RTX Real Time Ray Tracing Project Summary

Nathan DeStafeno

destafen@oregonstate.edu

CS406

Intro

The goal of this project was to learn about NVIDIA's Real-Time Ray Tracing implementation in Vulkan. The initial goal of the project was to get the Real-Time Ray Tracing pipeline set up and working in Vulkan with extra goals of adding shadows, anti-aliasing, transparency, reflections, and other graphical features not native to ray tracing. This work is based on the Vk_Raytracing_tutorial released by NVIDIA.

How to build and run

Building NVIDIA's Vulkan Ray Tracing Repo involves downloading three repositories, CMAKE, and the Vulkan SDK. It is recommended to have Vulkan SDK version 1.2.131.2 or greater. In the root directory of the project needs to exist three folders, the shared_external repo, the shared_sources repo, and the vk_raytracing_tutorial repo. Shared_sources includes the primary framework that the code depends on. Shared_external includes third party libraries that come pre-compiled for use.

After the Vulkan SDK downloaded and the three repo folders are set in the project folder, set both CMAKE's source code and build location to be the vk_raytracing_tutorial repo folder. Click configure and then generate for the installed version of Visual Studio. Ensure there are no errors.

After compiling in one of the Visual Studio projects, there may be an error running the built executable. The work around is to open file explorer to the project's root folder and navigate to bin_x64/Debug/<Project>.exe and running it manually. Unfortunately this has to be done to run the program every time it compiles.

My work was done Vk_RayTracing\vk_raytracing_tutorial\ray_tracing__simple\ray_tracing__simple.sln. This is what the intro to ray-tracing tutorial will create and is what is built off for the remaining of the projects.

Rasterization to Ray Tracing

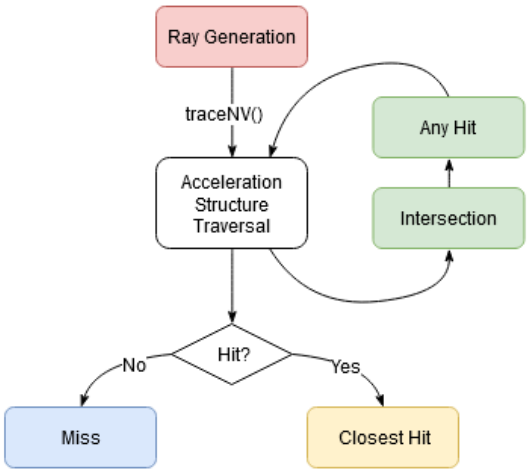
The initial set up for the project is in ray_tracing__before. This includes a very simple Vulkan Rasterization program that render a multi-colored cube in a white void. The two main source files are main.cpp and hello_vulkan.cpp. main.cpp is the program's main and hello_vulkan.cpp is all the functions main.cpp calls. There is an associated hello_vulkan.h for it's header file.

The first set to add ray tracing to this project is to load the VK_NV_RAY_TRACING_EXTENSION as well as the creation of an initRayTracing() function which queries the ray tracing capabilities of the GPU.

Next is the setup of the acceleration structures, which is used to reduce the number of intersection tests during rendering. It's divided into a two-level tree, a top level and bottom level. The Bottom-Level Acceleration Structure (BLAS) contains the geometry of an instance, containing the actual vertex data. The Top-Level Acceleration Structure (TLAS) contains object instances, with their own transformation matrix. We use a several helper functions provided by NVIDIA to make this easier. BuildBlas() which generates one BLAS per geometry and buildTlas() which builds the Top-Level AS from a vector of instanced objects.

Next is the Ray Tracing Descriptor Set. Vulkan Ray Tracing requires a single set of descriptor sets that contain all resources to render the scene as ray tracing doesn't know what resources it will need ahead of time. We first used createRtDescriptorSet() to create the descriptor set with scene information, references to the TLAS, and buffer for the output image. This is followed by the updateRtDescriptorSet() which is used to update the descriptor set should the image change, such as resizing the window.

Up next, we create the Ray Tracing Pipeline. As pictured below, when a ray is shot through the scene using the ray generation shader (rgen). If it doesn't hit an object, it calls the miss shader(rmiss). If it does hit, it will call the closest hit shader(crhit).

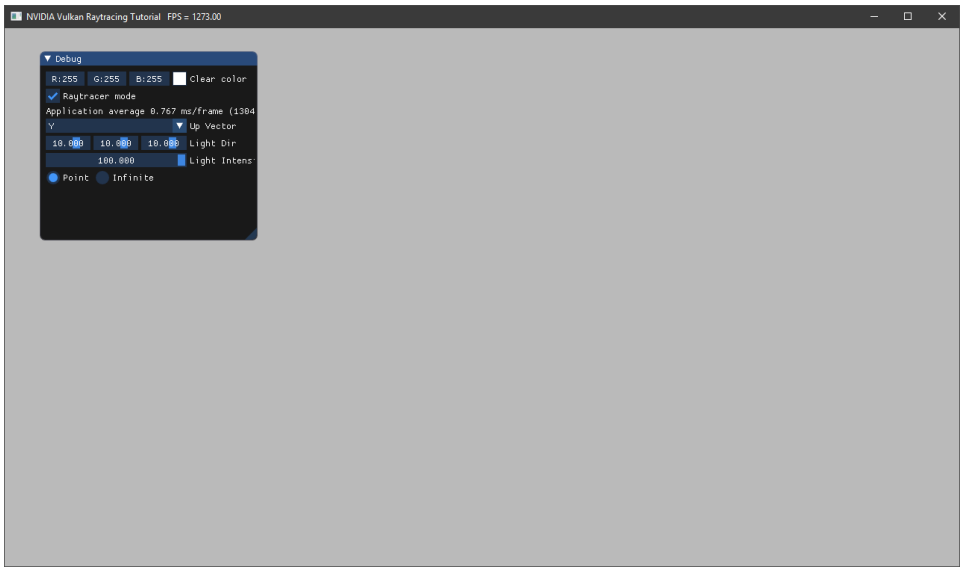


The first action of the createRtPipeline() function makes sure the associated SPIRV files can be loaded, followed up adding each shader to the appropriate shader group. It also set's the Push Constant's range for the ray tracing payload to allow the shaders to access global uniform variables. After the shaders are set up, we then setup the pipeline layout to describe how the pipeline will access external data.

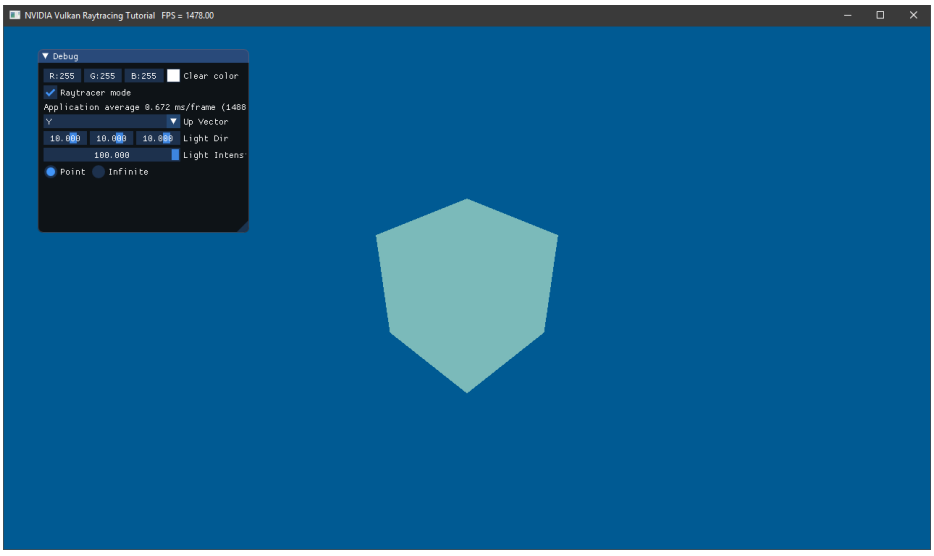
Then we have the shader binding table. We use this in Ray Tracing to indicate which ray generation shader to start with, which shader miss shader to run if there is a miss, and which hit shader to run if we have a hit. Shader groups and instances are associated in the Top-Level AS by assigning a 'hitgroupId'. We create a buffer for each shader type with an associated shader handle and allocate memory.

Next up is the main Ray Tracing function, used to call the execution of the ray tracer. It first binds the pipeline and it's layout, as well as sets the push constants. We also set up the Shader Binding Table and call the traceRaysNV() function. traceRaysNV() is fed in the Shader Binding Buffers for each shader group.

Finally, we update our main to alternate between rasterization and Ray Tracing via a GUI option. When switched to Ray Tracing, nothing is render as we haven't set up the shaders yet.

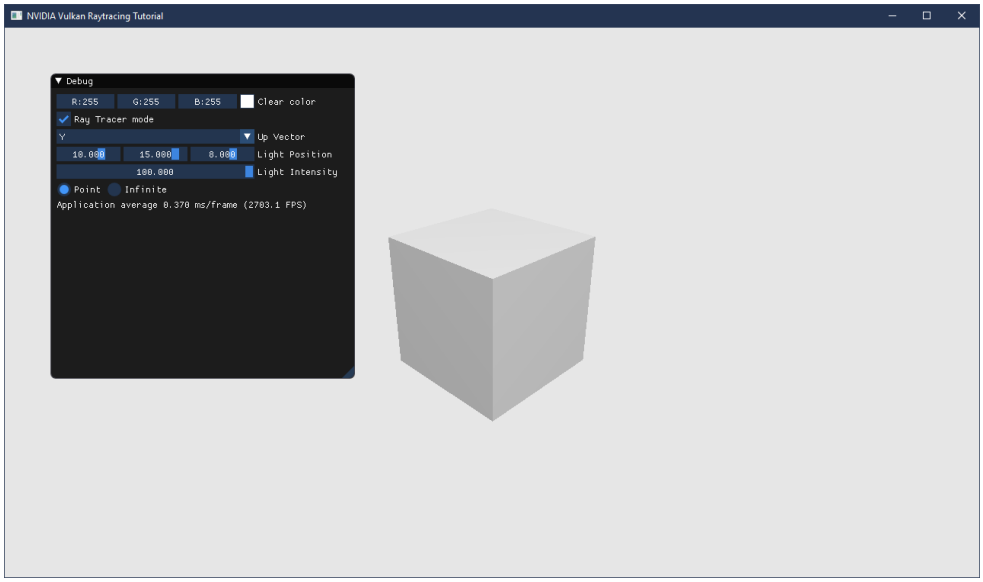


To get this started, a projection Inverse is set up in the camera matrix. This will be the point we generate rays from. We then add the camera properties to the rgen shader. The payload for the shader is reused many times, therefore is setup in raycommon.glsl, currently only containing a 'hitValue'. Rgen can now be written. It starts by computing a floating-point pixel coordinates, normalized between 0 and 1. From the pixel coordinates, we apply the inverse transformation of the view and projection matrices of the camera to get the origin and direction of the ray. We set all objects to be flagged as opaque, and then call traceNV(). traceNV() is fed the Top Level AS, ray flags, offsets, the origin, and other parameters. Finally, we can write the resulting payload into the output buffer.

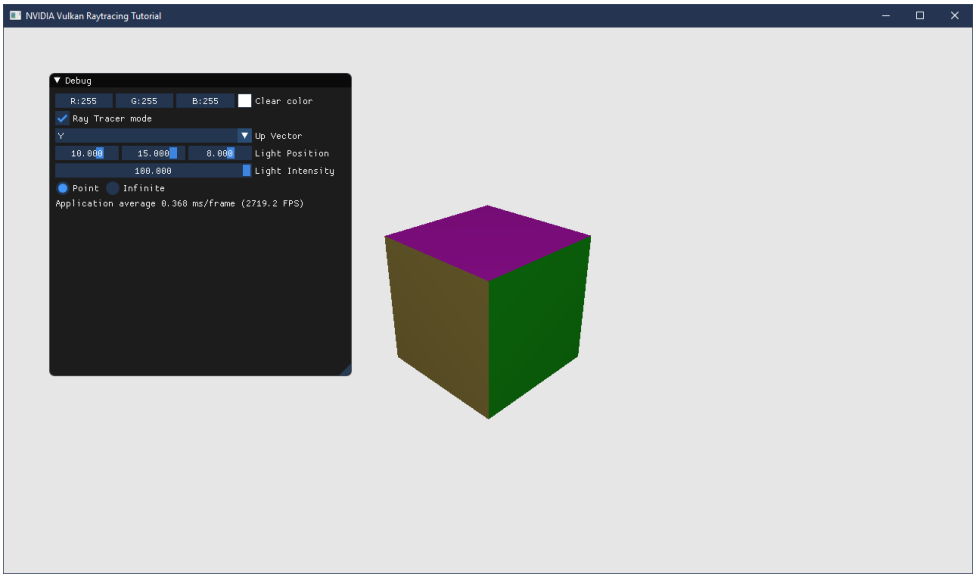


The miss shader is quickly implemented next. When called, the only thing it does is set it's hitValue to the 'clearColor', which is the background color what was originally set by the rasterization program.

Currently, the closest hit shader is only returning a flat color. To add lighting, we need to have surface normals. In the closest hit shader, we reference our descriptor set geometry definition, giving us the ability to reference the vertex and index buffers directly. We also set up our push constants, including the clearColor, light position, light Intensity, and lightType. Next, we find the vertices of a triangle hit by a ray using gl_PrimitiveID, then interpolate the normal. We calculate the world-space position of our light source and use the dot product of the normal with the lighting direction to create a diffuse lighting effect.

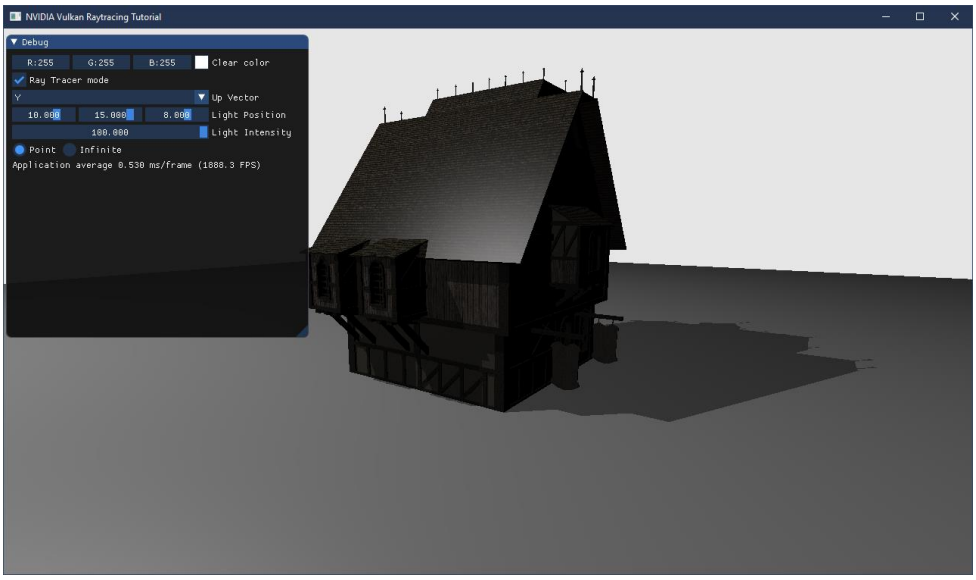


Now that we have rays being traced and lighting, the last step is to add materials to the objects we hit. In our rchit, we bind the material buffers that were set up when we imported the model's OBJ file. Instead of applying the dot product normal from before, we instead fetch the material. From this, we can use diffuse and specular reflectances for diffuse lighting, and create a new hit value.



Shadows

Shadows are implemented with a new miss ray that is shot from the closest hit shader. The first step to adding a new shader is updating the createRtPipeline() to check for and load in the .spv file as well as push the miss shader for shadow rays. We also have to update createRtDescriptorSet() to bind the new shader to our acceleration structure bindings. Lastly, we must update closest hit. We update the shader to be aware of the acceleration structure to be able to shoot it's down rays, and then set up a payload for those rays to flag if the ray isShadowed. We update rchit's main to fire off a shadow ray. If true, it will get rid of any specular lighting on the fragment in the shadow. Our final hitValue is also updated with 'attenuation', the darkness factor of a shadow.



Anti-Aliasing (Jitter)

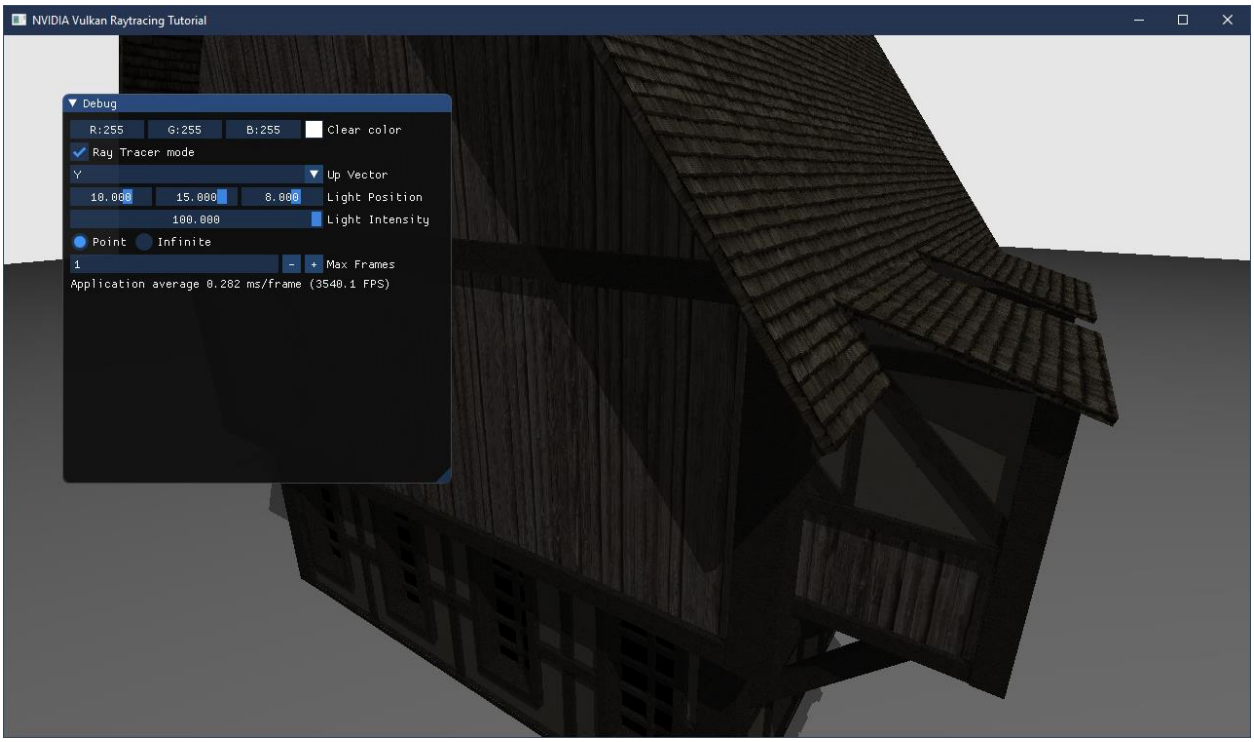
Jitter is used to implement antialiasing. This is done by jittering the offset of each ray for each pixel over time. We first create a new shader file called random.glsl which generates random unsigned integers from two unsigned integer values. Since our jittered samples are accumulated across frames, we now need to keep track of which frame we are rendering. Frame 0 will indicate a new frame. To track frames, we had int frame to our push constants. The frame counter is reset every time the camera is moved.

In rgen, we must initialize our random function by generating two random numbers and use them to redefine the pixel center when firing a ray, unless it's frame 0, which will always fire from the center of a pixel. At the end of rgen's main, we change our final written image. If it's on frame 0, it will write the image as normal. If not, we combine the new image with previous frames.

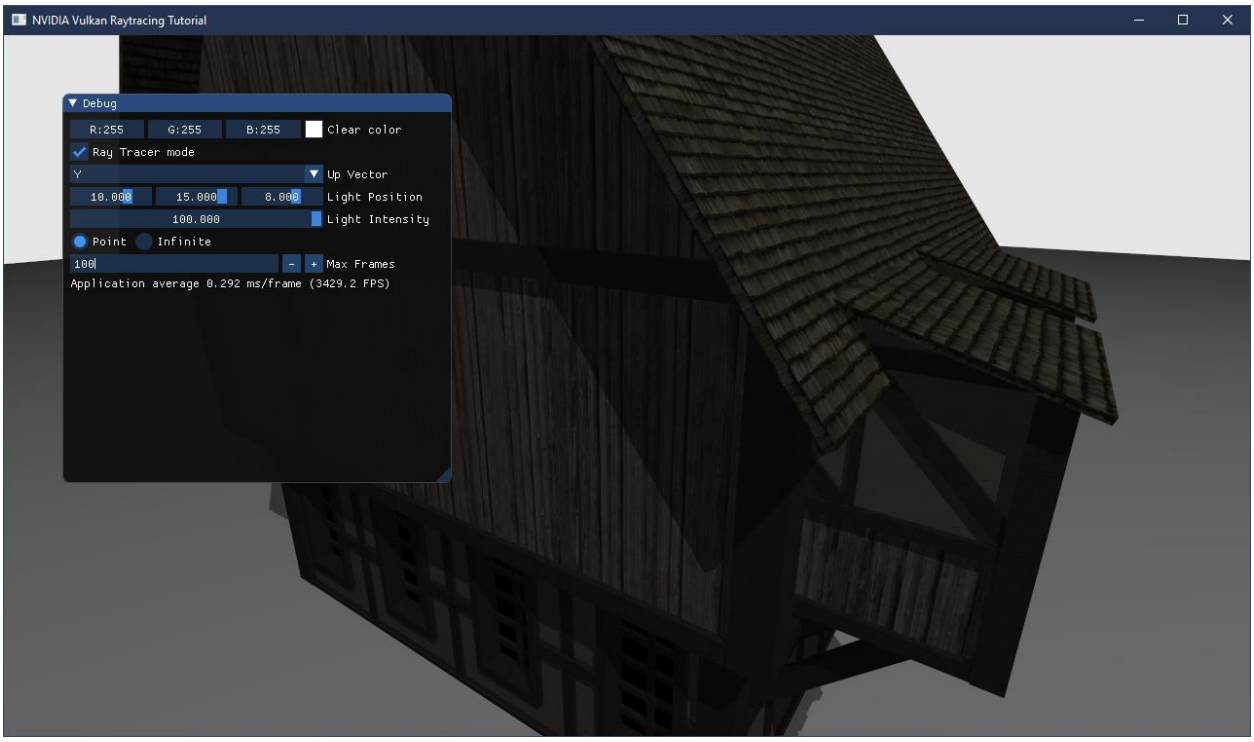
We create two new functions in HelloVulkan, updateframe() and resetframe(). Update frame resets the frame counter if the camera has changed, otherwise updates the frame counter. resetFrame() is called by updateFrame() if the camera moves and sets the frame counter to -1. The updateFrame() call is added to the very beginning of the raytrace() function.

We update our renderUI function in main.cpp to reset the frame counter every time we make a change, since we need to start over and re-accumulate frames if a change is made.

Lastly, we set an upper limit on the number of samples to avoid accumulating too many frames. A m_maxFrames{100} is added to the main program and is used to make updateFrame() return should the max limit be reached. To increase efficiency, we can perform multiple samples directly in the rgen shader. We add a constant to the rgen shader, and put our random number generator as well as our traceNV() call in a loop, only to run the number of times set by the constant. At the end of the loop, we divide by the number of samples that were taken.



Set to 1 max frames (no aliasing)



Set to 100 max frames (aliasing)

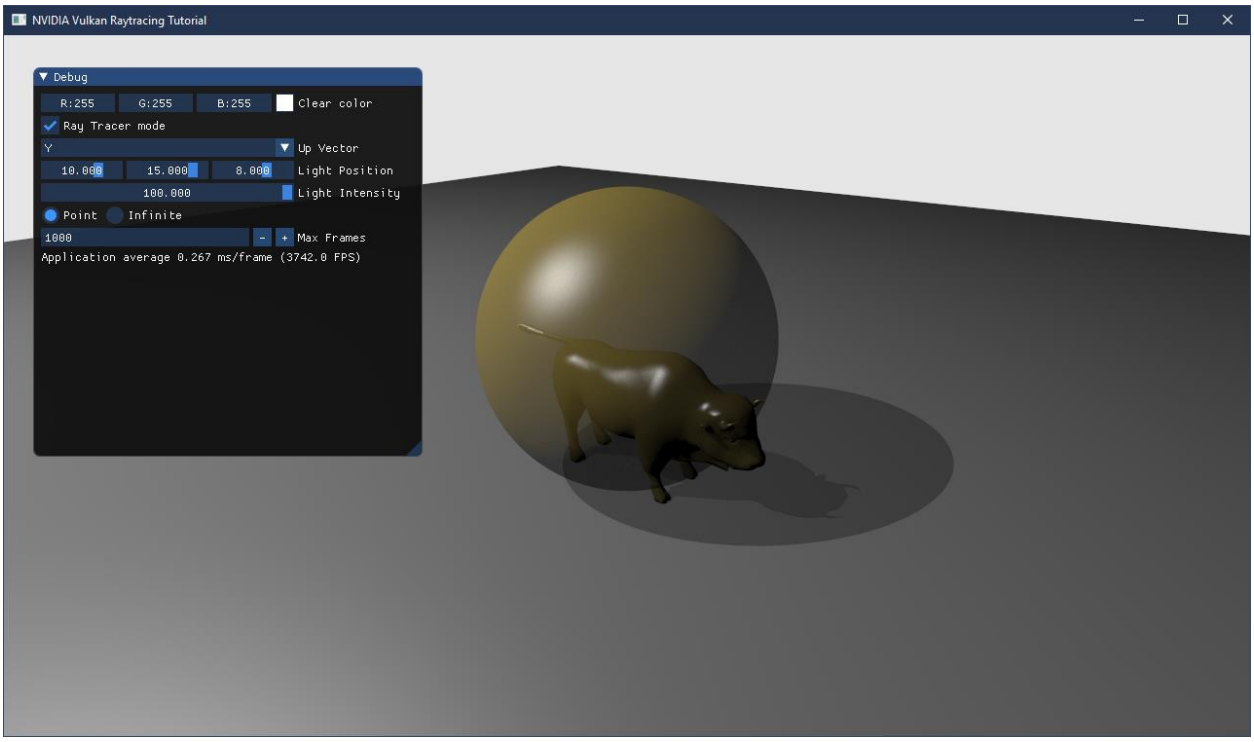
Transparency (Anyhit)

To implement transparency, we use the anyhit shader. The anyhit shader differs from the closesthit shader as anyhit will be executed for all hits along the ray where closethit only executes on the closest acceptable hit.

We add the anyhit shader (rahit) which is similar to the hit shader, but uses much less information. The anyhit shader will be looking at the material of an object it intersects with. If the object is opaque, it simply returns. If not, we add transparency by using our random number generator to determine if a ray hits or should be ignored. With enough accumulated rays, the result converges to a transparent effect.

A seed value is added to our raycommon, updating our payload. Following a similar process from before, we also add our new rahit shader to the createRtPipeline() and to createDescriptorSetLayout() to initialize the shader and give us access to the material and scene description buffers. We also have to remote a previously set flag that set all objects to opaque.

We also have to make small adjustments to the rgen and rchit shaders. In rgen, we have to update the seed generation and update the rayFlags. In rchit, we also have to update the flags to allow for the anyhit shader to be called.

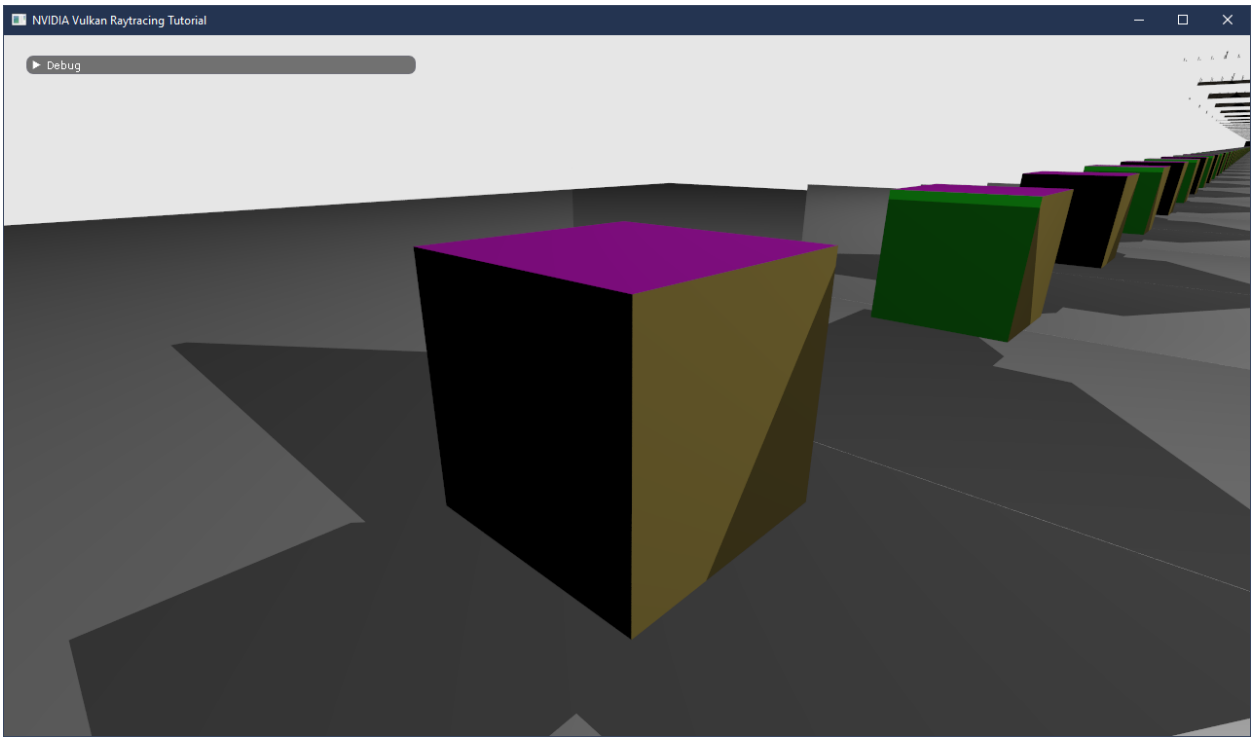


Reflections

Our implementation of reflections is done by iteratively shooting rays from the raygen shader when intersecting with a reflective object.

In our raycommon, we update the payload to include a depth integer. This will control how many bounces can be done when a mirror reflects a mirror and creates the infinity effect. We also add rayOrigin and rayDir. These both need to also be set in rgen based on the rays current xyz. With rgen's TraceNV() call in a loop, we're able to generate new rays when a reflective surface is hit.

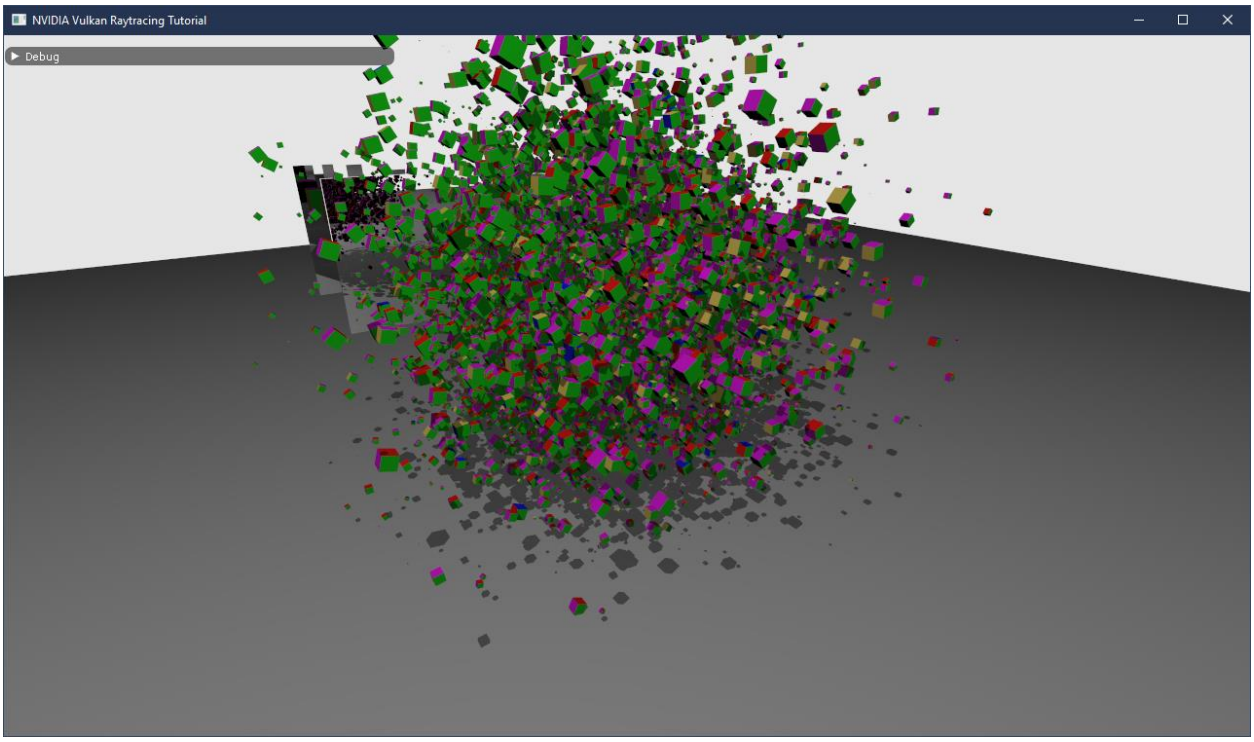
Rchit is also updated to calculate the reflections world position, ray direction, and new attenuation. We add a maxDepth to our RtPushConstant to be passed to the shader, as well as it this to our rgen push constant.



Device Memory Allocator (DMA)

Currently, our program has a maximum memory allocation problem. Many Vulkan implementations support no more than 4096 allocations, with our current implementation creating 4 allocations per object (Vertex, Index, material, and Bottom Level AS), meaning we have a limit of ~1000 objects. We will implement a memory allocator to fix this issue.

In our header, we'll need to include ALLOC_DMA, replacing ALLOC_DEDICATED and switch the buffer allocator and texture allocator to use the DMA. In hello_vulkan.cpp, we update setup() to use the DMA, along with the updateUnifromBuffer() and initRayTracing() functions. The result is significantly fewer allocations required for thousands of objects.



A random function is used to generate thousands of random cubes to test DMA

Intersection Shader

Previously, we were rendering sphere's by loading them in as OBJs, not primitives. For primitives, we need to use the intersection shader.

First we need to create actual spheres. We do this by updating the HelloVulkan class with a sphere struct containing center and radius. We also create an Axis aligned bounding box (Aabb) struct to hold the minimum and maximum for the geometry type. Then, we set up buffers for spheres, Aabb primitives, sphere materials, and material index.

We'll have to create two functions, createSpheres() and spheretoVkGeometryNV(). The createSpheres() function creates 2 milion spheres at random positions and radius as well as create the Aabb and assign materials. All will be stored in Vulkan buffers for the shaders to access. The spheretoVkGeomtryNV() function will be used to create a new bottom level acceleration structure to hold the implicit primitives.

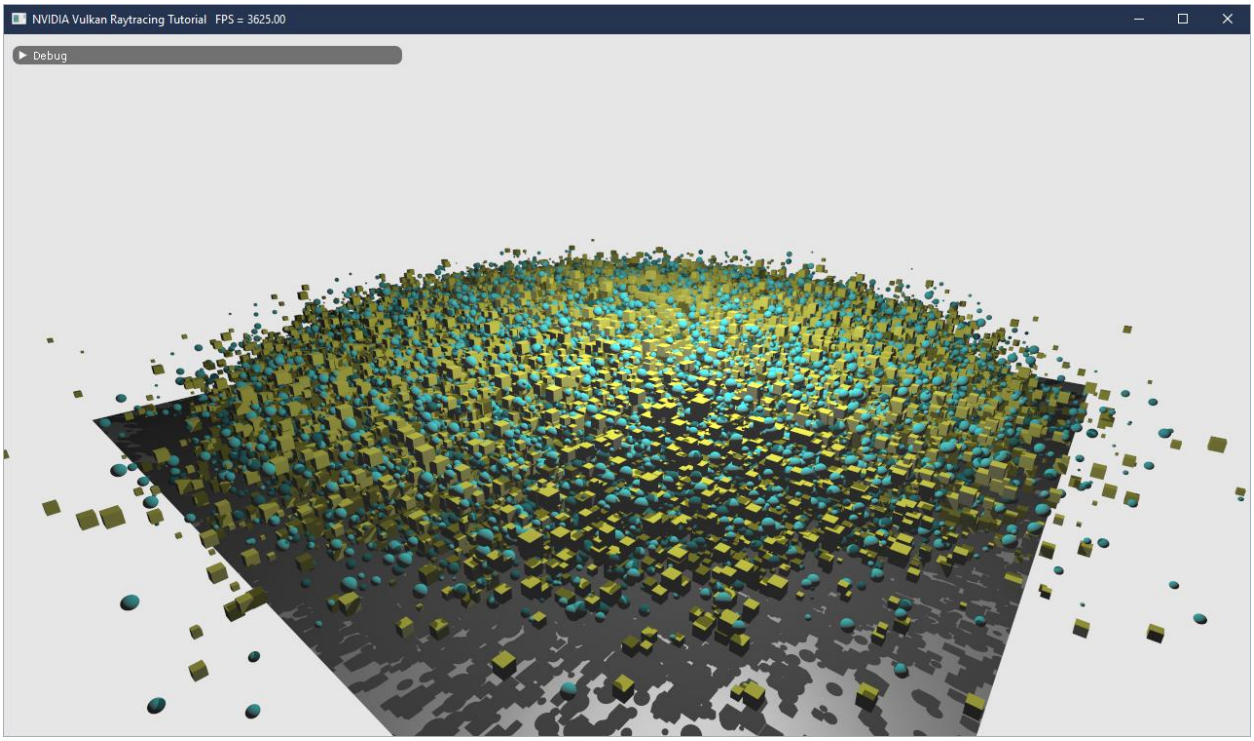
The original createbottomlevelAS() and createTopLevelAs() must next be modified. createBottomLEvelAS() is modified to add a new BLAS containing the Aabb's of all spheres. createTopLevelAS() needs a reference to the BLAS of spheres. We also change the hitGroupId to put implicit primitives in their own hit group.

To access the newly created buffers, we need to update the createDescriptorSetLayout() function to include the new buffers. We also modify updateDescriptorSet() to include the materials of these new buffers. We also update the raytracing pipeline with the intersection shader and it's hit group.

Raycommon is updated to share the structure of the data across all the shaders, this includes our sphere struct and Aabb struct from earlier.

Now we can start adding the intersection shader, rint. It will be called every time a ray hits one of the Aabb of the scene. Setting it up like other shaders, we also need to include the HitAttribute and sphere buffers. We will set up a ray struct with the origin and direction to be used for calculations. We then add two functions to shader, hitSphere() and hitAabb() that include all the intersection math for their respective types of hits. In the main of the shader, we initialize the ray's data and call our calculation functions for the types of hits.

Another shader we need to add is a second closest hit. It's similar to the original rchit shader, but will only be used for implicit primitives to compute the normal of the primitive that was hit. We retrieve the world position from the ray and intersection shader, retrieve the sphere information, and then compute the normal. If we hit a primitive cube, we set the normal to the major axis.



Issues Encountered

Initial Issues: When I first started the project, I had issues running the cmake file that built the tutorial. These issues included references to files in the Vulkan SDK that didn't exist and cmake directory issues. After spending quite some time troubleshooting, I created an issue on the main tutorial repo and it was fixed within 24 hours.

Callable Shader: The tutorial for the callable shader caused disastrous issues. It's meant to create a directional like (rather than the normal point light) using callable shaders. After finishing the tutorial, I noticed it still wasn't working and there were other build errors. After troubleshooting and fixing the

errors, I got the callable shaders working at the expensive all almost everything else. Antialiasing, reflections, and transparency were all broken. Reverting my code back several iterations for some reason didn't fix the issue, so I had to blow away my repo and start it over again.

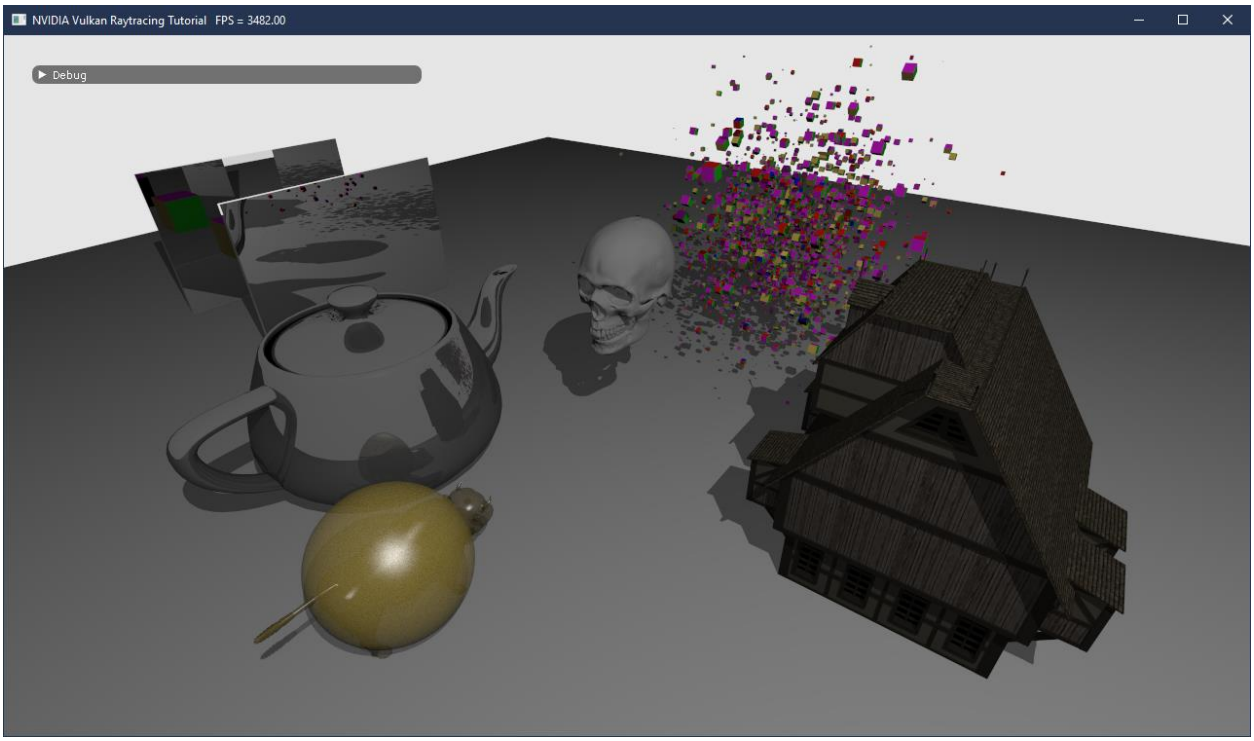
Intersection Shader: The intersection shader, while working, is buggy in its current form. For whatever reason, the primitives are only generating correctly if they are the only thing in the scene. If they aren't, then they can only be scene at certain angles from above. I have no idea what's causing this issue.

Animation: The tutorials were not meant to be combined in the way I've been doing. My current project has many modifications over the base code and unfortunately animation doesn't work with the current modifications. Animations in this tutorial work by modifying both the top level and bottom level accelerated structures. Since mine have already been heavily modified, animations were not something I could get working.

Multiple Closest Hits Shader and Shader Records: I skipped this tutorial as the material is basically already covered. The point of this tutorial is to assign different instances different shaders, creating different effects for different objects. We do this exact same thing in the intersection shader tutorial as we must call the intersection shader and 2nd rchit shader when a primitive is hit. A great application for this would be lower quality renders for far away or reflected objects.

Conclusion

This project was a labor of love. It was awesome to learn Vulkan and Real-Time Ray Tracing by going through this project, but it's obvious the technology is in its early stages. The number of external libraries required outside of Vulkan leaves a bit to be desired, and Vulkan's unfriendly built-in debugging lead to quite a few printf statements. I plan on continuing this project in my spare time to hopefully fix the issues I've had as well as update it as Vulkan standardizes Ray Tracing.



Sources

NVIDIA's Real-Time Ray Tracing with Vulkan: https://devblogs.nvidia.com/vulkan-raytracing/

NVIDA's RTX Ray Tracing: https://developer.nvidia.com/rtx/raytracing

Good reading: https://www.realtimerendering.com/raytracing.html

NVIDIA's VK_Raytracing_tutorial: https://github.com/nvpro-samples/vk_raytracing_tutorial

Vulkan SDK: https://vulkan.lunarg.com/

NIVIDIA's Shared_Sources Repo: https://github.com/nvpro-samples/shared_sources

NVIDIA's Shared_External Repo: https://github.com/nvpro-samples/shared_external