Vulkanised 2025

The 7th Vulkan Developer Conference
Cambridge, UK | February 11-13, 2025

Integrating Bindless Vulkan with Ray Tracing on Mobile Devices

Roman Kuznetsov, Meta
Sergey Kosarevsky, Meta

Authors

Roman Kuznetsov

Software Engineer @ Meta
Formerly: Mapbox, Organic Maps (ex MAPS.ME), Alawar
GitHub: @rokuz
X (Twitter): @rokuz7

Sergey Kosarevsky

Software Engineer @ Meta
Formerly: Ubisoft RedLynx Rendering Lead
GitHub: @corporateshark
X (Twitter): @CorporateShark

Prerequisites

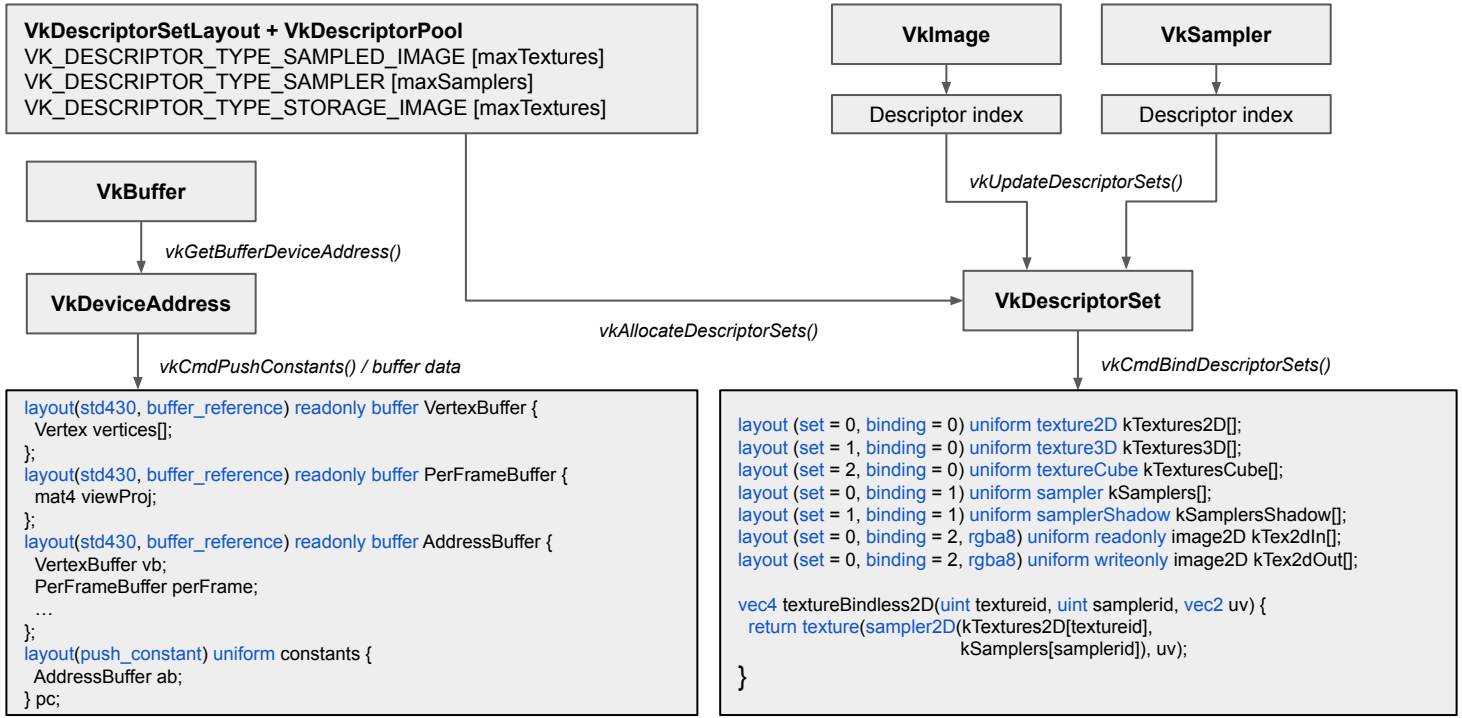
- [Vulkanised 2024] **Realistic Graphics with Ray Tracing on Mobile**
  Iago Calvo Lista, Arm, Graphics Software Engineer.
  https://youtu.be/jJyHzkWXEfY?si=DwH2_5o_sAZe9LQr

- [SIGGRAPH 2024] **Designing Mobile Rendering Engines with "Bindless" Vulkan**
  Sergey Kosarevsky, Alexey Medvedev
  https://doi.org/10.1145/3641233.3664326

Goals

- Demonstrate a practical approach to integrating Vulkan's ray-tracing with a "bindless" renderer
- Aim for GPUs designed for mobile devices
- Make it "lightweight" and open-source (https://github.com/corporateshark/lightweightvk)

Reminder: What's "bindless"?



Required extensions

- **VK_KHR_acceleration_structure** (7% Android devices)
- **VK_KHR_ray_query** (7% Android devices)
- **VK_KHR_ray_tracing_pipeline** (0.7% of Android devices)

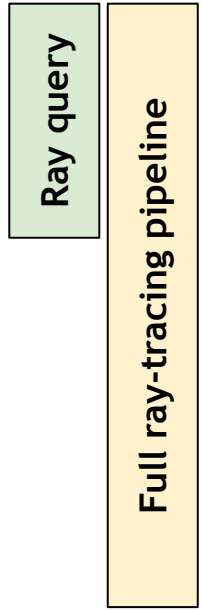
Sources:
Extensions - Vulkan Hardware Database by Sascha Willems
VK_KHR_acceleration_structure - Vulkan Hardware Database by Sascha Willems
VK_KHR_ray_query - Vulkan Hardware Database by Sascha Willems
VK_KHR_ray_tracing_pipeline - Vulkan Hardware Database by Sascha Willems

What did we add to LightweightVK?

- **Acceleration structures** (BLAS/TLAS) based on **"handle-based objects"** to manage their lifecycle and enable access (specifically TLAS) in shaders in a "bindless" way

- The **"Shader Binding Table"** is hidden from users; instead, we use the "ray-tracing pipeline" with shaders as the interface to ray-tracing capabilities

- **Pipeline binding, descriptor set updates, and synchronization barriers** are all hidden from users



Acceleration structures: BLAS

```
lvk::Holder<lvk::AccelStructHandle> lvk::VulkanContext::createAccelerationStructure(
    const AccelStructDesc& desc,
    Result* outResult);

// Example
res.BLAS = ctx_->createAccelerationStructure({
     .type = lvk::AccelStructType_BLAS,
     .geometryType = lvk::AccelStructGeomType_Triangles,
     .vertexFormat = lvk::VertexFormat::Float3,
     .vertexBuffer = res.vertexBuffer,                          // NOTE! Reference to a handle
     .numVertices = LVK_ARRAY_NUM_ELEMENTS(vertices),
     .indexFormat = lvk::IndexFormat_UI32,
     .indexBuffer = res.indexBuffer,                            // NOTE! Reference to a handle
     .transformBuffer = transformBuffer,                        // NOTE! Reference to a handle
     .buildRange = {.primitiveCount = LVK_ARRAY_NUM_ELEMENTS(indices) / 3},
 });
```

Acceleration structures: BLAS

```
res.vertexBuffer = ctx_->createBuffer({
    .usage = lvk::BufferUsageBits_AccelStructBuildInputReadOnly,
    .storage = lvk::StorageType_HostVisible,
    .size = sizeof(vertices),
    .data = vertices,
});
res.indexBuffer = ctx_->createBuffer({
    .usage = lvk::BufferUsageBits_AccelStructBuildInputReadOnly,
    .storage = lvk::StorageType_HostVisible,
    .size = sizeof(indices),
    .data = indices,
});
lvk::Holder<lvk::BufferHandle> transformBuffer = ctx_->createBuffer({
    .usage = lvk::BufferUsageBits_AccelStructBuildInputReadOnly,
    .storage = lvk::StorageType_HostVisible,
    .size = sizeof(glm::mat3x4),
    .data = &transformMatrix,
});
```

Acceleration structures: BLAS

```
lvk::AccelStructHandle lvk::VulkanContext::createBLAS(const AccelStructDesc& desc, Result* outResult) {
 // 1. Convert handles to GPU addresses
 // ...
 const VkAccelerationStructureGeometryKHR accelerationStructureGeometry{
   // ...
   .geometry = {
     .triangles = {
       // ...
       .vertexData = {.deviceAddress = gpuAddress(desc.vertexBuffer)},
       // ...
     },
   },
   // ...
 };

 // ...
```

Acceleration structures: BLAS

```
struct AccelerationStructure {
 bool isTLAS = false;
 VkAccelerationStructureBuildRangeInfoKHR buildRangeInfo = {};
 VkAccelerationStructureKHR vkHandle = VK_NULL_HANDLE;
 uint64_t deviceAddress = 0;
 lvk::Holder<lvk::BufferHandle> buffer;
};
```

```
// 2. Create acceleration structure
lvk::AccelerationStructure accelStruct = {
  // ...
  .buffer = createBuffer({
    .usage = lvk::BufferUsageBits_AccelStructStorage,
    .storage = lvk::StorageType_Device,
    .size = accelerationStructureBuildSizesInfo.accelerationStructureSize,
    .debugName = debugNameBuffer }),
};

const VkAccelerationStructureCreateInfoKHR ciAccelerationStructure = {
  // ...
  .buffer = getVkBuffer(this, accelStruct.buffer),
  // ...
};
VK_ASSERT(vkCreateAccelerationStructureKHR(vkDevice_, &ciAccelerationStructure,
                                           nullptr, &accelStruct.vkHandle));
```

Acceleration structures: BLAS

```
// 3. Create scratch buffer and build acceleration structure using command buffer
lvk::Holder<lvk::BufferHandle> scratchBuffer = createBuffer({
  .usage = lvk::BufferUsageBits_Storage,
  .storage = lvk::StorageType_Device,
  .size = accelerationStructureBuildSizesInfo.buildScratchSize,
  .overwrittenAlignment = props.minAccelerationStructureScratchOffsetAlignment,
  .debugName = "Buffer: BLAS scratch",
});
const VkAccelerationStructureBuildGeometryInfoKHR accelerationBuildGeometryInfo{
  // ...
  .scratchData = {.deviceAddress = gpuAddress(scratchBuffer)},
};

// ...
lvk::ICommandBuffer& buffer = acquireCommandBuffer();
vkCmdBuildAccelerationStructuresKHR(
    lvk::getVkCommandBuffer(buffer), 1, &accelerationBuildGeometryInfo, accelerationBuildStructureRangeInfos);
wait(submit(buffer, {}));
```

Acceleration structures: BLAS

```
Validation Error: VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03710

Object 0: handle = 0xb4000077ba6e54d0, type = VK_OBJECT_TYPE_COMMAND_BUFFER;
MessageID = 0x63ff8904

vkCmdBuildAccelerationStructuresKHR(): pInfos[0].scratchData.deviceAddress (12887524100)
must be a multiple of minAccelerationStructureScratchOffsetAlignment (256).
The Vulkan spec states: For each element of pInfos, its scratchData.deviceAddress member
must be a multiple of
VkPhysicalDeviceAccelerationStructurePropertiesKHR::minAccelerationStructureScratchOffset
Alignment
```

More details:
https://github.com/corporateshark/lightweightvk/pull/39

Acceleration structures: BLAS

```
 // 4. Extract physical device address and put acceleration structure to a pool accessing by handle

 // ...
 accelStruct.deviceAddress = vkGetAccelerationStructureDeviceAddressKHR(vkDevice_,
                                                                        &accelerationDeviceAddressInfo);

 return accelStructuresPool_.create(std::move(accelStruct));
}
```

Acceleration structures: BLAS build error

You might exceed limits.maxStorageBufferRange for acceleration structures or scratch buffer.

Mali-G715-Immortalis MC11 (v1.r38p1-01eac0.c1a71ccca2acf211eb87c5db5322f569) for Bistro mesh (2.8m primitives):
buildScratchSize = 7852992128 bytes (7.3GB)
accelerationStructureSize = 612837208 bytes (584.4MB)

maxStorageBufferSize = 268435456 bytes (256Mb)

It requires to subdivide BLAS at least into 30 parts to build on device!

Samsung Xclipse 940 for the same mesh:
buildScratchSize = 601488316 bytes (573.6Mb)
accelerationStructureSize = 461519536 bytes (440.1MB)

maxStorageBufferSize = -1 (No limits)

More details:
https://github.com/corporateshark/lightweightvk/pull/37

Acceleration structures: TLAS

```
 // Initialize TLAS in a similar way referencing to BLAS in instance buffer

res.instancesBuffer = ctx_->createBuffer(lvk::BufferDesc{
  .usage = lvk::BufferUsageBits_AccelStructBuildInputReadOnly,
  .storage = lvk::StorageType_HostVisible,
  .size = sizeof(lvk::AccelStructInstance),
  .data = &instance,
  .debugName = "instanceBuffer",
});

res.TLAS = ctx_->createAccelerationStructure({
  .type = lvk::AccelStructType_TLAS,
  .geometryType = lvk::AccelStructGeomType_Instances,
  .instancesBuffer = res.instancesBuffer,
  .buildRange = {.primitiveCount = 1},
  .buildFlags = lvk::AccelStructBuildFlagBits_PreferFastTrace |
                lvk::AccelStructBuildFlagBits_AllowUpdate,
});
```

Acceleration structures: TLAS

```
// Initialize TLAS in a similar way referencing to BLAS in instance buffer

res.instancesBuffer = ctx_->createBuffer(lvk::BufferDesc{
  .usage = lvk::BufferUsageBits_AccelStructBuildInputReadOnly,
  .storage = lvk::StorageType_HostVisible,
  .size = sizeof(lvk::AccelStructInstance),
  .data = &instance,
  .debugName = "instanceBuffer",
});

res.TLAS = ctx_->createAccelerationStructure({
  .type = lvk::AccelStructType_TLAS,
  .geometryType = lvk::AccelStructGeomType_Instances,
  .instancesBuffer = res.instancesBuffer,
  .buildRange = {.primitiveCount = 1},
  .buildFlags = lvk::AccelStructBuildFlagBits_PreferFastTrace |
                lvk::AccelStructBuildFlagBits_AllowUpdate,
});
```

Acceleration structures: TLAS in ray-gen shader

```
#extension GL_EXT_ray_tracing : require
#extension GL_EXT_buffer_reference : require

// ...
layout (set = 0, binding = 4) uniform accelerationStructureEXT kTLAS[];
// ...

layout(push_constant) uniform constants {
  // ...
  uint tlas;
};

// ...
void main() {
  // ...
  traceRayEXT(kTLAS[tlas], gl_RayFlagsOpaqueEXT, 0xff, 0, 0, 0, origin.xyz, tmin, direction.xyz, tmax, 0);
  // ...
}
```

Acceleration structures: TLAS in fragment shader for ray query

```
#extension GL_EXT_ray_query : require
// ...
layout(set = 0, binding = 4) uniform accelerationStructureEXT kTLAS[];
layout(push_constant) uniform constants {
  // ...
  uint tlas;
} pc;

float traceAO(rayQueryEXT rq, vec3 origin, vec3 dir) {
  uint flags = pc.aoDistanceBased ? gl_RayFlagsTerminateOnFirstHitEXT : gl_RayFlagsNoneEXT;
  rayQueryInitializeEXT(rq, kTLAS[pc.tlas], flags, 0xFF, origin, 0.0f, dir, pc.aoRadius);
  while (rayQueryProceedEXT(rq)) {}

  if (rayQueryGetIntersectionTypeEXT(rq, true) != gl_RayQueryCommittedIntersectionNoneEXT) {
    if (pc.aoDistanceBased) return 1;
    float length = 1.0 - (rayQueryGetIntersectionTEXT(rq, true) / pc.aoRadius);
    return length;
  }
  return 0;
}
```

Acceleration structures: TLAS in updating descriptor sets

```
std::vector<VkAccelerationStructureKHR> handlesAccelStructs;
handlesAccelStructs.reserve(accelStructuresPool_.objects_.size());

VkAccelerationStructureKHR dummyTLAS = VK_NULL_HANDLE;
// use the first valid TLAS as a dummy
for (const auto& as : accelStructuresPool_.objects_) {
  if (as.obj_.vkHandle && as.obj_.isTLAS) {
    dummyTLAS = as.obj_.vkHandle;
  }
}
for (const auto& as : accelStructuresPool_.objects_) {
  handlesAccelStructs.push_back(as.obj_.isTLAS ? as.obj_.vkHandle : dummyTLAS);
}

VkWriteDescriptorSetAccelerationStructureKHR writeAccelStruct = {
    .sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_ACCELERATION_STRUCTURE_KHR,
    .accelerationStructureCount = (uint32_t)handlesAccelStructs.size(),
    .pAccelerationStructures = handlesAccelStructs.data(),
};
```

Acceleration structures: set and update TLAS

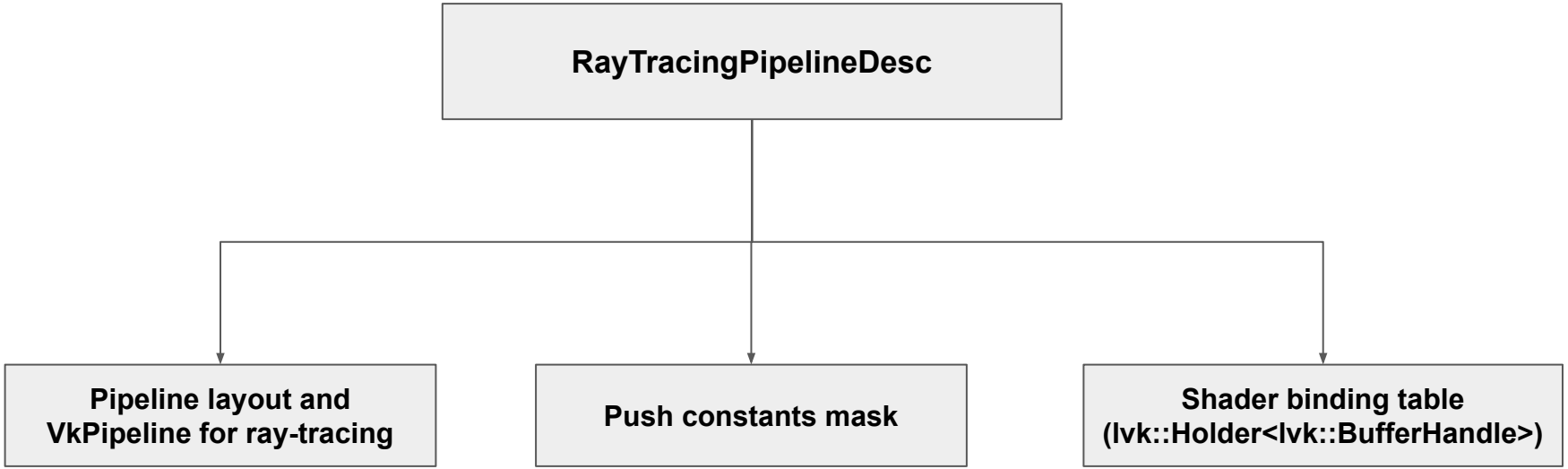
```
struct {
   uint64_t camBuffer;
   uint32_t outTexture;
   uint32_t tlas;
   float time;
 } pc = {
   // ...
   .tlas = res.TLAS.index(),
 };

 buffer.cmdUpdateTLAS(res.TLAS, res.instancesBuffer);
 buffer.cmdBindRayTracingPipeline(res.pipeline);
 buffer.cmdPushConstants(pc);
 buffer.cmdTraceRays((uint32_t)width_, (uint32_t)height_, 1,
                     {.textures = {lvk::TextureHandle(res.storageImage)}});
 buffer.cmdCopyImage(res.storageImage, ctx_->getCurrentSwapchainTexture(),
                     ctx_->getDimensions(ctx_->getCurrentSwapchainTexture()));
```

Ray-tracing pipeline

```
// Ray-tracing pipeline descriptor contains LightweightVK handles of shader modules
struct RayTracingPipelineDesc final {
  ShaderModuleHandle smRayGen[LVK_MAX_RAY_TRACING_SHADER_GROUP_SIZE] = {};
  ShaderModuleHandle smAnyHit[LVK_MAX_RAY_TRACING_SHADER_GROUP_SIZE] = {};
  ShaderModuleHandle smClosestHit[LVK_MAX_RAY_TRACING_SHADER_GROUP_SIZE] = {};
  ShaderModuleHandle smMiss[LVK_MAX_RAY_TRACING_SHADER_GROUP_SIZE] = {};
  ShaderModuleHandle smIntersection[LVK_MAX_RAY_TRACING_SHADER_GROUP_SIZE] = {};
  ShaderModuleHandle smCallable[LVK_MAX_RAY_TRACING_SHADER_GROUP_SIZE] = {};
  SpecializationConstantDesc specInfo = {};
  const char* entryPoint = "main";
  const char* debugName = "";
};
```

Ray-tracing pipeline



Shader binding table

Shader groups and indexes defined by the order of shader modules in lvk::RayTracingPipelineDesc

```
rayTracingPipeline_ = ctx_->createRayTracingPipeline(lvk::RayTracingPipelineDesc{
    .smRayGen = {smRaygen_},
    .smClosestHit = {smHit_},
    .smMiss = {smMiss_, smMissShadow_},
});
```

In this example,
Group 0: [smRaygen_]
Group 1: [smMiss_]
Group 2: [smMissShadow_]
Group 3: [smHit_]

Note: corresponding any-hit, closest-hit and intersection shader will be put to the same group

Shader binding table

To access specific miss shader by index in `lvk::RayTracingPipelineDesc::smMiss_` array, just use `missIndex` parameter in `traceRayEXT(...)`

```
isShadowed = true;
vec3 hitPoint = gl_WorldRayOriginEXT + gl_WorldRayDirectionEXT * gl_HitTEXT;
traceRayEXT(kTLAS[tlas],
            gl_RayFlagsTerminateOnFirstHitEXT | gl_RayFlagsOpaqueEXT | gl_RayFlagsSkipClosestHitShaderEXT,
            0xff, 0, 0, 1, hitPoint, tmin, lightDir.xyz, tmax, 1);
```

The access to the specific hit/intersections shaders may be a bit trickier depending on the your setup.

Formula (see https://docs.vulkan.org/spec/latest/chapters/raytracing.html for details):

```
pHitShaderBindingTable->deviceAddress + pHitShaderBindingTable->stride ×
(instanceShaderBindingTableRecordOffset + geometryIndex × sbtRecordStride + sbtRecordOffset)
```

In simple cases (if `instanceShaderBindingTableRecordOffset == 0` and `sbtRecordStride == 0`), `sbtRecordOffset` can be used the same way as `missIndex`.

Ray-tracing pipeline state

```
struct RayTracingPipelineState final {
  RayTracingPipelineDesc desc_;

  VkDescriptorSetLayout lastVkDescriptorSetLayout_ = VK_NULL_HANDLE;

  VkShaderStageFlags shaderStageFlags_ = 0;
  VkPipelineLayout pipelineLayout_ = VK_NULL_HANDLE;
  VkPipeline pipeline_ = VK_NULL_HANDLE;

  void* specConstantDataStorage_ = nullptr;

  lvk::Holder<lvk::BufferHandle> sbt;

  VkStridedDeviceAddressRegionKHR sbtEntryRayGen = {};
  VkStridedDeviceAddressRegionKHR sbtEntryMiss = {};
  VkStridedDeviceAddressRegionKHR sbtEntryHit = {};
  VkStridedDeviceAddressRegionKHR sbtEntryCallable = {};
};
```

Bind ray-tracing pipeline state

```
void lvk::CommandBuffer::cmdBindRayTracingPipeline(lvk::RayTracingPipelineHandle handle) {
  // ...
  const lvk::RayTracingPipelineState* rtps = ctx_->rayTracingPipelinesPool_.get(handle);
  // ...
  if (lastPipelineBound_ != pipeline) {
    lastPipelineBound_ = pipeline;
    vkCmdBindPipeline(wrapper_->cmdBuf_, VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR, pipeline);
    ctx_->checkAndUpdateDescriptorSets();
    ctx_->bindDefaultDescriptorSets(wrapper_->cmdBuf_,
                                    VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR, rtps->pipelineLayout_);
  }
}
```

Bind ray-tracing pipeline state

```
void lvk::CommandBuffer::cmdTraceRays(uint32_t width, uint32_t height, uint32_t depth,
                                      const Dependencies& deps) {
  lvk::RayTracingPipelineState* rtps =
    ctx_->rayTracingPipelinesPool_.get(currentPipelineRayTracing_);
  // ...
  vkCmdTraceRaysKHR(
    wrapper_->cmdBuf_, &rtps->sbtEntryRayGen, &rtps->sbtEntryMiss, &rtps->sbtEntryHit,
    &rtps->sbtEntryCallable, width, height, depth);
}
```

Synchronization: barriers for buffers and transition layouts for images

```
void lvk::CommandBuffer::cmdTraceRays(uint32_t width, uint32_t height, uint32_t depth,
                                      const Dependencies& deps) {
  // ...
  for (uint32_t i = 0; i != Dependencies::LVK_MAX_SUBMIT_DEPENDENCIES && deps.textures[i]; i++) {
    useComputeTexture(deps.textures[i], VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_KHR);
  }
  for (uint32_t i = 0; i != Dependencies::LVK_MAX_SUBMIT_DEPENDENCIES && deps.buffers[i]; i++) {
    bufferBarrier(deps.buffers[i],
                  VK_PIPELINE_STAGE_VERTEX_INPUT_BIT | VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,
                  VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_KHR);
  }

  vkCmdTraceRaysKHR(
    wrapper_->cmdBuf_, &rtps->sbtEntryRayGen, &rtps->sbtEntryMiss, &rtps->sbtEntryHit,
    &rtps->sbtEntryCallable, width, height, depth);
}
```

Synchronization: barriers in updating TLAS

```
void lvk::CommandBuffer::cmdUpdateTLAS(AccelStructHandle handle, BufferHandle instancesBuffer) {
  // ...
  const VkBufferMemoryBarrier2 barriers[] = {
      {
          .sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER_2,
          .srcStageMask = VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_KHR,
          .srcAccessMask = VK_ACCESS_MEMORY_READ_BIT,
          .dstStageMask = VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,
          .dstAccessMask = VK_ACCESS_MEMORY_READ_BIT | VK_ACCESS_MEMORY_WRITE_BIT,
          .buffer = getVkBuffer(ctx_, handle),
          .size = VK_WHOLE_SIZE,
      },
      {
          .sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER_2,
          .srcStageMask = VK_PIPELINE_STAGE_TRANSFER_BIT | VK_PIPELINE_STAGE_HOST_BIT,
          .srcAccessMask = VK_ACCESS_MEMORY_WRITE_BIT,
          .dstStageMask = VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,
          .dstAccessMask = VK_ACCESS_MEMORY_READ_BIT,
          .buffer = getVkBuffer(ctx_, instancesBuffer),
          .size = VK_WHOLE_SIZE,
      },
  };
```

Synchronization: barriers in updating TLAS

```
  const VkDependencyInfo dependencyInfo{
      .sType = VK_STRUCTURE_TYPE_DEPENDENCY_INFO,
      .bufferMemoryBarrierCount = LVK_ARRAY_NUM_ELEMENTS(barriers),
      .pBufferMemoryBarriers = barriers,
  };
  vkCmdPipelineBarrier2(wrapper_->cmdBuf_, &dependencyInfo);

  vkCmdBuildAccelerationStructuresKHR(wrapper_->cmdBuf_, 1, &accelerationBuildGeometryInfo,
                                      accelerationBuildStructureRangeInfos);
  const VkBufferMemoryBarrier2 barrier = {
      .sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER_2,
      .srcStageMask = VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,
      .srcAccessMask = VK_ACCESS_MEMORY_READ_BIT | VK_ACCESS_MEMORY_WRITE_BIT,
      .dstStageMask = VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_KHR,
      .dstAccessMask = VK_ACCESS_MEMORY_READ_BIT,
      .buffer = getVkBuffer(ctx_, handle),
      .offset = 0, .size = VK_WHOLE_SIZE,
  };
  const VkDependencyInfo dependencyInfo{.sType = VK_STRUCTURE_TYPE_DEPENDENCY_INFO,
                                        .bufferMemoryBarrierCount = 1, .pBufferMemoryBarriers = &barrier};
  vkCmdPipelineBarrier2(wrapper_->cmdBuf_, &dependencyInfo);
}
```

Synchronization: transition layouts during copying image to a swapchain

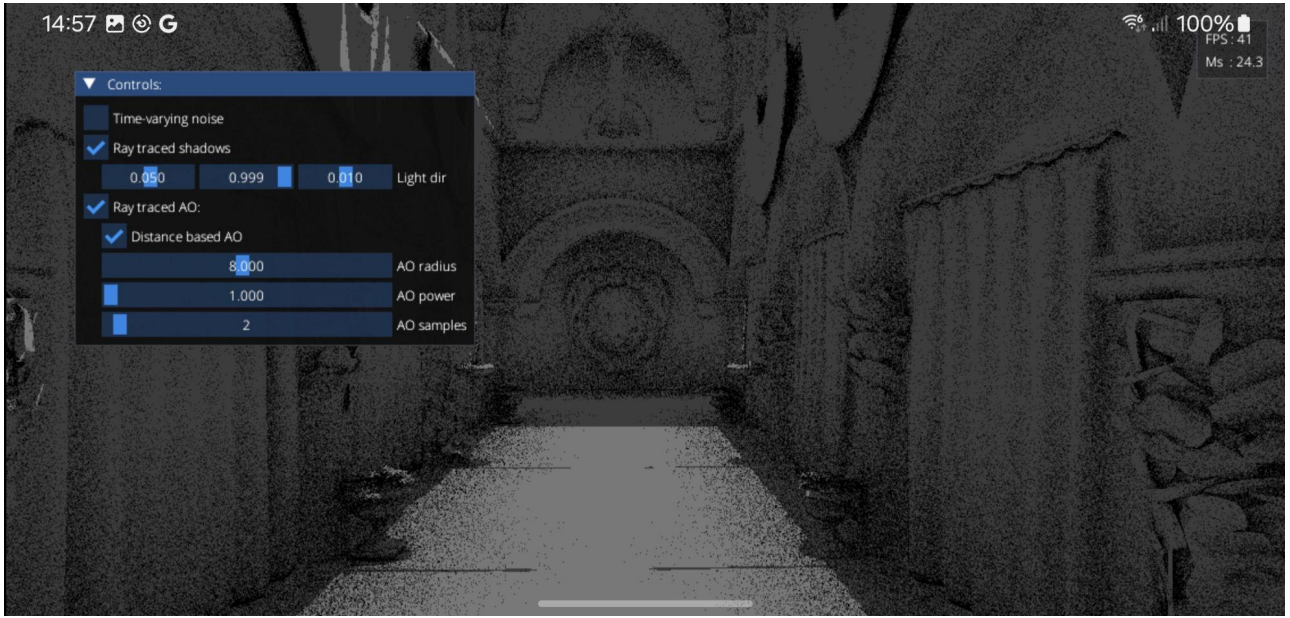
```
buffer.cmdTraceRays((uint32_t)width_, (uint32_t)height_, 1,
                    {.textures = {lvk::TextureHandle(res.storageImage)}});

buffer.cmdCopyImage(res.storageImage, ctx_->getCurrentSwapchainTexture(),
                    ctx_->getDimensions(ctx_->getCurrentSwapchainTexture()));
```

See implementation of **cmdCopyImage()** here:

https://github.com/corporateshark/lightweightvk/blob/c297911d8daefb26d4e0a9826a667c6ea9807a46/lvk/vulkan/VulkanClasses.cpp#L2667

Demo: Ambient occlusion + shadows using ray queries



Resolution: ~720P, scene: 260K triangles, no MSAA

Samsung S24 (SM-S921B, GPU: Samsung Xclipse 940)
Frame time: 24.3ms

Xiaomi 13T (GPU: Mali-G715-Immortalis MC11)
Frame time: 493ms

Demo: Ambient occlusion + shadows using ray queries

Resolution: ~720P, scene: 2.8M triangles, no MSAA

Samsung S24 (SM-S921B, GPU: Samsung Xclipse 940)
Frame time: 220ms

Xiaomi 13T (GPU: Mali-G715-Immortalis MC11)
Frame time: 550ms

Demo: Full ray-tracing pipeline

Resolution: ~720P, scene: 260K triangles
Samsung S24 (SM-S921B, GPU: Samsung Xclipse 940)
Frame time: 12.1ms

Demo: Full ray-tracing pipeline

Resolution: ~720P, scene: 2.8M triangles
Samsung S24 (SM-S921B, GPU: Samsung Xclipse 940)
Frame time: 14.9ms

Outcomes

- Ray-tracing can be done “lightweight”, please refer to LVK repository (https://github.com/corporateshark/lightweightvk)

- **VK_KHR_ray_query** is more common on the mass-market devices (7%) in comparison with **VK_KHR_ray_tracing_pipeline** (0.7%).

- Some mobile GPUs demonstrate outstanding performance doing hardware accelerated realtime ray-tracing, some are not yet

Q & A