15-468 Project Report

Max Slater
mjslater
15-468 Section 1/A

1 Features

Simple:

1. GGX and Blinn-Phong BRDFs
2. Normal maps
3. Tone mapping
4. QMC sampling (camera rays)

Advanced:

1. Real-time path tracing with RTX in Vulkan. Similar features to DIRT:
   (a) Integrators: material path tracing, direct lighting, next event estimation/MIS
   (b) BRDF importance sampling, sampling emissive triangle meshes, russian roulette
   (c) GLTF scene loading (geometry, materials, textures)
2. Direct lighting sampling with ReSTIR
   (a) Reservoir re-sampling
   (b) Temporal re-use
   (c) Spatial re-use (not completed yet)

2 Implementation

Shader code may be found in src/shaders. Vulkan code may be found in src/vk. Renderer orchestration code may be found in src/gpurt.cpp. A GPU with hardware RT support is required to run my renderer. I developed it on an RTX 3090, so I'm sure it will work with any 30-series NVIDIA GPU. It should also work on the 20-series and AMD 6000 series, but I have not validated this.

My renderer is based on the hardware accelerated ray tracing extension standardized in Vulkan 1.2. The extension's API allows us to build ray tracing specific GPU pipelines that consist of ray generation, any/closest hit, intersection, and miss shaders. The RTX hardware implements BVH traversal as well as ray-bounding box, and ray-triangle intersections. Building acceleration structures (BVHs) is also GPU accelerated through special build commands.

To construct the renderer, I connected together a ray generation, closest hit, and miss shader. Each frame, I invoke the ray generation shader for each pixel of the output image. This shader samples camera rays for each pixel before iteratively performing intersections and shading with the scene. Intersections and visibility tests are performed with the implementation defined traceRayEXT function, which performs BVH traversal and invokes either the closest hit or miss shader. After

sampling, the ray generation shader blends the new samples onto the output image, which allows the result to converge over several real-time frames. It also writes out the first-hit position, normal, and albedo to separate images, which are used as a g-buffer for ReSTIR temporal re-use.

The closest hit shader returns a minimal amount of information about the intersection to the ray generation shader. All information about the scene (object transforms, geometry, lights, textures) are gathered into contiguous buffers accessible by the ray-gen shader, so it may look up the intersection geometry, material properties, and shading information at the hit point. This approach gives far higher performance than doing sampling and recursive tracing in the closest hit shader. Finally, the ray-generation shader calls the specified integration function.

Each integrator uses Blinn-Phong or GGX importance sampling, also implemented in shader code. Lights are sampled by uniformly choosing an emissive object, and then uniformly choosing a triangle on that object.

The ReSTIR integrator uses reservoir resampling to choose the best light sample of several per iteration. It also finds the corresponding reservoir from the previous frame by re-projecting the hit position with the old projection-view matrix. If the geometry at the previous pixel is close enough to the current hit geometry (position, normal, and albedo), it combines the previous (temporal) reservoir with the newly sampled one. Checking the g-buffer reduces bias to an acceptable level, as only reservoirs that can produce similar samples are combined. However, keeping the algorithm entirely unbiased would involve detecting exactly whether or not samples could have been produced by the previous frame, which involves much costlier intersection tests.

ReSTIR spatial re-use would further reduce noise, but I didn't get around to adding another render pass for this step. It would be straightforward to add, since I already write out a buffer of the current reservoirs for temporal re-use—the spatial pass simply samples a disk at each screen-space reservoir and attempts to combine with neighbors. I plan on adding this next week (currently have to travel).

3 Results

3.1 Integrator Tests

To validate correctness, I started by comparing my BRDF path tracing result with an example from NVIDIA (the cornell box). I then made sure that all other integrators converged to the same result as the BRDF integrator.

The 60 FPS results show the render after one frame rendered in 16.66 ms. This is what it looks like while the camera is in motion, or the scene geometry is moving (everything can be dynamic).

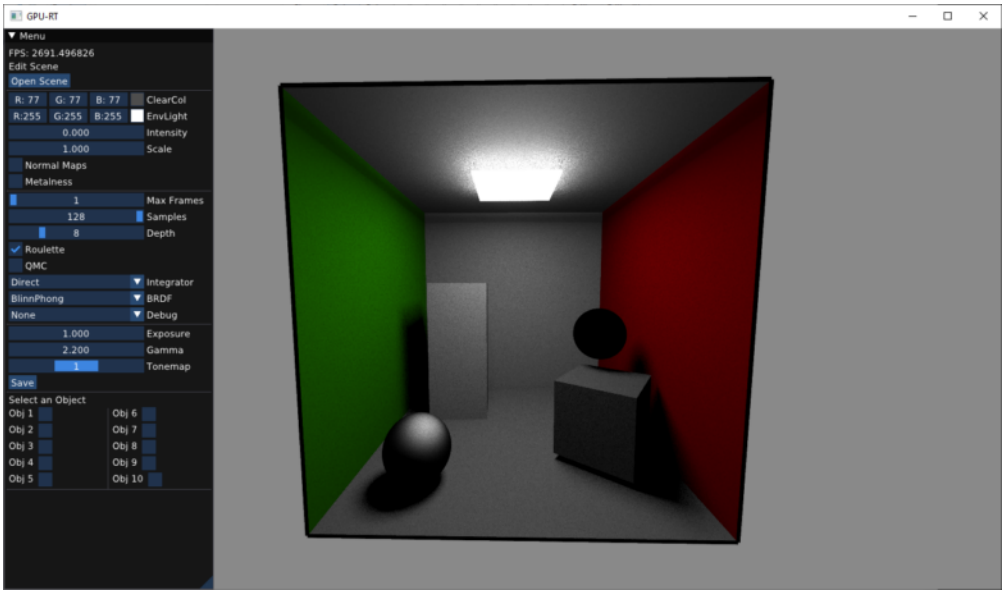

Figure 1: Direct lighting (128 SPP @ 60 FPS)

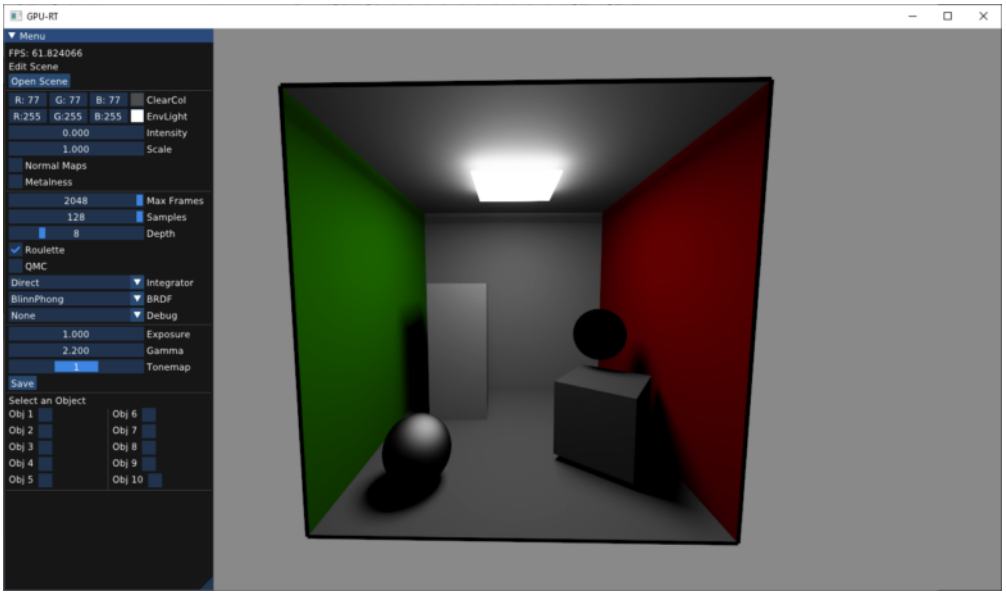

Figure 2: Direct lighting (converged)

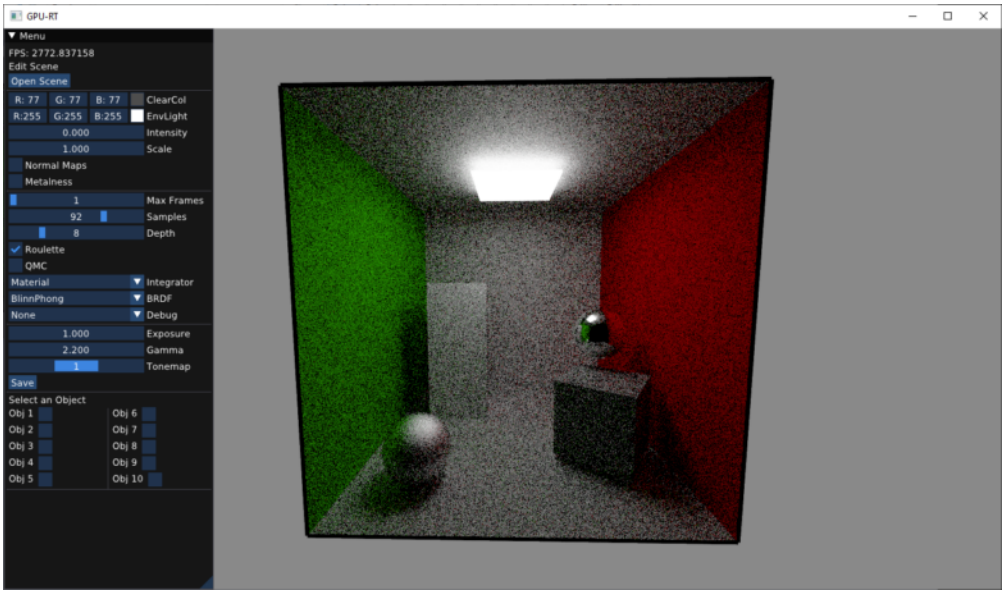

Figure 3: BRDF Path Tracing (92 SPP @ 60 FPS)

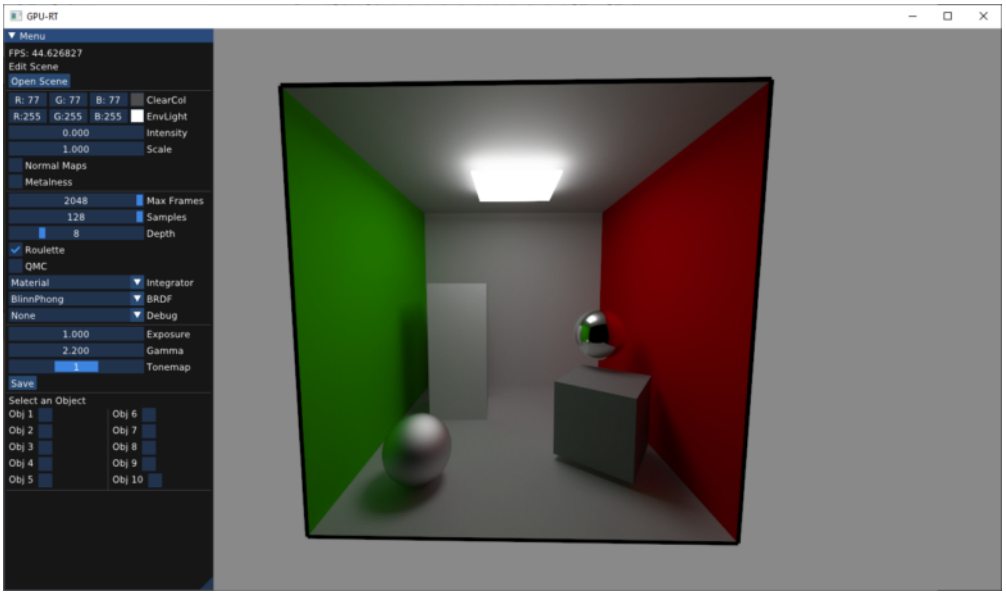

Figure 4: BRDF Path Tracing (converged)

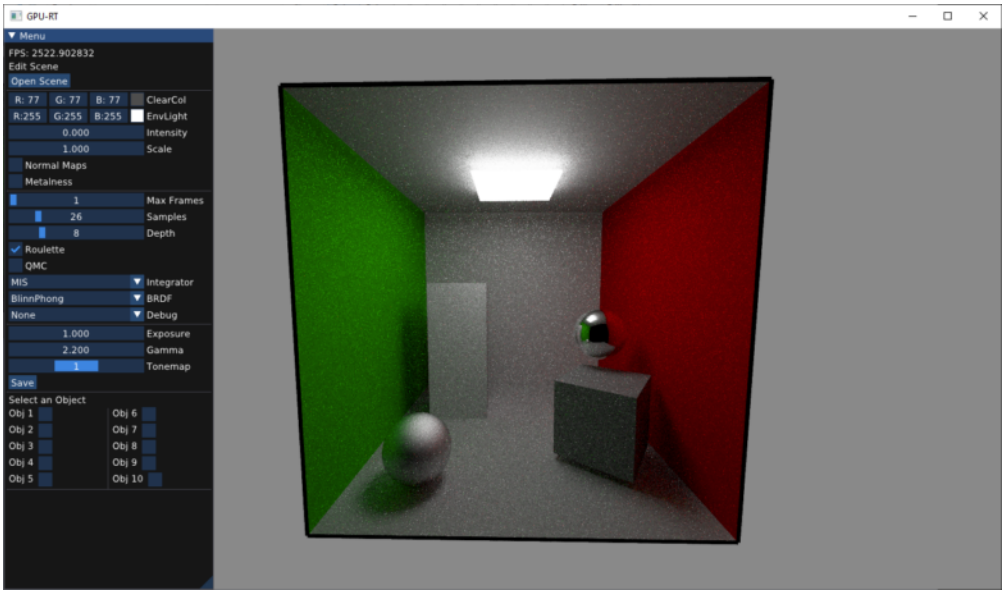

Figure 5: NEE + MIS Path Tracing (26 SPP @ 60 FPS)

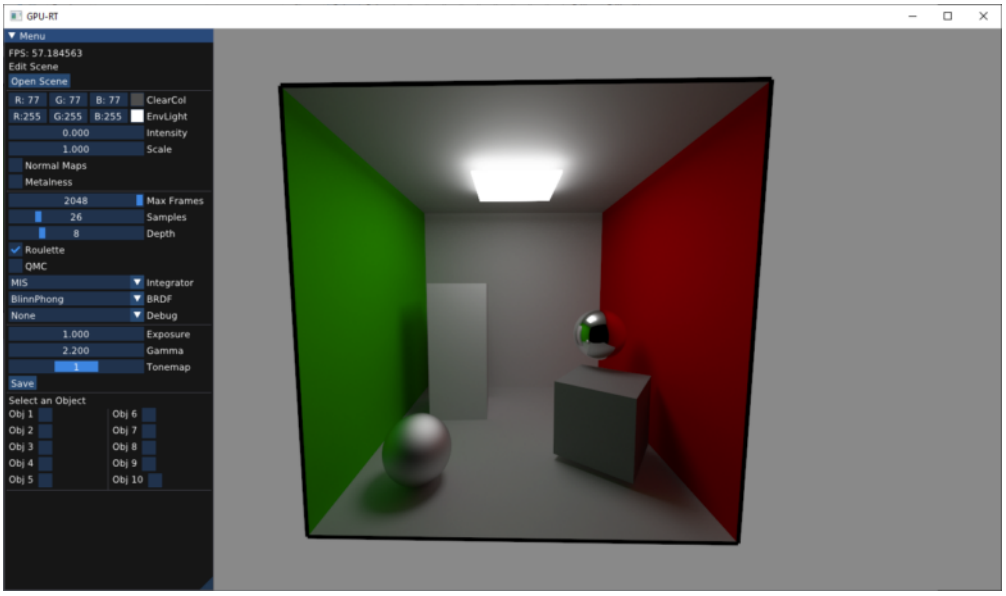

Figure 6: NEE + MIS Path Tracing (converged)

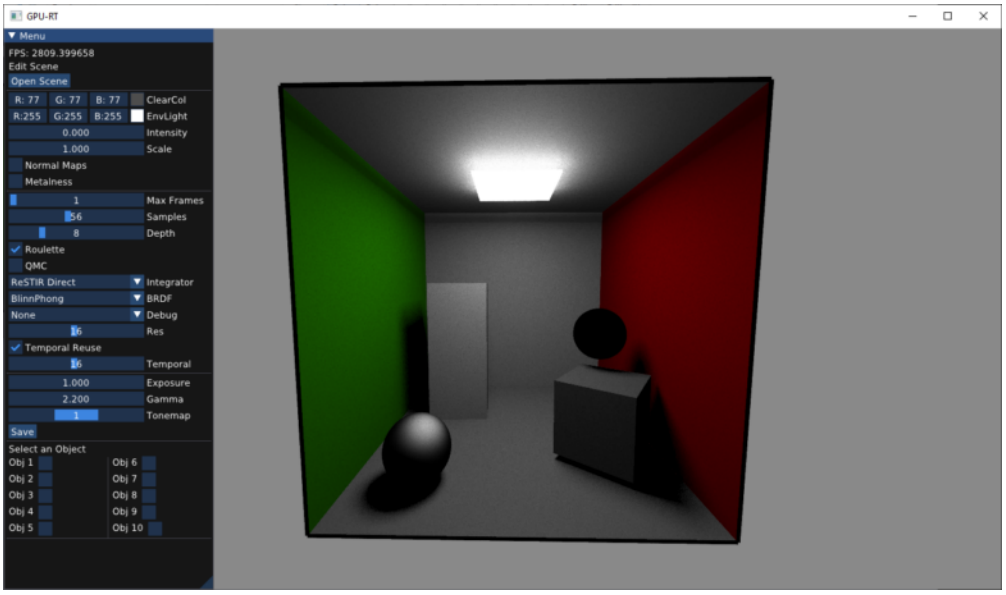

Figure 7: ReSTIR Direct Lighting (56 SPP, 16 light samples @ 60 FPS)

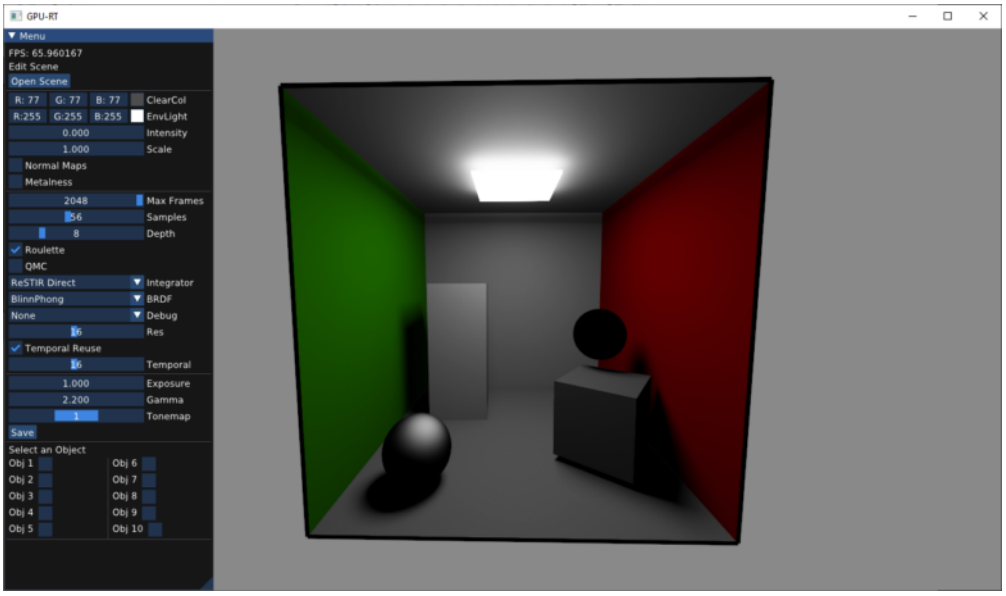

Figure 8: ReSTIR Direct Lighting (converged)

3.2 Simple Features

3.2.1 BRDFs

Both types (Blinn-Phong and GGX) are parameterized by roughness and metalness specified by material properties or textures.

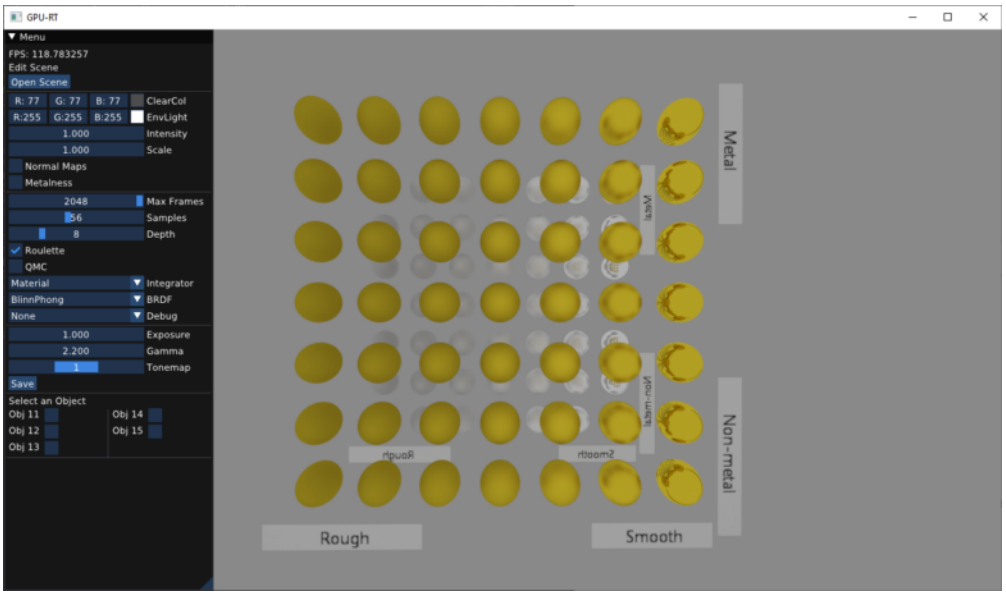

Figure 9: Blinn-Phong Test

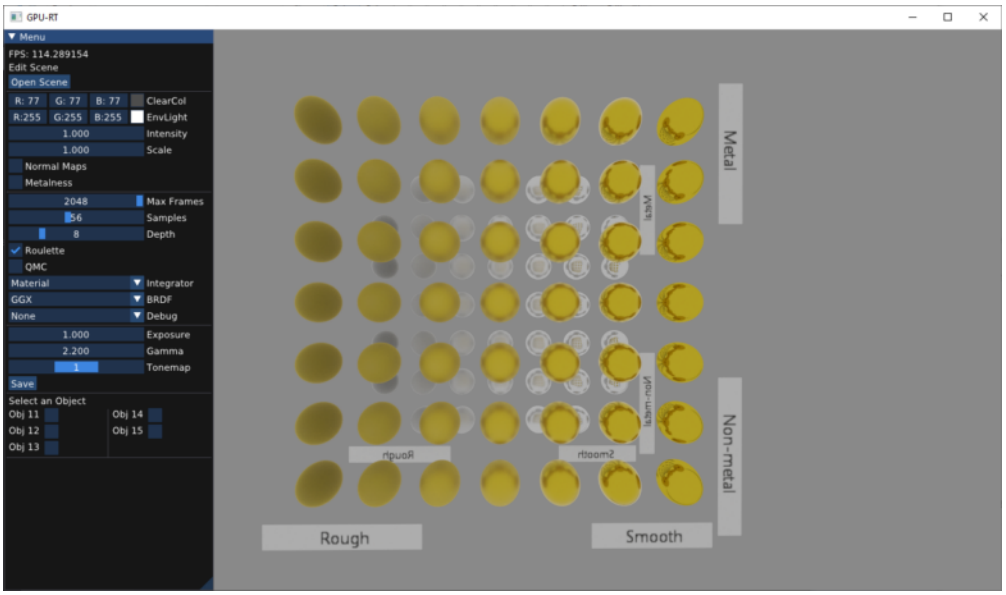

Figure 10: GGX Test

3.2.2 Tone Mapping

I implemented a simple exponential tonemapping operator, as well as the Uncharted 2 tonemap. Tonemapping and gamma correction is implemented as a post-processing pass that converts the HDR output of the path tracer to an LDR image for display.

The following examples are the Sponza scene lit by a bright sky-light.

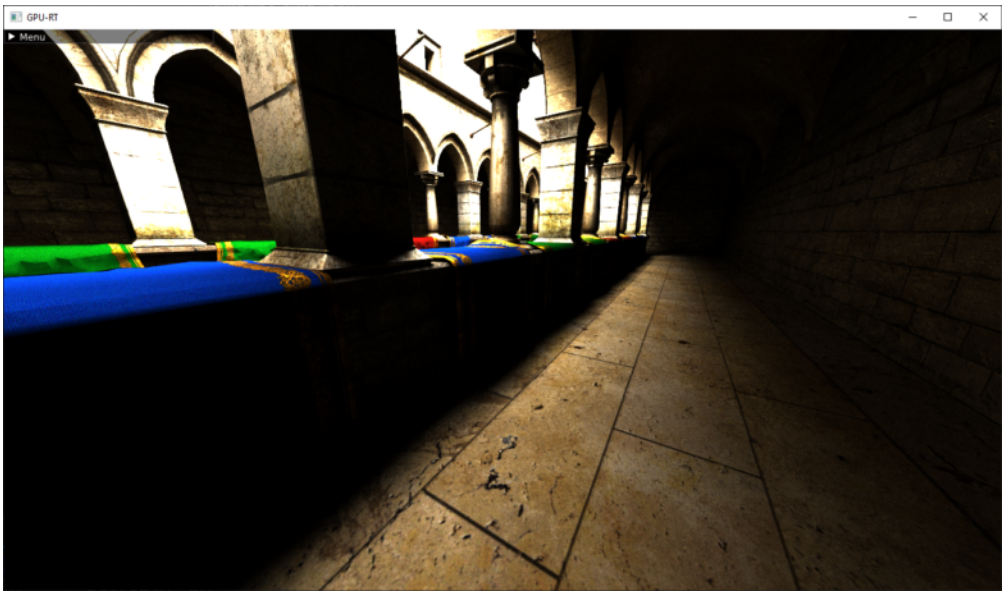

Figure 11: No Tonemap

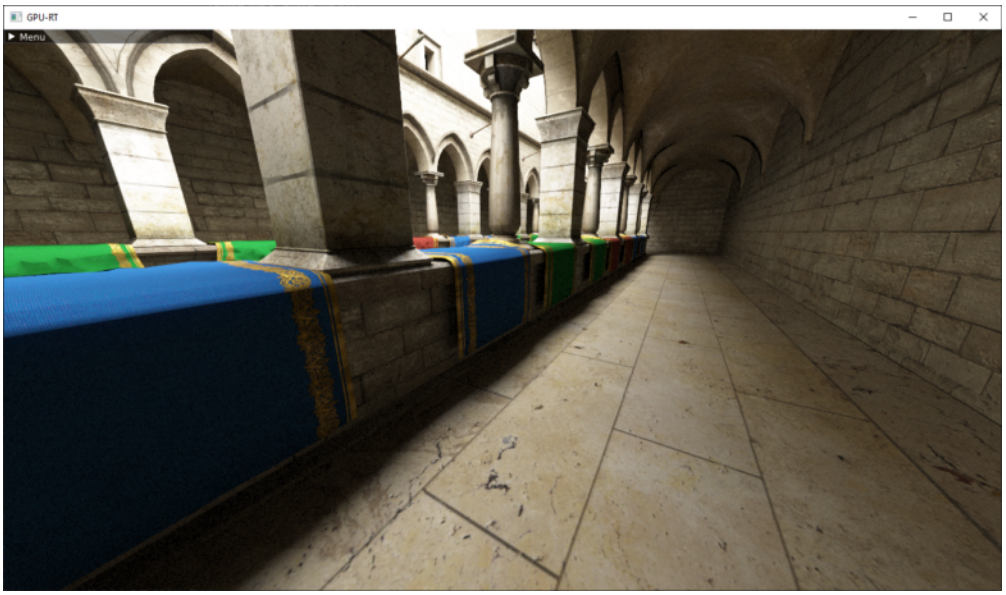

Figure 12: Exponential Tonemap

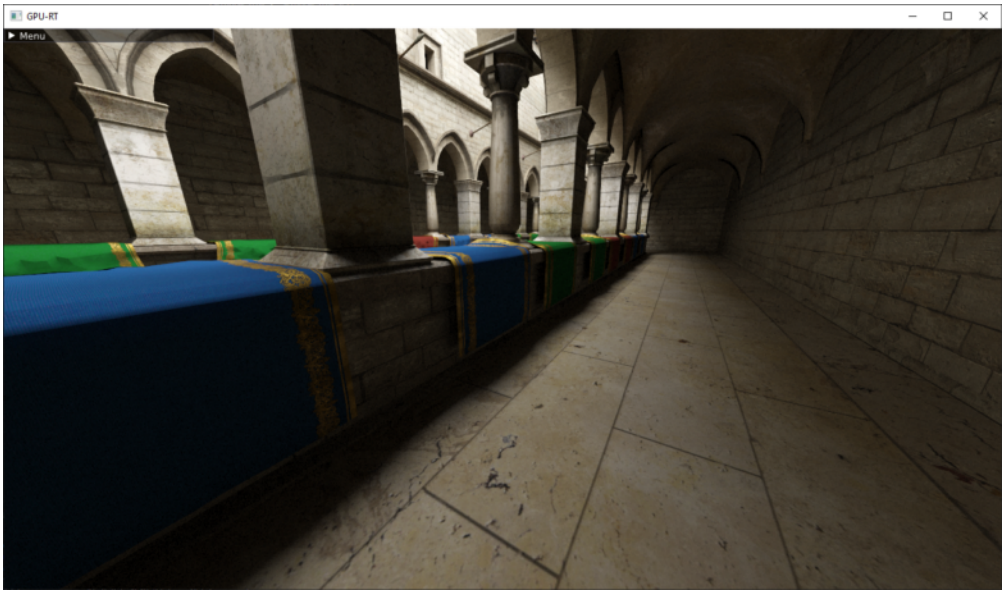

Figure 13: Uncharted 2 Tonemap

3.2.3 Normal Maps

Normal maps improved surface detail when looking at them head-on, but for many surfaces at grazing angles, the map redirects most BRDF rays into the surface itself. This causes the material to appear black, so normal mapping is not very well suited for path tracing—bump mapping would be a better choice.

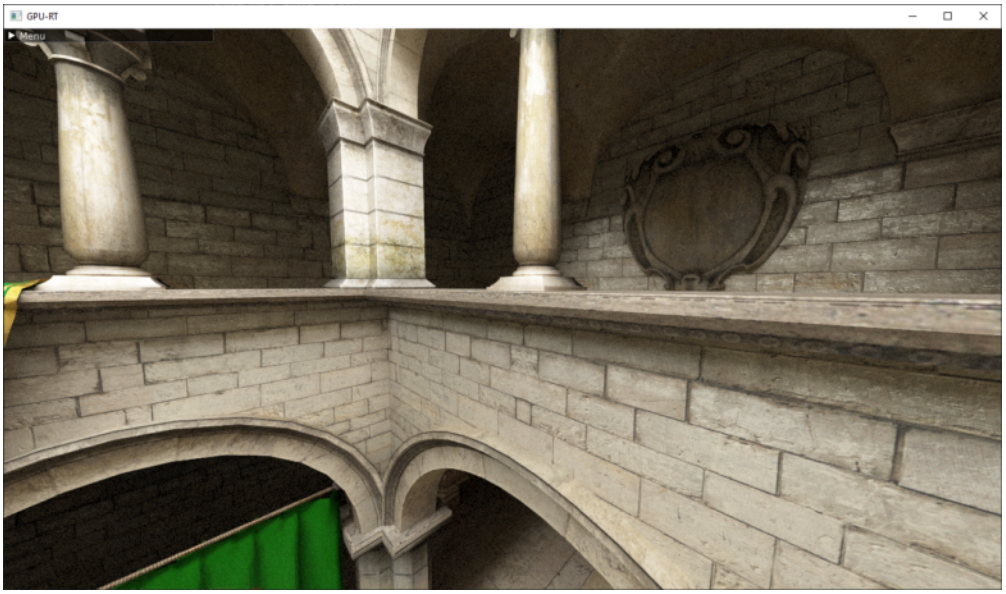

Figure 14: No Normal Maps

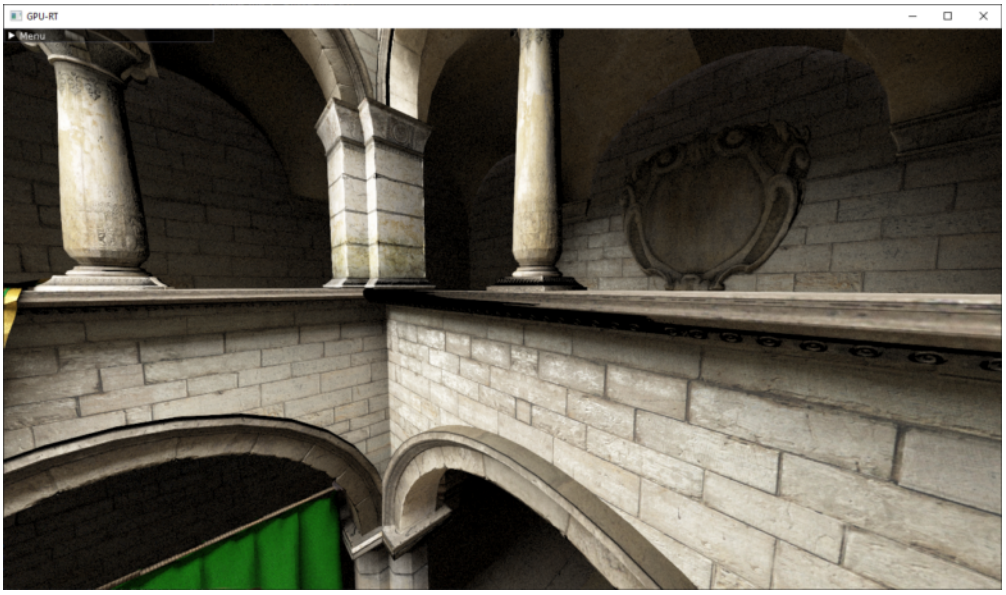

Figure 15: Normal Maps

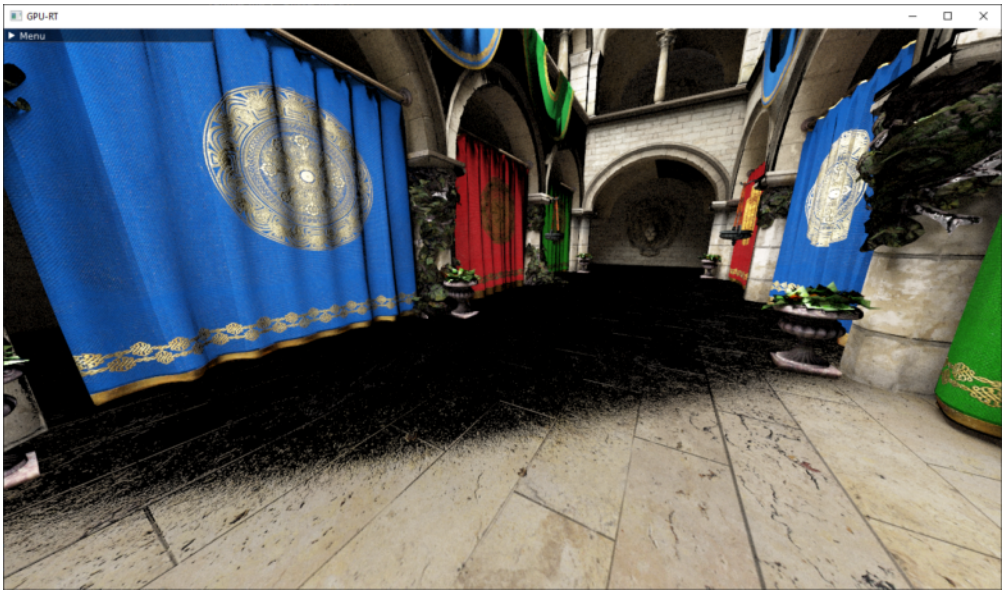

Figure 16: Normal Map BRDF Issues

3.2.4 Quasi Monte Carlo Sampling

I used a Hammersley sequence for generating camera ray samples. This didn't make a huge difference, but did improve the convergence rate of anti-aliasing.

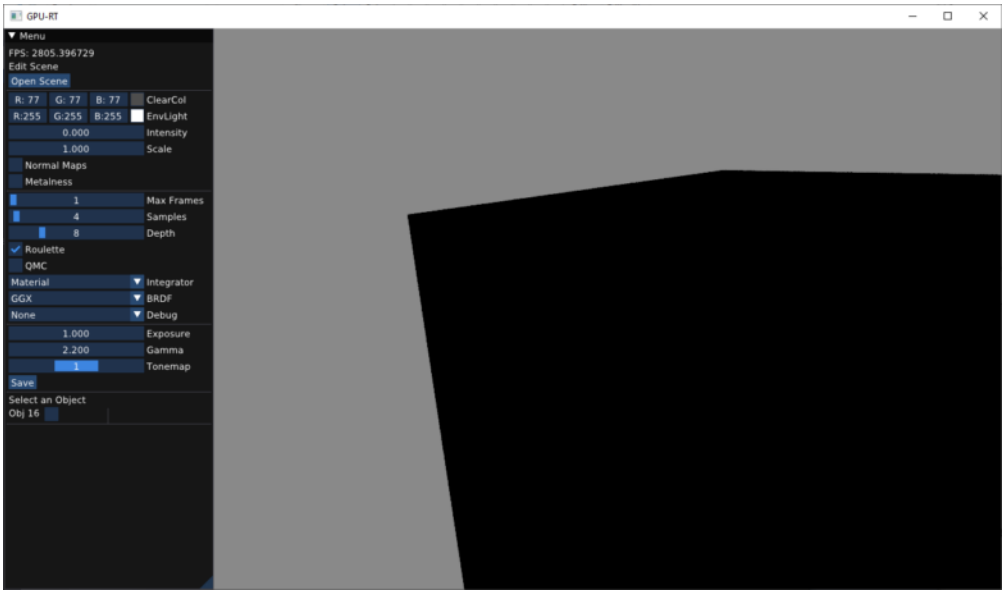

Figure 17: Uniform Pixel Samples

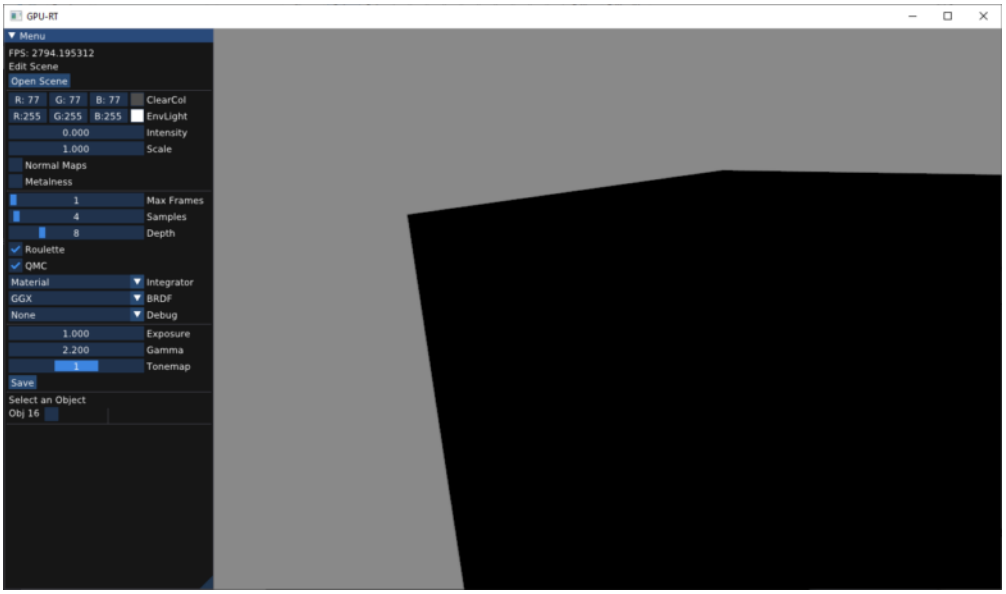

Figure 18: Hammersley Pixel Samples

3.3 MIS Test

I made a Veach-esque MIS test scene to validate my multiple importance sampling:

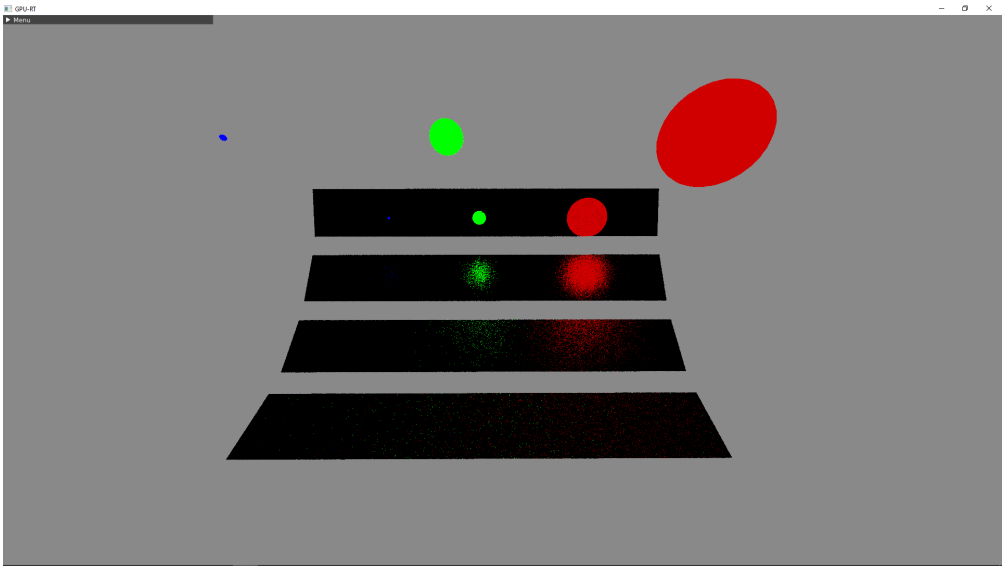

Figure 19: BRDF Sampling

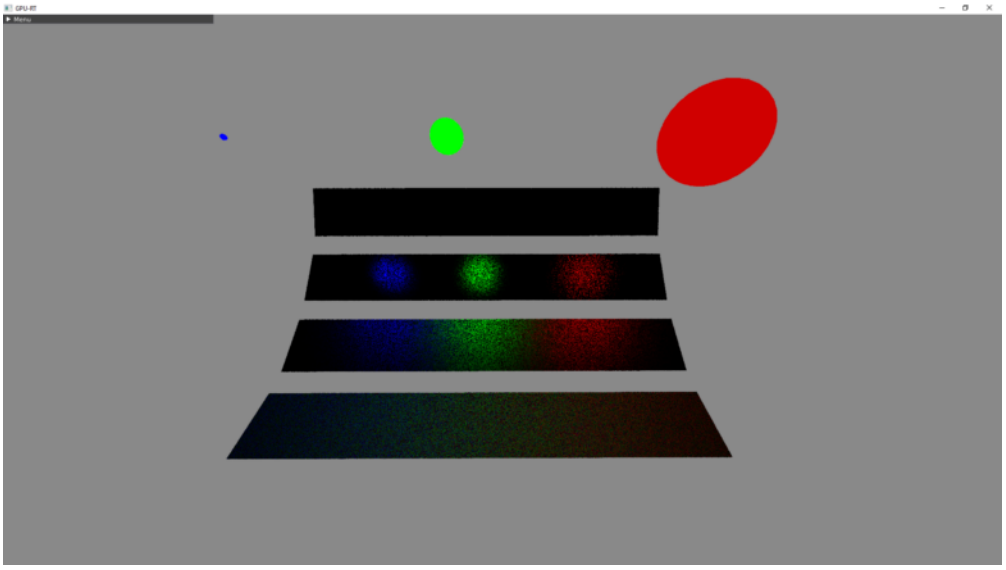

Figure 20: Light Sampling

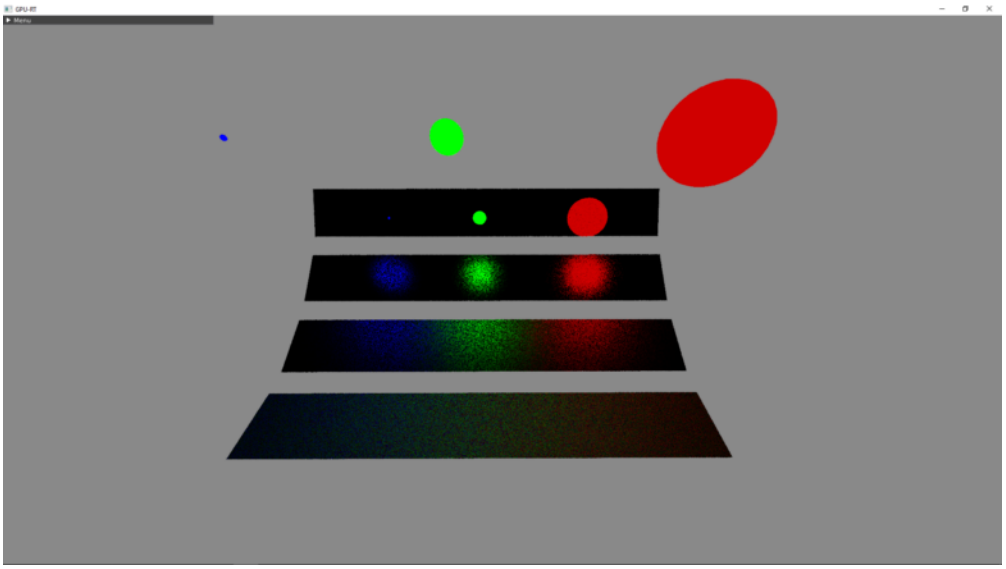

Figure 21: MIS

3.4 ReSTIR

To dest the effectiveness of ReSTIR, I used Amazon's Bistro scene. This scene is difficult to sample, as it is lit by many small emissive meshes scattered throughout the street. Particularly in the (green) back alley, the majority of the light comes from a single small green sign, and all the other lights are either occluded or too far away.

With BRDF sampling, this scene is awful at 60 FPS:

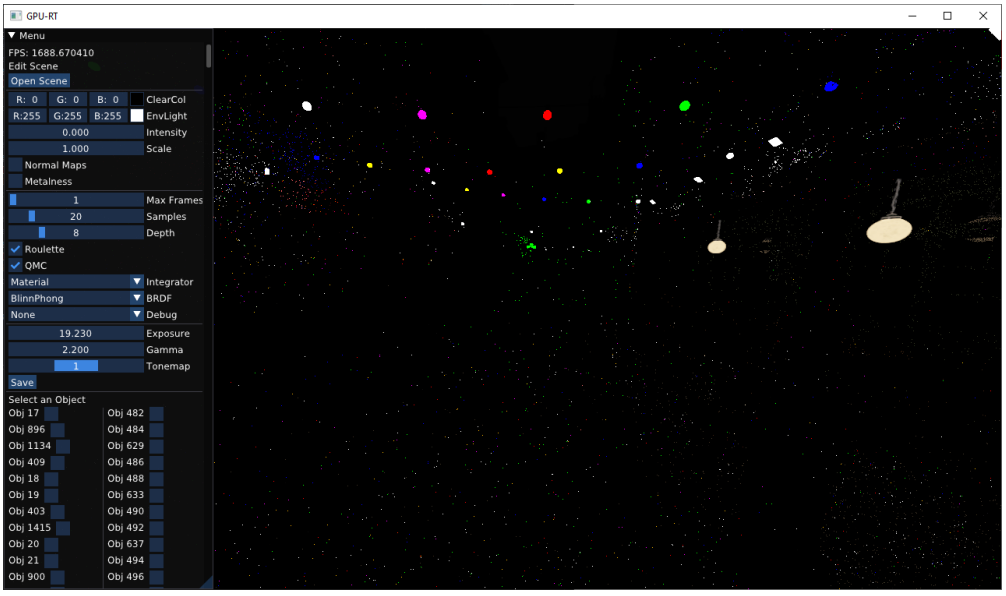

Figure 22: BRDF Path Tracing (20 SPP @ 60 FPS)

With MIS, the scene is better, but runs extremely slowly due to iterating all emissive objects in

order to compute the light PDF for the BRDF sample. This is accelerated by checking the ray against the object bounding box before iterating each triangle in an emissive object, but still only runs at 4 samples/second.

Figure 23: MIS Path Tracing (1 SPP @ 4 FPS)

For direct lighting, uniform sampling gave the following result:

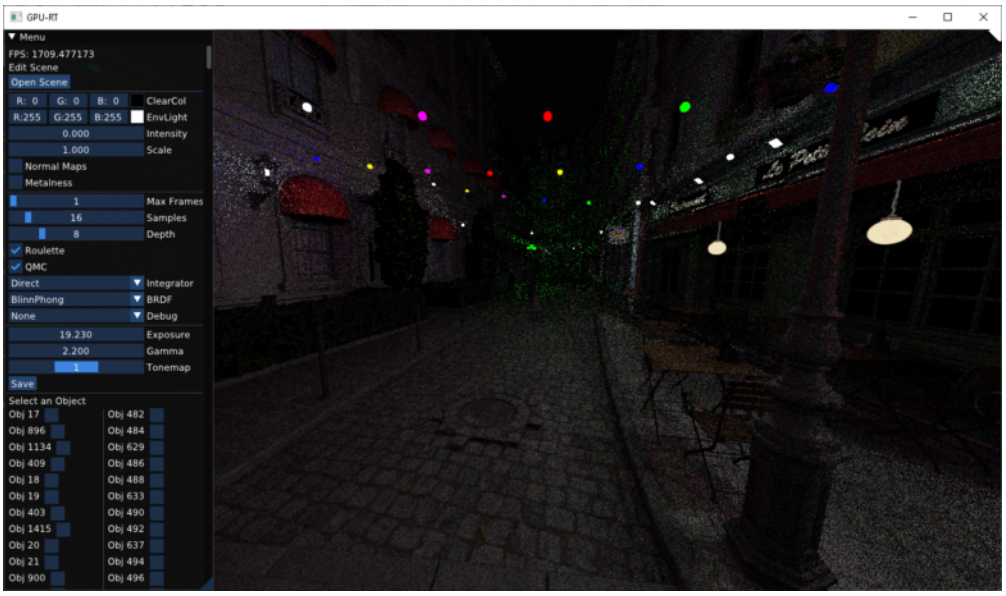

Figure 24: Direct Lighting (16 SPP @ 60 FPS)

And with ReSTIR:

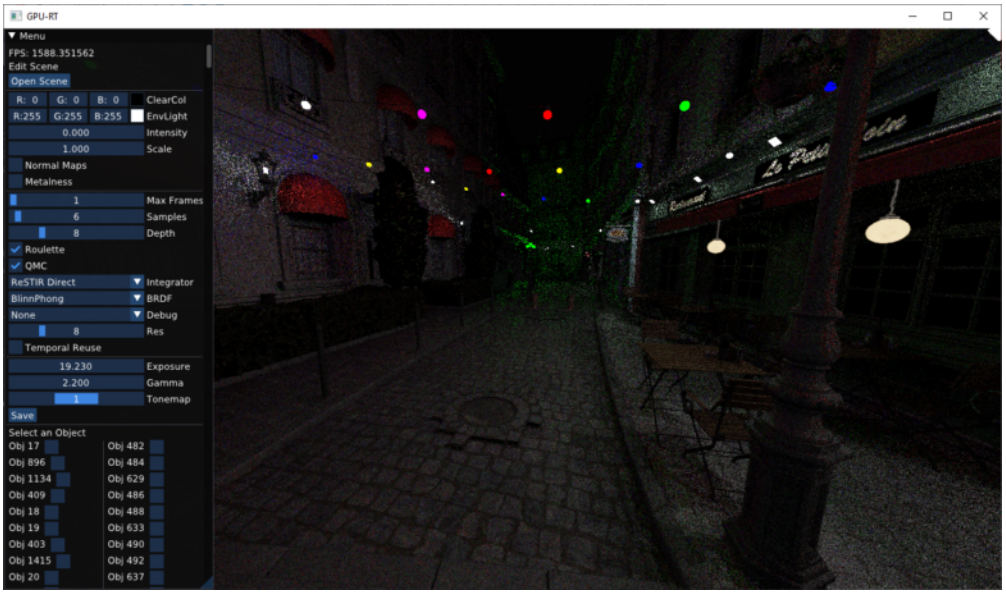

Figure 25: ReSTIR (6 SPP, 8 light samples @ 60 FPS)

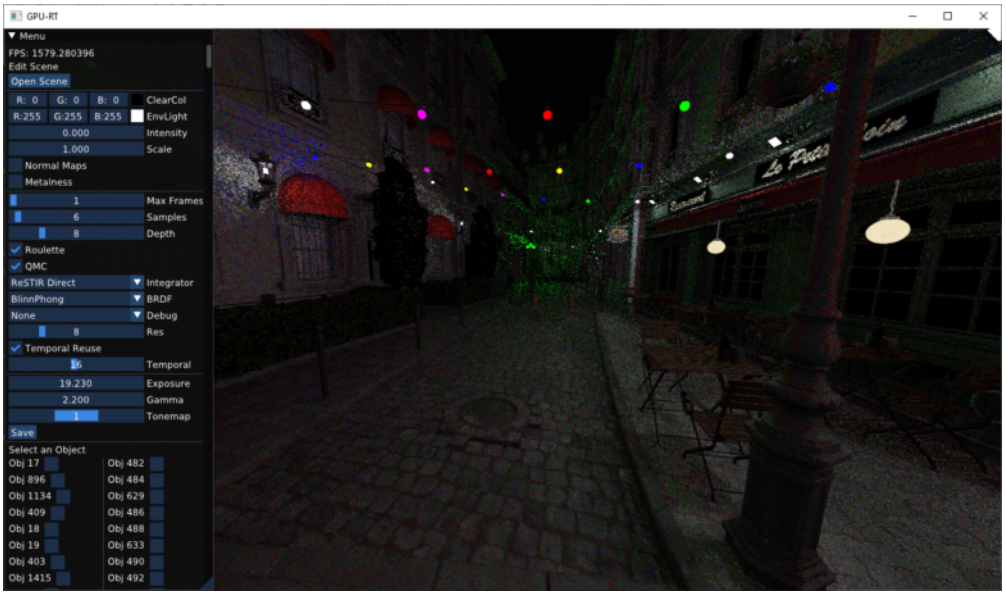

Figure 26: ReSTIR + Temporal Re-use (6 SPP, 8 light samples @ 60 FPS)

Temporal re-use improves noise across only 6 samples, but is not hugely impactful. The more important benefit is that re-using samples across frames allows us to turn down the samples per frame to 1, and still get a very similar result with a fully dynamic scene. Without temporal re-use, we have to throw out old samples when the camera moves.

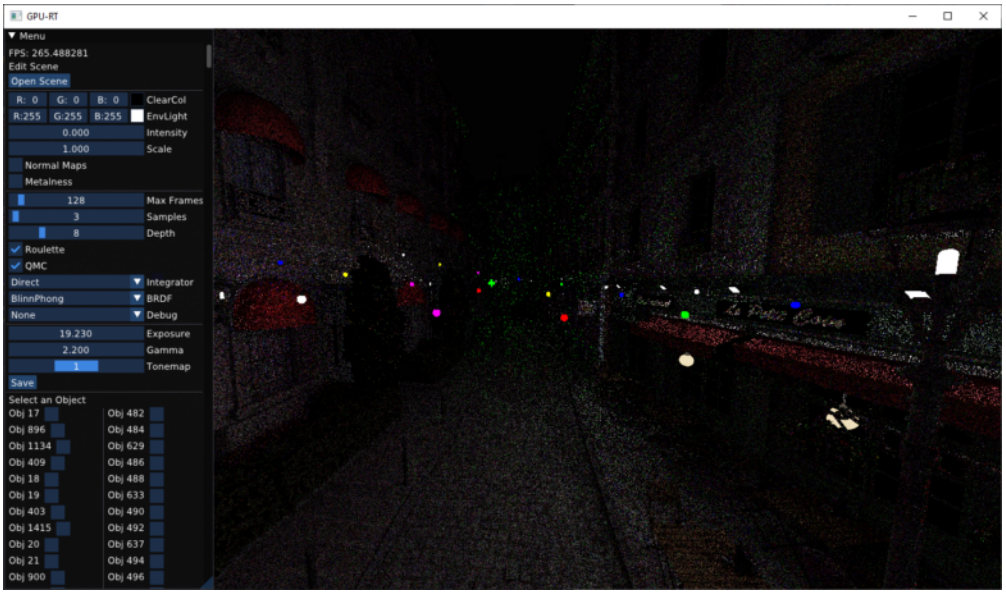

Figure 27: ReSTIR (3 SPP, 8 light samples @ 300 FPS)

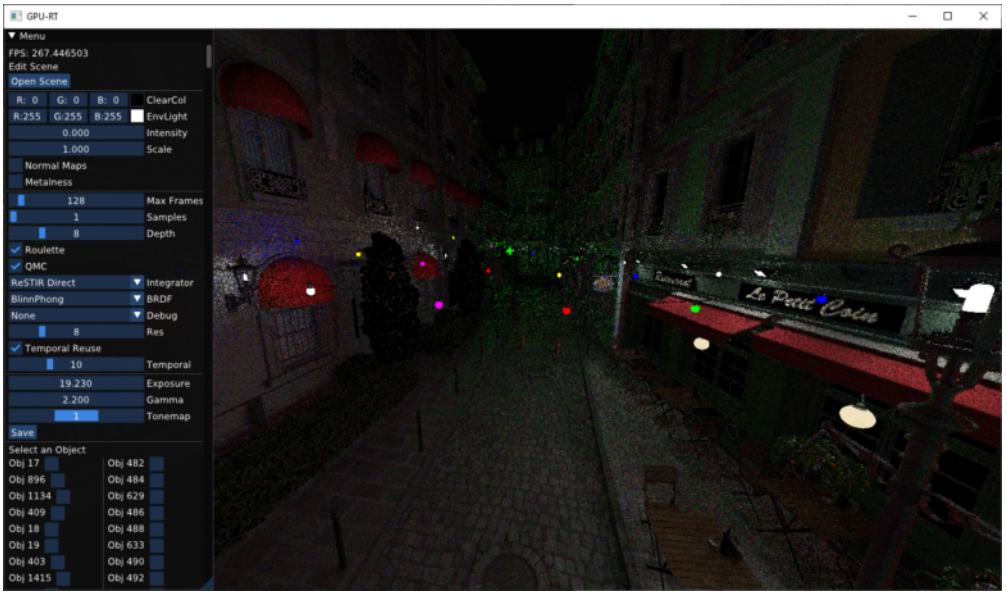

Figure 28: ReSTIR + Temporal Re-use (1 SPP, 8 light samples @ 300 FPS)

Lastly, the converged direct lighting:

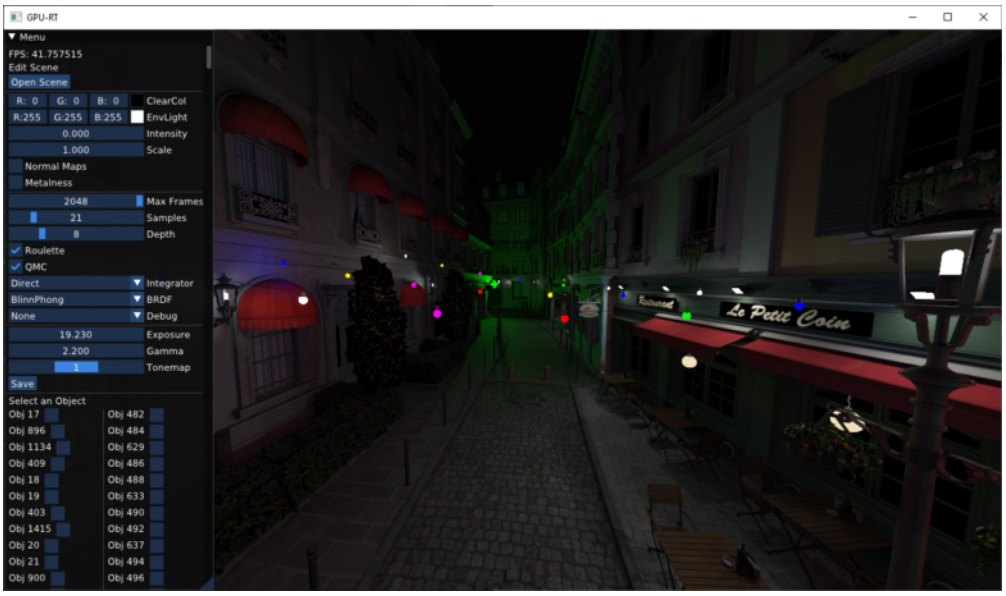

Figure 29: Uniform Direct (converged)

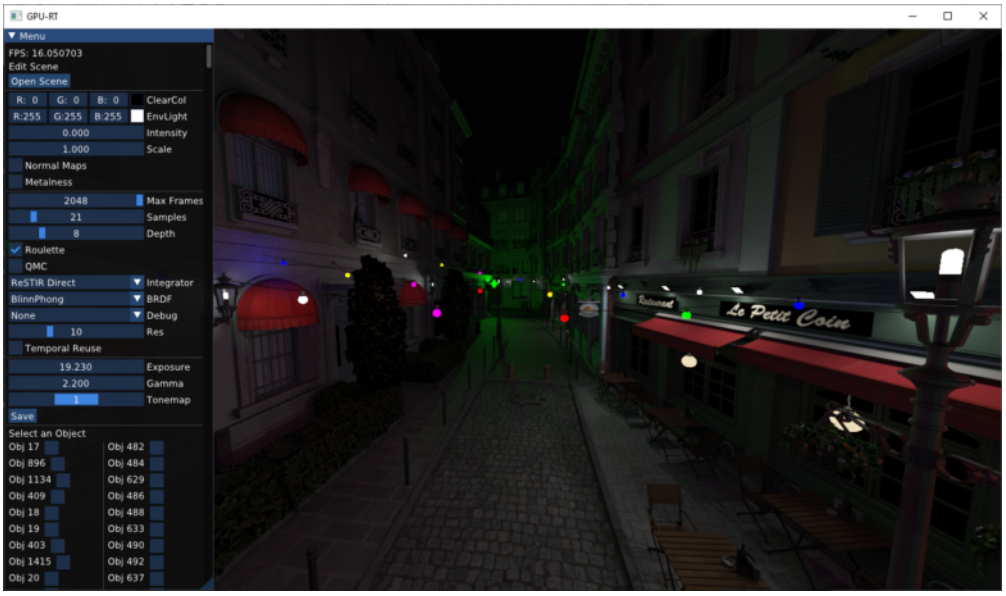

Figure 30: ReSTIR Direct (converged)

Note the slight darkening in areas with highly varying surface normals—this is the bias introduced by ReSTIR, as combining samples from different surfaces is not always energy-preserving.

3.5 Final Render

I made a scene depicting an endless 3D maze of blue-tinted mirrors holding some sort of infinite clockwork machinery. The clockwork and stairs models are found online from various sources. I arranged the geometry, specified materials/camera settings, and added lighting via a grid of emissive hexagons floating over the scene.

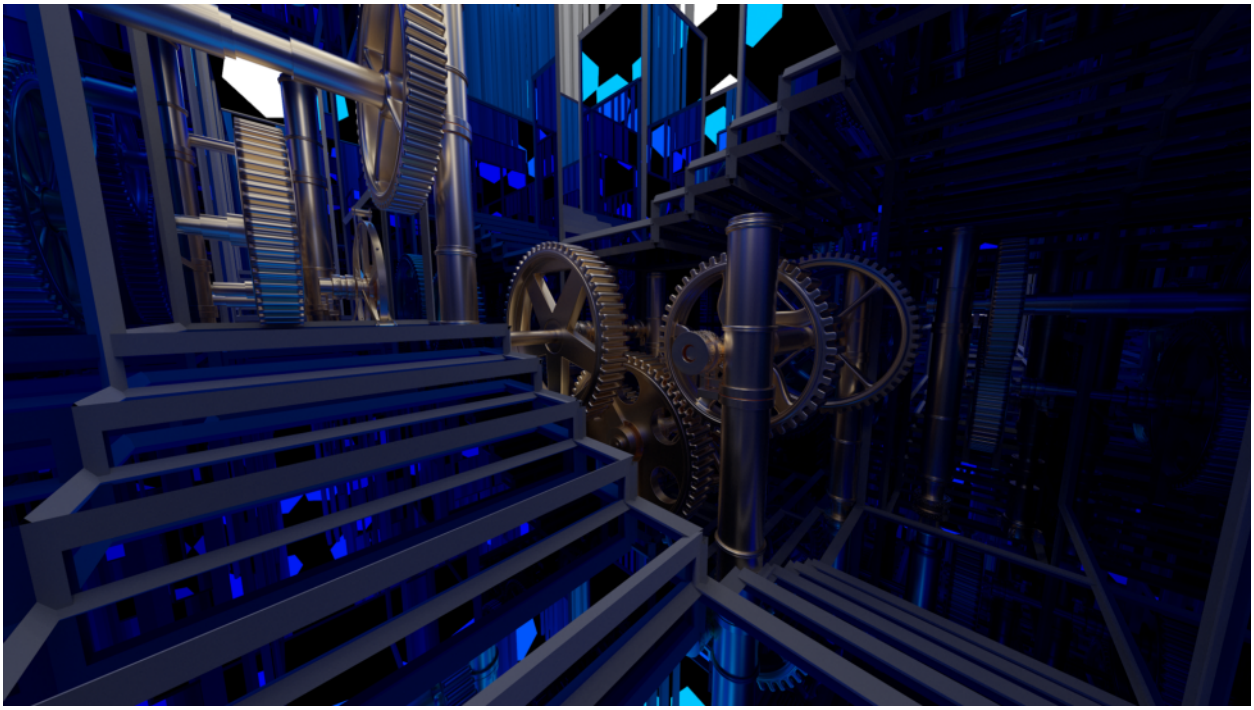

Figure 31: Final render (1080p)

3.6 Resources

Code Referenced:

1. NVIDIA Vulkan ray-tracing sample code
2. My DIRT implementation
3. ReSTIR implementation

Papers:

1. ReSTIR

Scene Geometry:

1. 3D Labyrinth
2. Clockwork Gears