Project 3: Ray Casting

(as a step toward path tracing)

Ray casting is the first step of a full **path tracing** algorithm. Each pixel generates a single ray, and invokes the ray tracing features to calculate its intersection with the scene. In a later step, path tracing will extend further rays from that intersection point, "hoping to hit a light", but for now we don't. For each ray, just use the same lighting calculation as the scanline algorithm. This should produce the exact same picture as the scanline algorithm, but with the scanline technology replaced with ray tracing technology.

- New **VkApp** setup procedures to implement for all the ray tracing steps (only 3 denoising steps will remain commented out):

  createRtBuffers();
  initRayTracing();
  createRtAccelerationStructure();
  createRtDescriptorSet();
  createRtPipeline();
  createRtShaderBindingTable();

- The drawFrame procedure can now call **raytrace()**, though not **denoise()** yet. You should provide some GUI checkbox to change the **useRaytracer** boolean at will.

```
if (useRaytracer) {
    raytrace();
    // denoise();
 }  else {
    rasterize(); }
```

- The push constant structure, **PushConstantRay**, for the ray tracing pipeline is defined in **shaders/shared_struct.h**. Create three temporary light related values, and fill them in (see VkApp::raytrace) with the indicated values.

```
ALIGNAS(16) vec4 tempLightPos;  // TEMPORARY: vec4(0.5f, 2.5f, 3.0f, 0.0);
ALIGNAS(16) vec4 tempLightInt;   // TEMPORARY: vec4(2.5, 2.5, 2.5, 0.0);
ALIGNAS(16) vec4 tempAmbient;  // TEMPORARY: vec4(0.2);
```

  Why temporary? Because, while the scene does have light emitting surfaces, this (ray casting) algorithm has no way to find them. So, for now we provide this one light placed high in the middle of the scene and some ambient light.

  Beware: Adding anything to **PushConstantRay** can result in ALIGNMENT-HELL –- a mismatch between two compilers (C++ and GLSL) alignment conventions. Here are three solutions:

  - Use the ALIGNAS macro defined in **shared_struct.h;** Use ALIGNAS(4) for bool, int, unit, float, vec2(?), and ALIGNAS(16) for vec3, vec4 and mat4(?).

  - Set and test for a sentinel value as outlined in the comments in **shared_struct.h.**

  - Watch for this validation layer message:
    VUID-VkRayTracingPipelineCreateInfoKHR-layout-03427(ERROR / SPEC): msgNum: 1749323802 - Validation Error: [VUID-VkRayTracingPipelineCreateInfoKHR-layout-03427] Object 0: handle = 0xd600000000d6, type = VK_OBJECT_TYPE_SHADER_MODULE; Object 1: handle = 0xda00000000da, type = VK_OBJECT_TYPE_PIPELINE_LAYOUT; | MessageID = 0x6844901a | Push constant buffer: member:0 member:13 offset:4 in VK_SHADER_STAGE_RAYGEN_BIT_KHR is out of range in VkPipelineLayout 0xda00000000da[]. The Vulkan spec states: layout must be consistent with all shaders specified in pStages (https://www.khronos.org/registry/vulkan/specs/1.3-extensions/html/vkspec.html#VUID-VkRayTracingPipelineCreateInfoKHR-layout-03427)

Finally, some actual ray tracing! Three new shaders and their shared payload structure must be defined. In each of the shader files mentioned, find and follow the comments labeled with **// @@ Raycasting: …**

- In **shared_struct.h,** verify that the ray-payload structure, **struct RayPayload,** as defined, contains at least the following items**:**

```
struct RayPayload
{
    bool hit;               // Does the ray intersect anything or not?
    vec3 hitPos;            // The world coordinates of the hit point.
    int instanceIndex;  // Index of the object instance hit (we have only one, so =0)
    int primitiveIndex; // Index of the hit triangle primitive within object
    vec3 bc;                // Barycentric coordinates of the hit point within triangle
};
```

- In the ray generation shader **raytrace.rgen,** follow all "@@ RayCasting" instructions. (Ignore @@Pathtracing and @@History and @@Explicit instructions until later projects.) The @@RayCasting instructions will direct you to do:
  - Write the BRDF lighting calculation EvalBRDF(…)
  - Verify there are no CPU/GPU structure alignment problems.
  - Cast a ray, check for hit/miss, and and calculate a output color as one of:
    - Return (0,0,0) if nothing was hit
    - Call EvalBRDF() if a hit is detected on a reflective triangle
    - Lookup and return a light's color if a hit is detected on a light triangle
  - Then write the resulting pixel color to the output image.

- In the miss shader, **raytrace.rmiss**, follow the @@ instruction to fill in the ray payload's **hit** boolean with **false**:

```
payload.hit = false;
```

- In the hit shader, **raytrace.rchit, f**ollow the @@ instruction fill in the hit boolean with **true**, and fill in the remaining 4 values with information about the hit triangle as provided by Vulkan:

```
payload.hit = true;

payload.instanceIndex = gl_InstanceCustomIndexEXT;
payload.primitiveIndex = gl_PrimitiveID;
payload.bc = vec3(1.0-bc.x-bc.y,  bc.x,  bc.y);
payload.hitPos = gl_WorldRayOriginEXT + gl_WorldRayDirectionEXT * gl_HitTEXT;
```