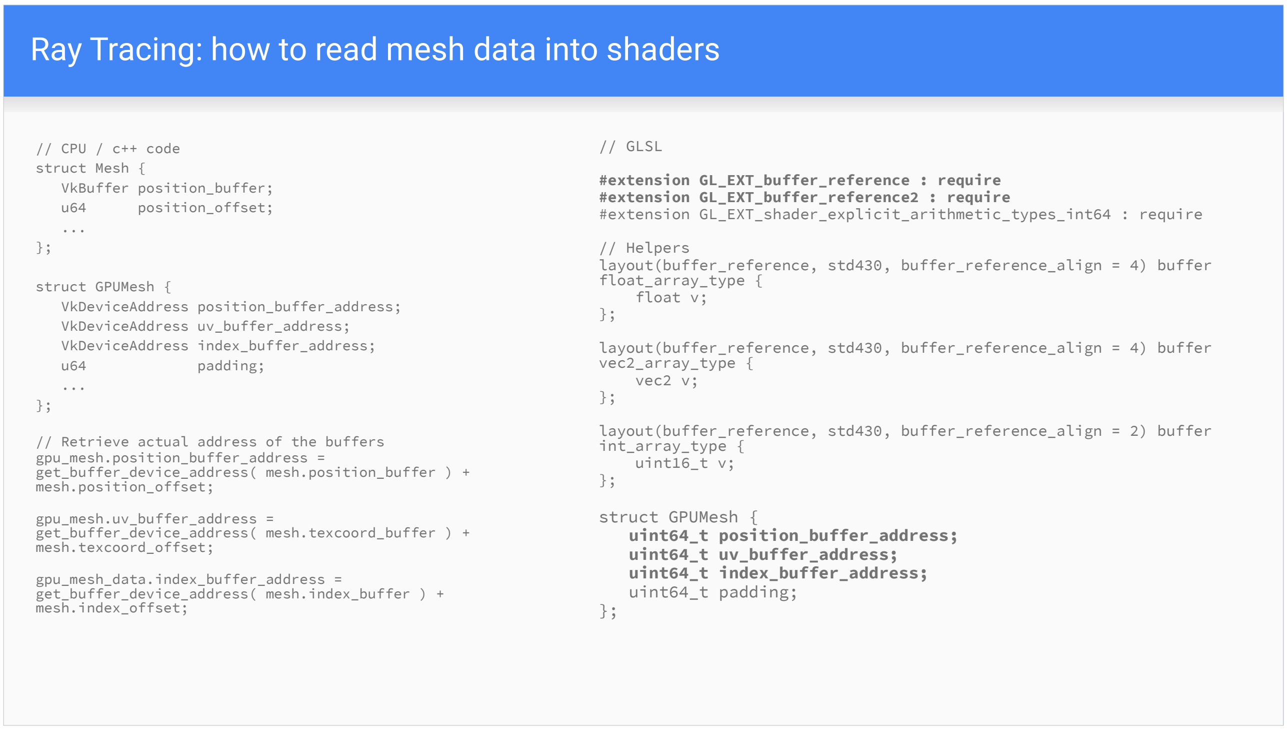
First we declare the payload with rayPayloadInEXT (notice the ‘in’), for the hit shader we also ask to read the barycentric weights to cache in the payload.
GLSL exposes many variables to retrieve useful informations: gl_GeometryIndexEXT is the mesh instance ID, gl_PrimitiveID is the triangle relative to the mesh index, gl_ObjectToWorldEXT is the transform, gl_HitTEXT is the length of the ray where the hit happened.

For the miss shader, we declare the same payload and just write -1 so we know that we did not hit anything.

## Ray Tracing: barycentric weights rendering



And this is the result of the shader we wrote before!
Not the fanciest, but it is a solid foundational block for the future.



## Ray Tracing: how to read mesh data into shaders

```
// CPU / c++ code
struct Mesh {
   VkBuffer position_buffer;
   u64      position_offset;
   ...
};

struct GPUMesh {
   VkDeviceAddress position_buffer_address;
   VkDeviceAddress uv_buffer_address;
   VkDeviceAddress index_buffer_address;
   u64             padding;
   ...
};

// Retrieve actual address of the buffers
gpu_mesh.position_buffer_address =
get_buffer_device_address( mesh.position_buffer ) +
mesh.position_offset;

gpu_mesh.uv_buffer_address =
get_buffer_device_address( mesh.texcoord_buffer ) +
mesh.texcoord_offset;

gpu_mesh_data.index_buffer_address =
get_buffer_device_address( mesh.index_buffer ) +
mesh.index_offset;
```

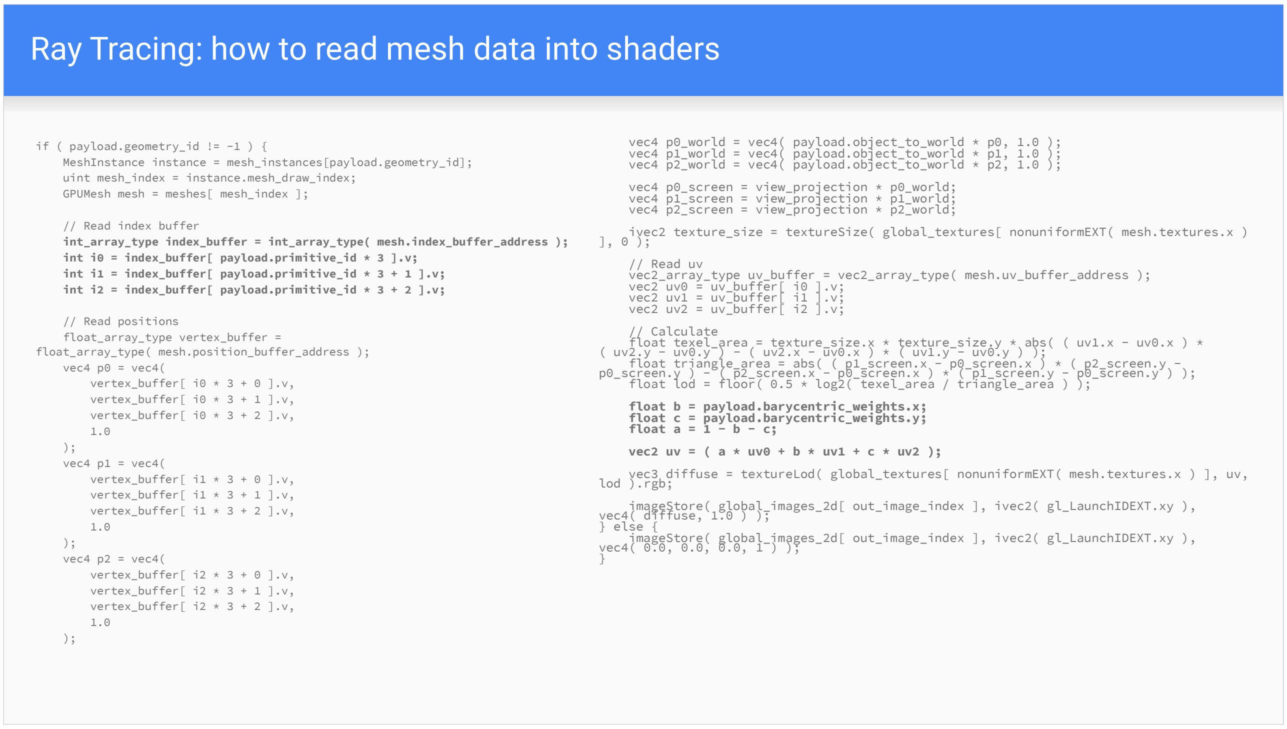
```
// GLSL

#extension GL_EXT_buffer_reference : require
#extension GL_EXT_buffer_reference2 : require
#extension GL_EXT_shader_explicit_arithmetic_types_int64 : require

// Helpers
layout(buffer_reference, std430, buffer_reference_align = 4) buffer
float_array_type {
    float v;
};

layout(buffer_reference, std430, buffer_reference_align = 4) buffer
vec2_array_type {
    vec2 v;
};

layout(buffer_reference, std430, buffer_reference_align = 2) buffer
int_array_type {
    uint16_t v;
};

struct GPUMesh {
   uint64_t position_buffer_address;
   uint64_t uv_buffer_address;
   uint64_t index_buffer_address;
   uint64_t padding;
};
```

The next logical step is to read for example the UV of the triangle we hit, so we can read a texture and output it right ?
This requires a little bit of preparation on the CPU side as well.

As a simple example, on the CPU we have our VkBuffers holding positions, uv, indices, that will be bound normally when calling a standard draw using vkBindVertexBuffers.
For each buffer, we retrieve the address (applying offsets if we need them).

GLSL side, we define the same structure GPUMesh using the uint64_t, and add some helpers definitions to cast the address into the shader!

Notice that in this case we elected to define our buffers using the device buffer address extension. This allows us to pass the address of the buffer (think of it like a pointer) to the shader, which simplifies some of the computation we are about to perform.
It also simplifies binding these buffers: all we need to do is pass in a “pointer” for each mesh. If we had to bind individual buffers for each mesh, we would have a lot more bookkeeping to do.



## Ray Tracing: how to read mesh data into shaders

```
if ( payload.geometry_id != -1 ) {
    MeshInstance instance = mesh_instances[payload.geometry_id];
    uint mesh_index = instance.mesh_draw_index;
    GPUMesh mesh = meshes[ mesh_index ];

    // Read index buffer
    int_array_type index_buffer = int_array_type( mesh.index_buffer_address );
    int i0 = index_buffer[ payload.primitive_id * 3 ].v;
    int i1 = index_buffer[ payload.primitive_id * 3 + 1 ].v;
    int i2 = index_buffer[ payload.primitive_id * 3 + 2 ].v;

    // Read positions
    float_array_type vertex_buffer =
float_array_type( mesh.position_buffer_address );
    vec4 p0 = vec4(
        vertex_buffer[ i0 * 3 + 0 ].v,
        vertex_buffer[ i0 * 3 + 1 ].v,
        vertex_buffer[ i0 * 3 + 2 ].v,
        1.0
    );
    vec4 p1 = vec4(
        vertex_buffer[ i1 * 3 + 0 ].v,
        vertex_buffer[ i1 * 3 + 1 ].v,
        vertex_buffer[ i1 * 3 + 2 ].v,
        1.0
    );
    vec4 p2 = vec4(
        vertex_buffer[ i2 * 3 + 0 ].v,
        vertex_buffer[ i2 * 3 + 1 ].v,
        vertex_buffer[ i2 * 3 + 2 ].v,
        1.0
    );

    vec4 p0_world = vec4( payload.object_to_world * p0, 1.0 );
    vec4 p1_world = vec4( payload.object_to_world * p1, 1.0 );
    vec4 p2_world = vec4( payload.object_to_world * p2, 1.0 );

    vec4 p0_screen = view_projection * p0_world;
    vec4 p1_screen = view_projection * p1_world;
    vec4 p2_screen = view_projection * p2_world;

    ivec2 texture_size = textureSize( global_textures[ nonuniformEXT( mesh.textures.x )
], 0 );

    // Read uv
    vec2_array_type uv_buffer = vec2_array_type( mesh.uv_buffer_address );
    vec2 uv0 = uv_buffer[ i0 ].v;
    vec2 uv1 = uv_buffer[ i1 ].v;
    vec2 uv2 = uv_buffer[ i2 ].v;

    // Calculate
    float texel_area = texture_size.x * texture_size.y * abs( ( uv1.x - uv0.x ) *
( uv2.y - uv0.y ) - ( uv2.x - uv0.x ) * ( uv1.y - uv0.y ) );
    float triangle_area = abs( ( p1_screen.x - p0_screen.x ) * ( p2_screen.y -
p0_screen.y ) - ( p2_screen.x - p0_screen.x ) * ( p1_screen.y - p0_screen.y ) );
    float lod = floor( 0.5 * log2( texel_area / triangle_area ) );

    float b = payload.barycentric_weights.x;
    float c = payload.barycentric_weights.y;
    float a = 1 - b - c;

    vec2 uv = ( a * uv0 + b * uv1 + c * uv2 );

    vec3 diffuse = textureLod( global_textures[ nonuniformEXT( mesh.textures.x ) ], uv,
lod ).rgb;

    imageStore( global_images_2d[ out_image_index ], ivec2( gl_LaunchIDEXT.xy ),
vec4( diffuse, 1.0 ) );
} else {
    imageStore( global_images_2d[ out_image_index ], ivec2( gl_LaunchIDEXT.xy ),
vec4( 0.0, 0.0, 0.0, 1 ) );
}
```

In this slide we have a look at actual code to render the albedo texture coming from the ray tracing shader from before!
We are in the generation shader, and instead of outputting just the barycentric weights, we calculate the actual UVs to read the albedo.
First we retrieve the actual mesh, as we need the buffer addresses we cached before.

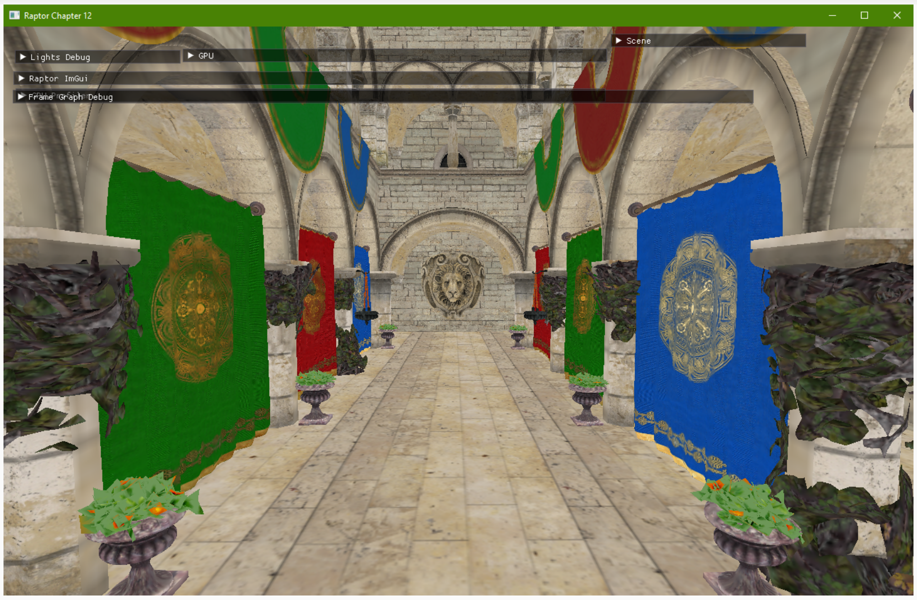
Then, using the helpers from the previous slide, we read the index buffer.
At this point we read the vertices position for the 3 vertices of the triangle: we need this to calculate the screen size for reading the correct LOD.

Following that we read the UVs, and read the diffuse texture using the UV coordinates and the texture LOD.

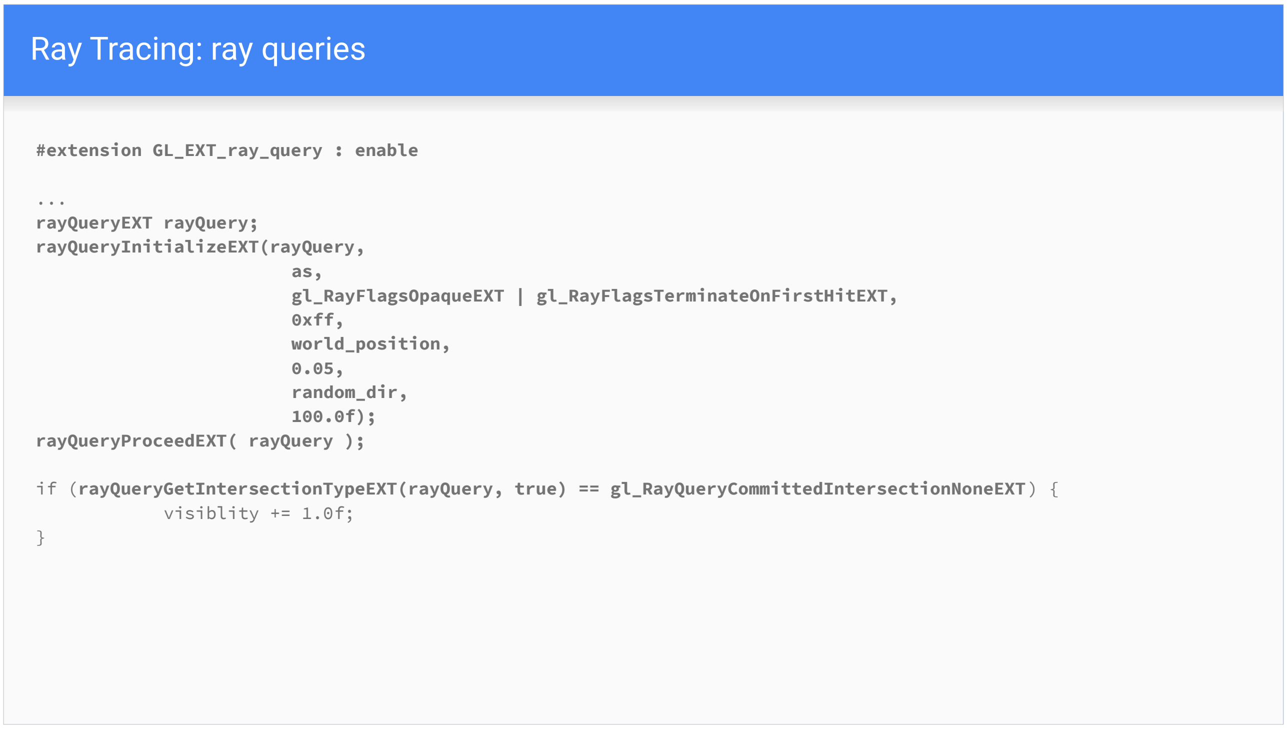
At this point we can output the albedo color!

When starting to work with ray-tracing, one starts to appreciate all the work that the GPU does for us when using the traditional rasterization pipeline: texture mip-map level selection, tri-linear filtering, derivatives, coordinate interpolation, etc. All of this has to be done manually when we move to ray-tracing.

## Ray Tracing: reading albedo texture color



Here’s an example of the previous shader in action. Notice there is some aliasing on the more distant meshes, our mip level selection likely needs improving



## Ray Tracing: ray queries

```
#extension GL_EXT_ray_query : enable

...
rayQueryEXT rayQuery;
rayQueryInitializeEXT(rayQuery,
                        as,
                        gl_RayFlagsOpaqueEXT | gl_RayFlagsTerminateOnFirstHitEXT,
                        0xff,
                        world_position,
                        0.05,
                        random_dir,
                        100.0f);
rayQueryProceedEXT( rayQuery );

if (rayQueryGetIntersectionTypeEXT(rayQuery, true) == gl_RayQueryCommittedIntersectionNoneEXT) {
            visiblity += 1.0f;
}
```

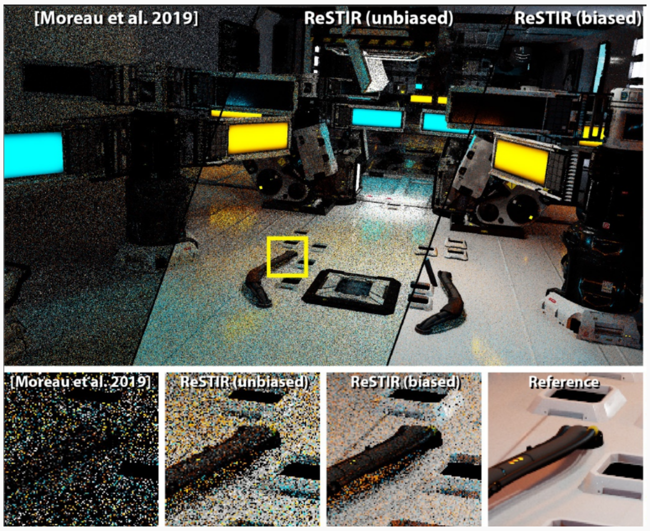
All the examples we have shown so far used a ray generation shader and the dispatch rays API call.
It’s also possible to cast rays into the scene from compute and fragment shaders.
Here we show how this is implemented.
We define a rayQueryEXT variable, that is then initialized using rayQueryInitializeEXT. We then call rayQueryProceedEXT to start the traversal.
No other shaders (hit, miss, closest hit) are involved in this case, so we have to manually check the result of the query.
You can then decide whether to continue the traversal or stop (i.e. to determine the closest hit).
This is an example of our ray-traced shadows implementation, so we only care about the first hit.

## Future of ray-tracing

- Shadows
- Reflections
- Lighting
- Path-tracing

- Denoising

- Audio
- AI
- Physics



Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting, Bitterli et al., Siggraph, 2020

This concludes our introduction to ray-tracing. Real-time ray-tracing is still evolving and new algorithms are still being researched.
Here we list a few passes that applications have started using for real-time ray-tracing.

I am sure you have heard of ReSTIR, which is a technique that allows ray-tracing very complex scenes at interactive frame rates.

All of the techniques above can only afford to shoot one (or less) ray per pixel. This leads to noisy results that cannot be used directly. For this reason, a number of denoising algorithms have been developed to maintain real-time rates.

We also list a few other areas might use ray casting on the CPU and that might be ported to the GPU to accelerate their implementations.

Of course, nothing prevents you to take advantage of this technology for offline rendering. Just be careful about how you manage your iterations, as shader execution on the GPU is limited to 2s*, after which the OS will terminate your program.

* This the limit on Windows, the mechanism is called TDR (timeout detection and recovery) which allows the OS to terminate the offending program without restarting the whole machine.

## References

- Vulkan spec: https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html
- SPIR-V spec: https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.html
- GLSL extensions: https://github.com/KhronosGroup/GLSL/tree/master/extensions
- Vulkan guide: https://github.com/KhronosGroup/Vulkan-Guide
- Mastering Graphics Programming with Vulkan, Marco Castorina and Gabriel Sassone, Packt Publishing, 2023
- GPU-Driven Rendering Pipelines, Ulrich Haar and Sebastian Aaltonen, Siggraph, 2015
- Optimizing the Graphics Pipeline With Compute, Graham Wihlidal, GDC, 2016
- Niagara, https://github.com/zeux/niagara, Arseny Kapoulkine
- Ray Tracing Gems, Eric Haines and Tomas Akenine-Möller, Apress, 2019
- Ray Tracing Gems II, Adam Marrs, Peter Shirley and Ingo Wald, Apress, 2021
- Radeon Ray Tracing Analyzer, https://gpuopen.com/radeon-raytracing-analyzer/

## Questions?