# CUDA PathTracer

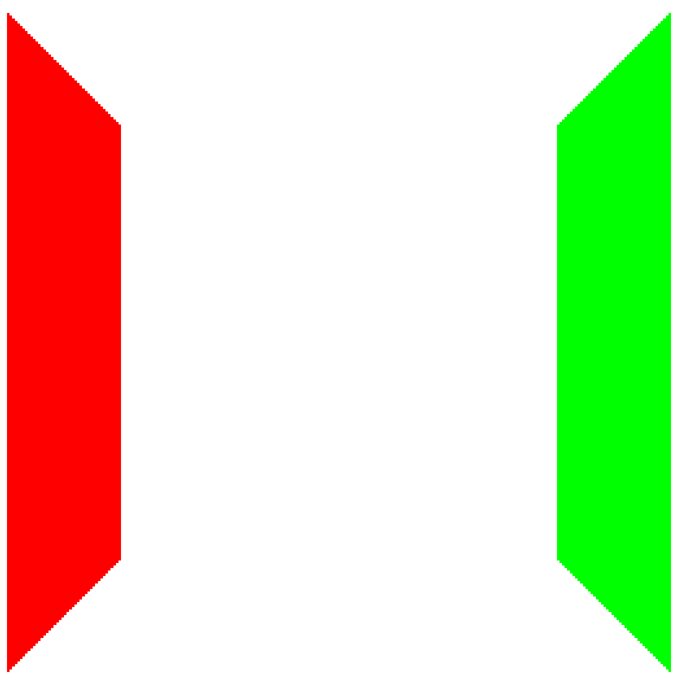
(Prototype)

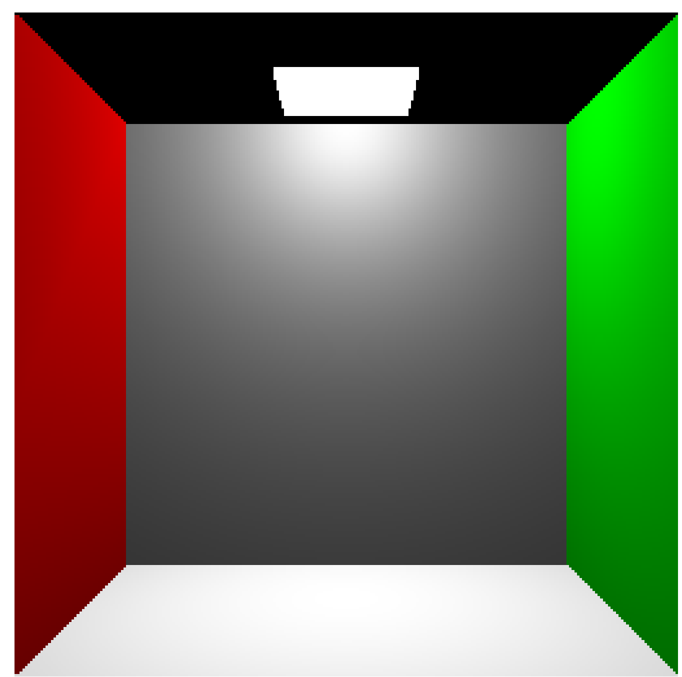
Raimond Tunnel

Study IT in .ee

# Standard Graphics Pipeline

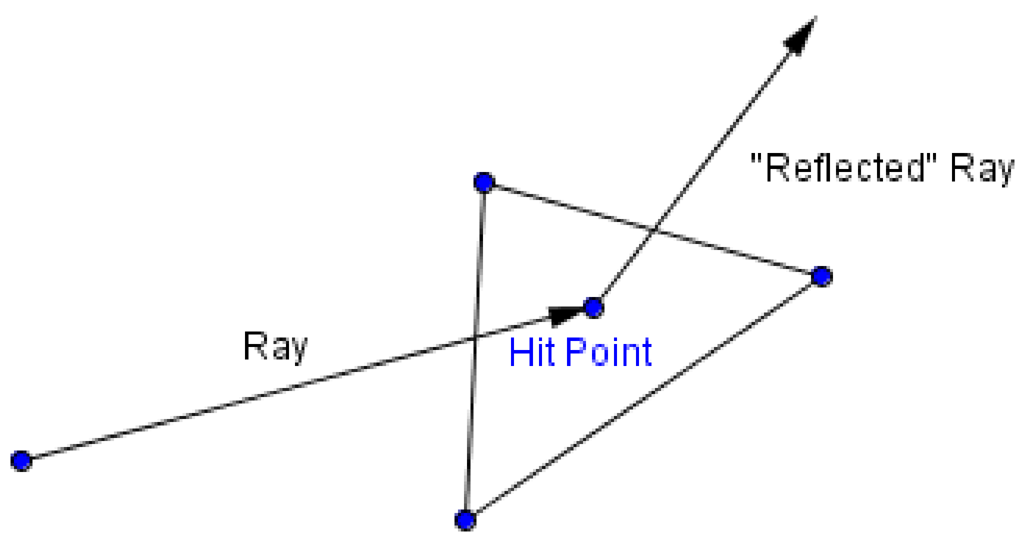
1) Transform vertices (vertex shader, GPU)
2) Rasterize the geometry (GPU)
3) Color the pixels (fragment shader, GPU)





# Ray Tracing

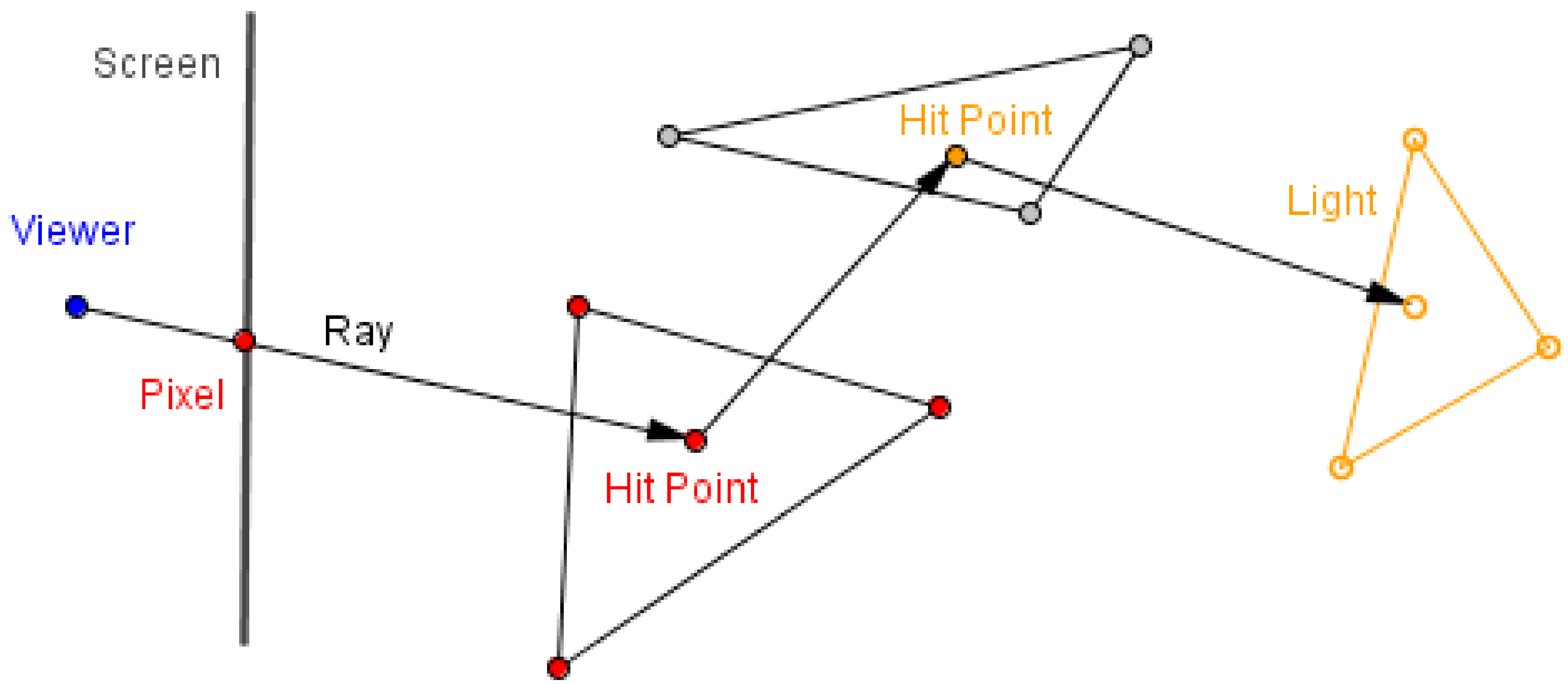
- Cast a ray (point & direction)
- Find what it hits in our scene
  - Möller-Trumbore intersection algorithm
    http://en.wikipedia.org/wiki/M%C3%B6ller%E2%80%93Trumbore_intersection_algorithm
- Cast another ray from the hitpoint (in some direction)



# Path Tracing

- Shoot rays from the viewer, through the pixels.
- Have them bounce around in the scene.
- Bounce with a random new direction.
- Accumulate the reflected light (BRDF).
- Upon hitting a light source, you know the color.
- Do this 800+ times per pixel, average results.

# Path Tracing

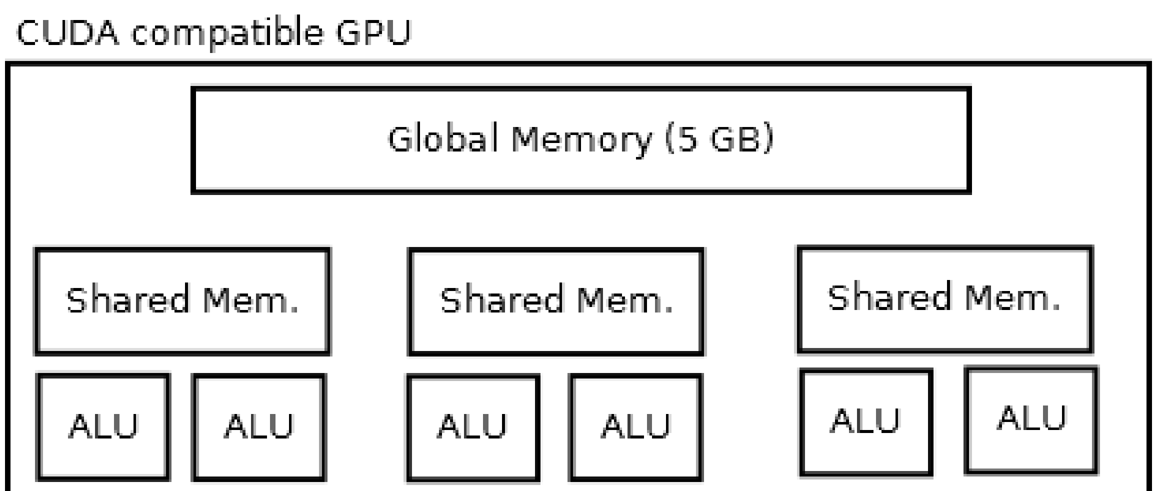


# Path Tracing

- Computationally more expensive than SGP.
- Random direction, in hopes of finding good enough approximation of light paths.
- Monte-Carlo like integration
- Can be easily parallelized, no dependancies between computations!

# CUDA

- Language for programming Nvidia GPU-s to do general purpose computing.
  (you can always program any modern GPU with shaders, but then you are restricted to SGP)
- CUDA-compatible GPU-s have a grid of blocks. Blocks have many arithmetic-logic units (ALU-s) in them and its own shared memory.




Max 1024 threads in a block.
Can queue huge number of blocks.

# My Implementation

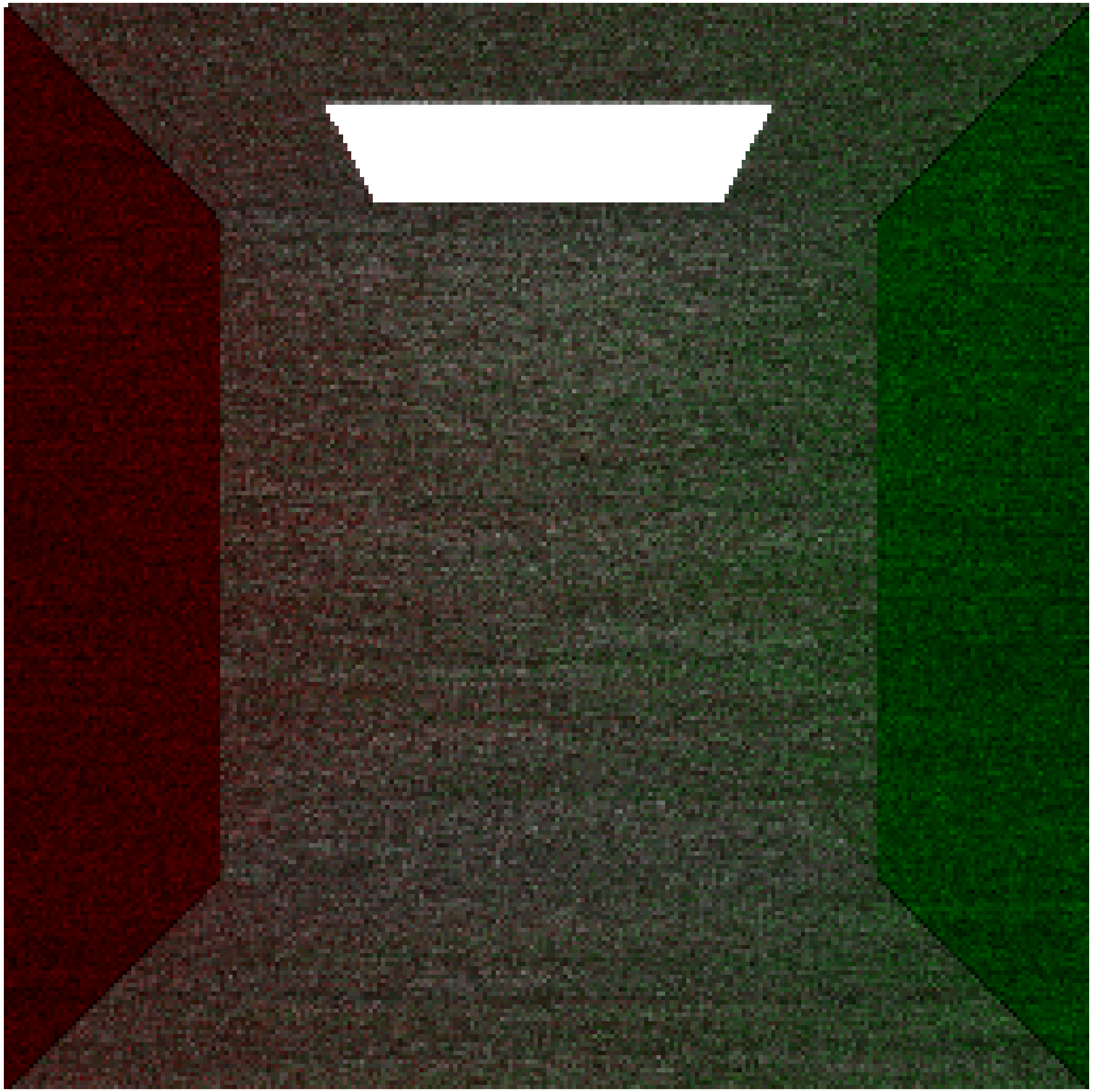
https://github.com/jee7/cuda-pathtracer

- Parallelization by rays
- Global queue of rays that need a resolution

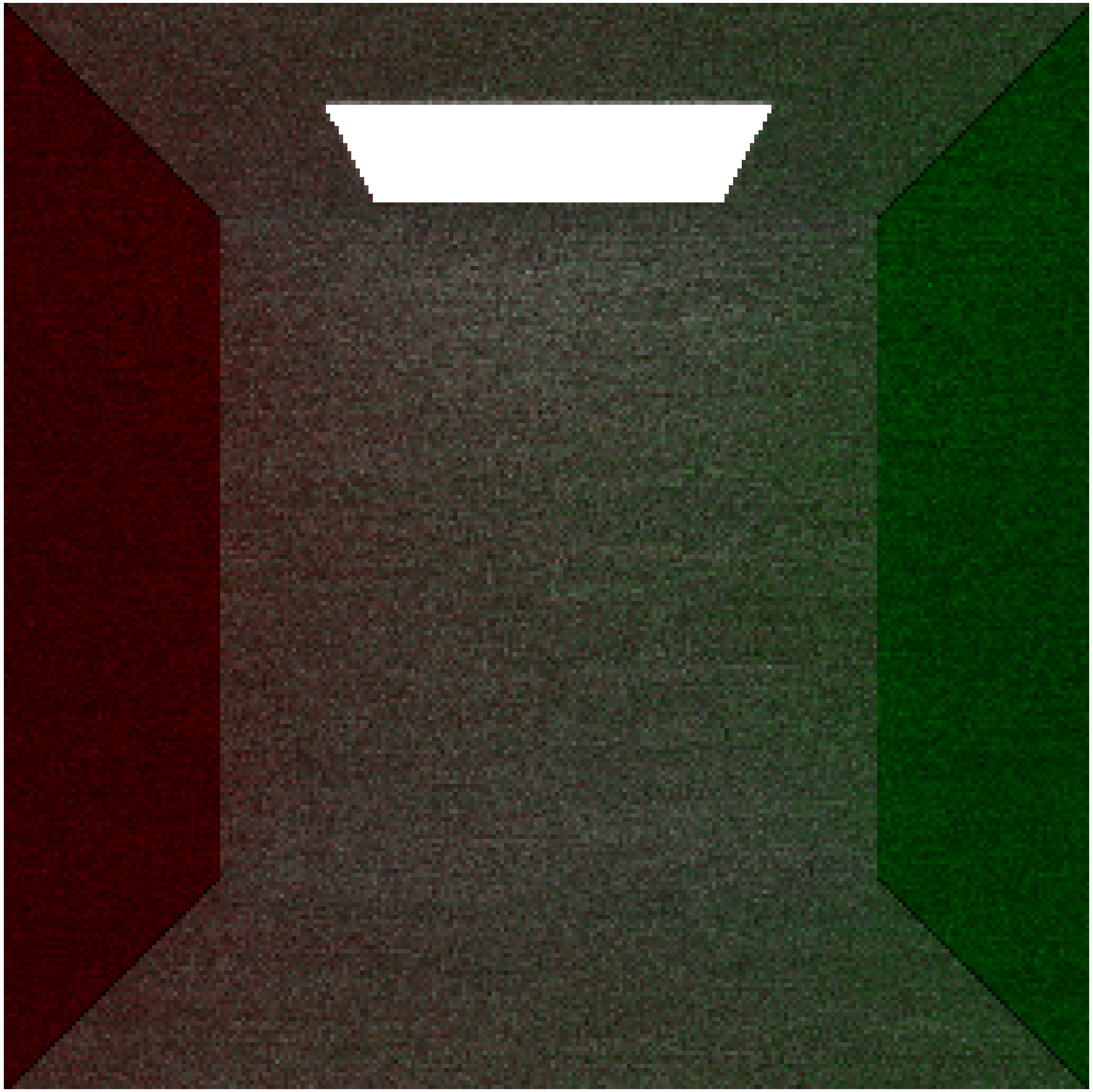
1) Generate all initial rays, put them to queue
2) Call a large number of worker threads to resolve the queue (dynamic parallelization)
3) Save results to array and collect from CPU
4) Write stuff to file, use JavaScript to see the pic.

Because of the EENet cluster. No OpenGL or any other display.
Maybe VirtualGL?

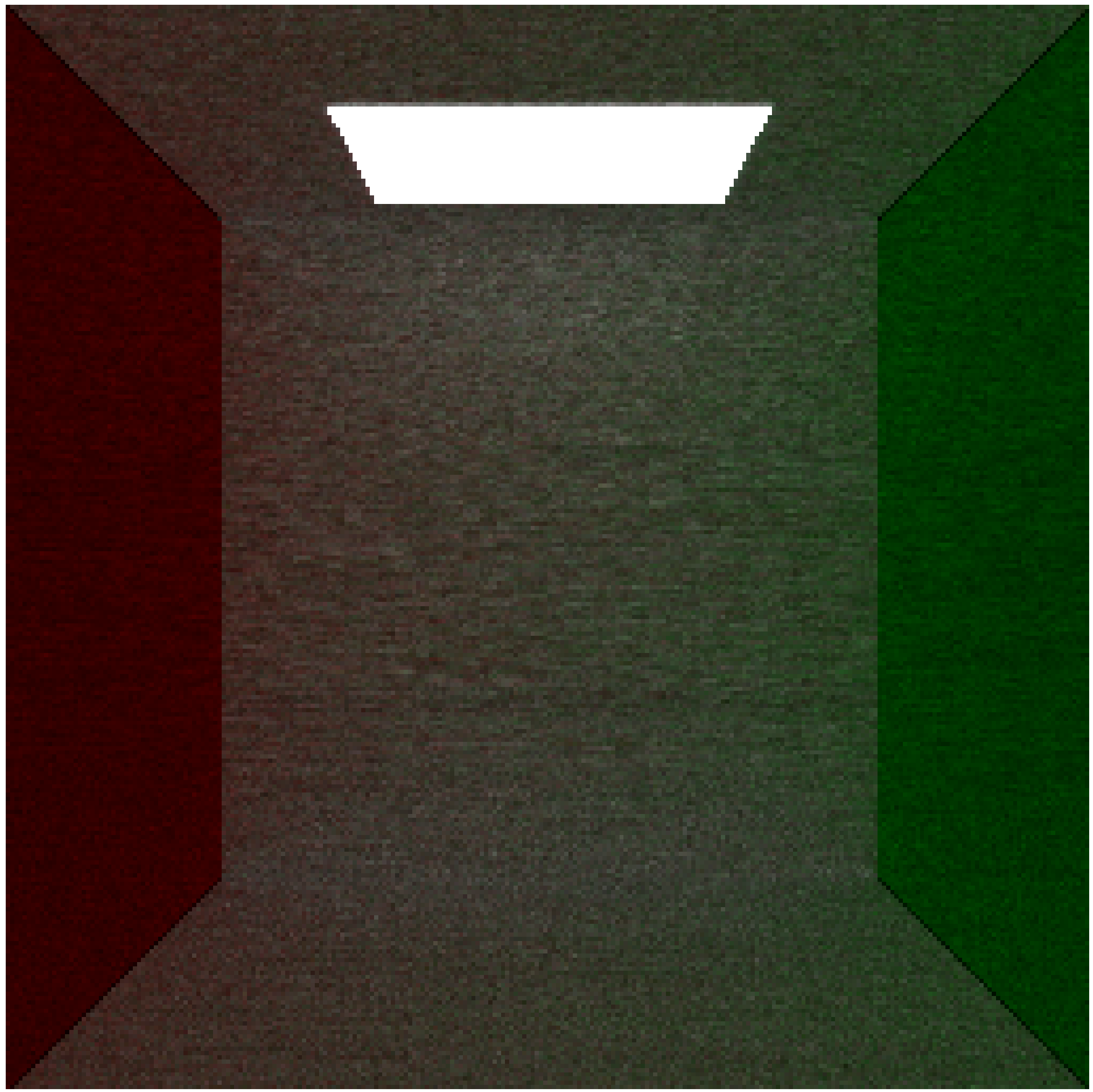
# Results (256×256, 20 iter.)



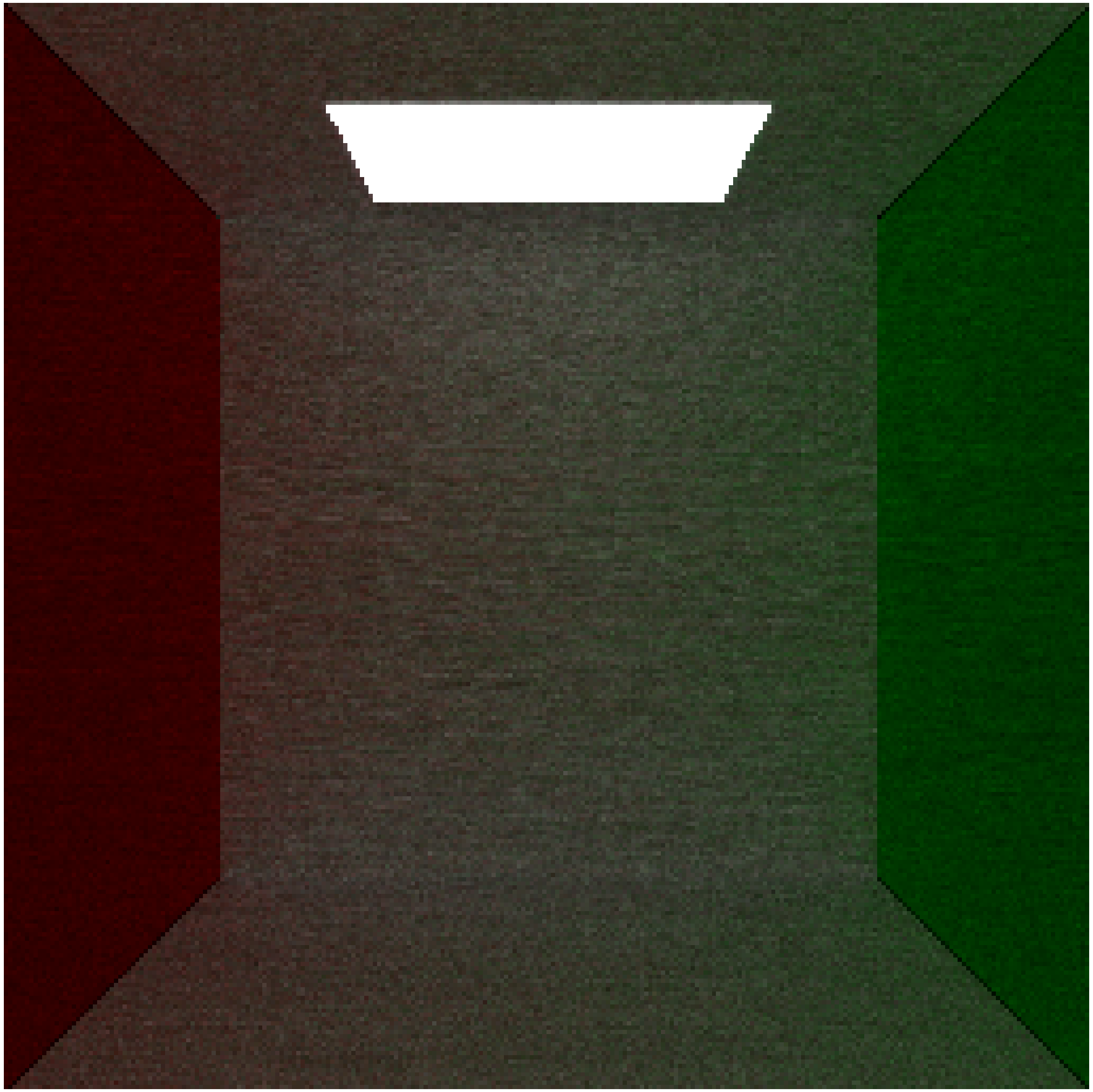
# Results (256×256, 40 iter.)



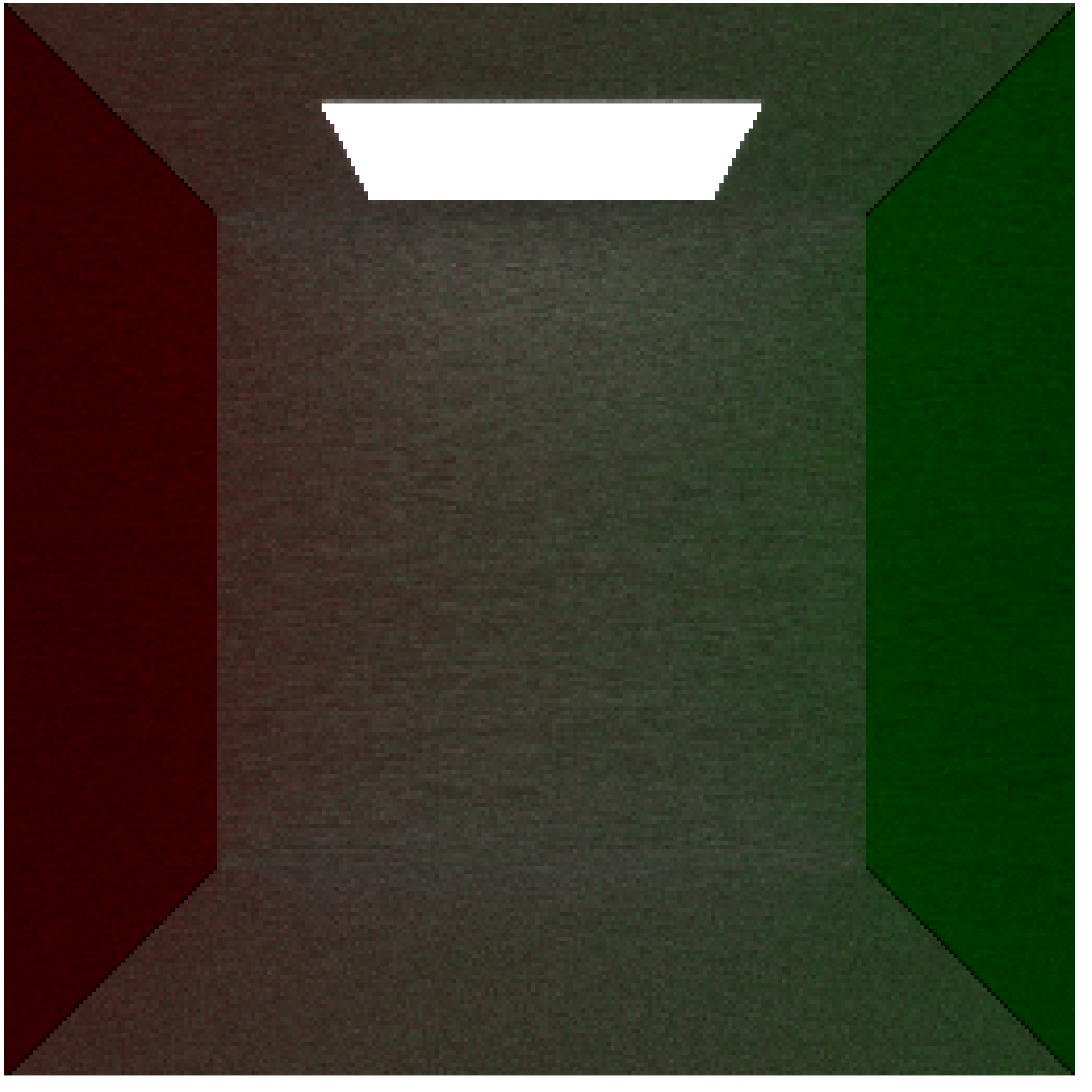
# Results (256×256, 80 iter.)



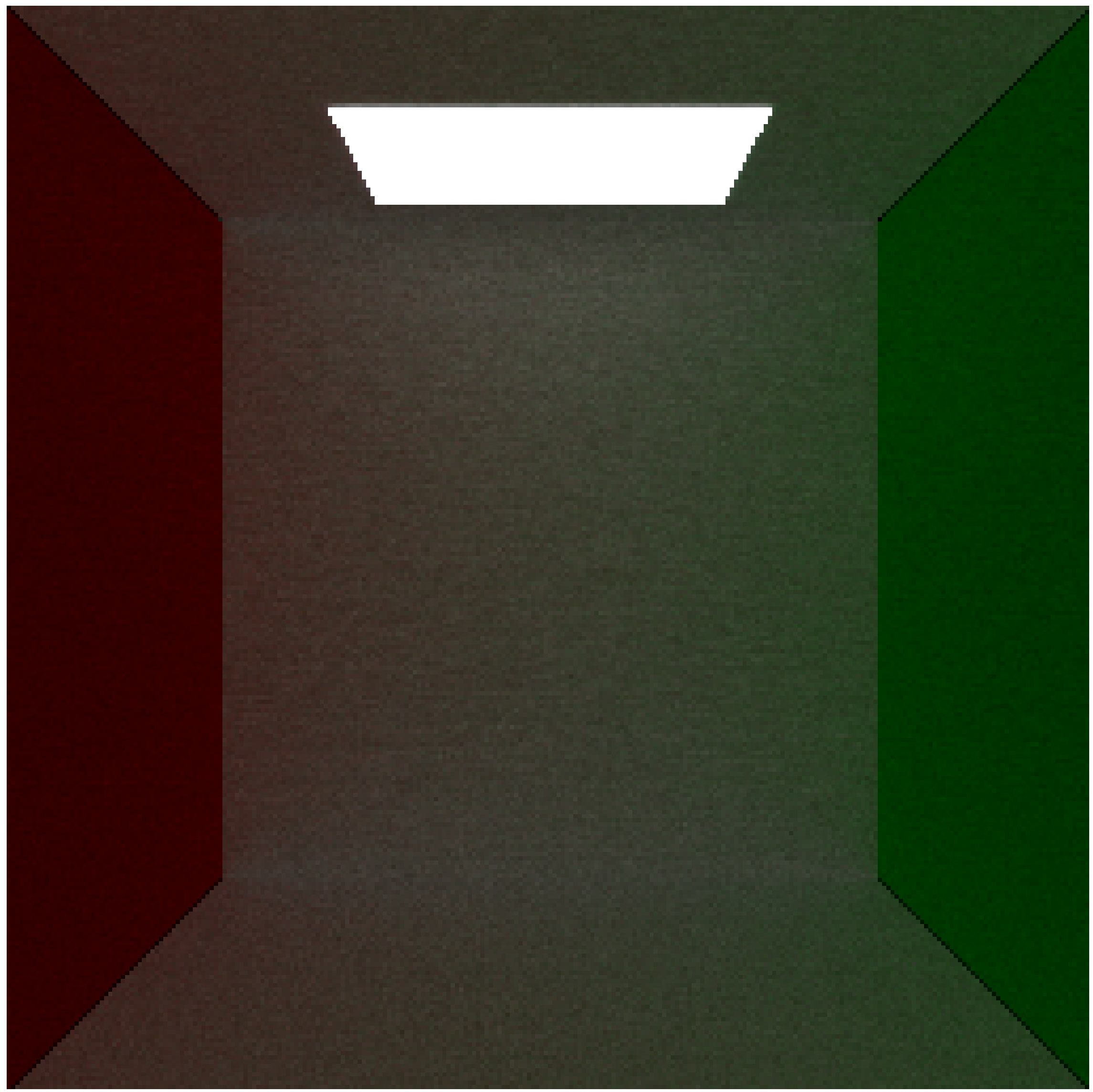
# Results (256×256, 100 iter.)



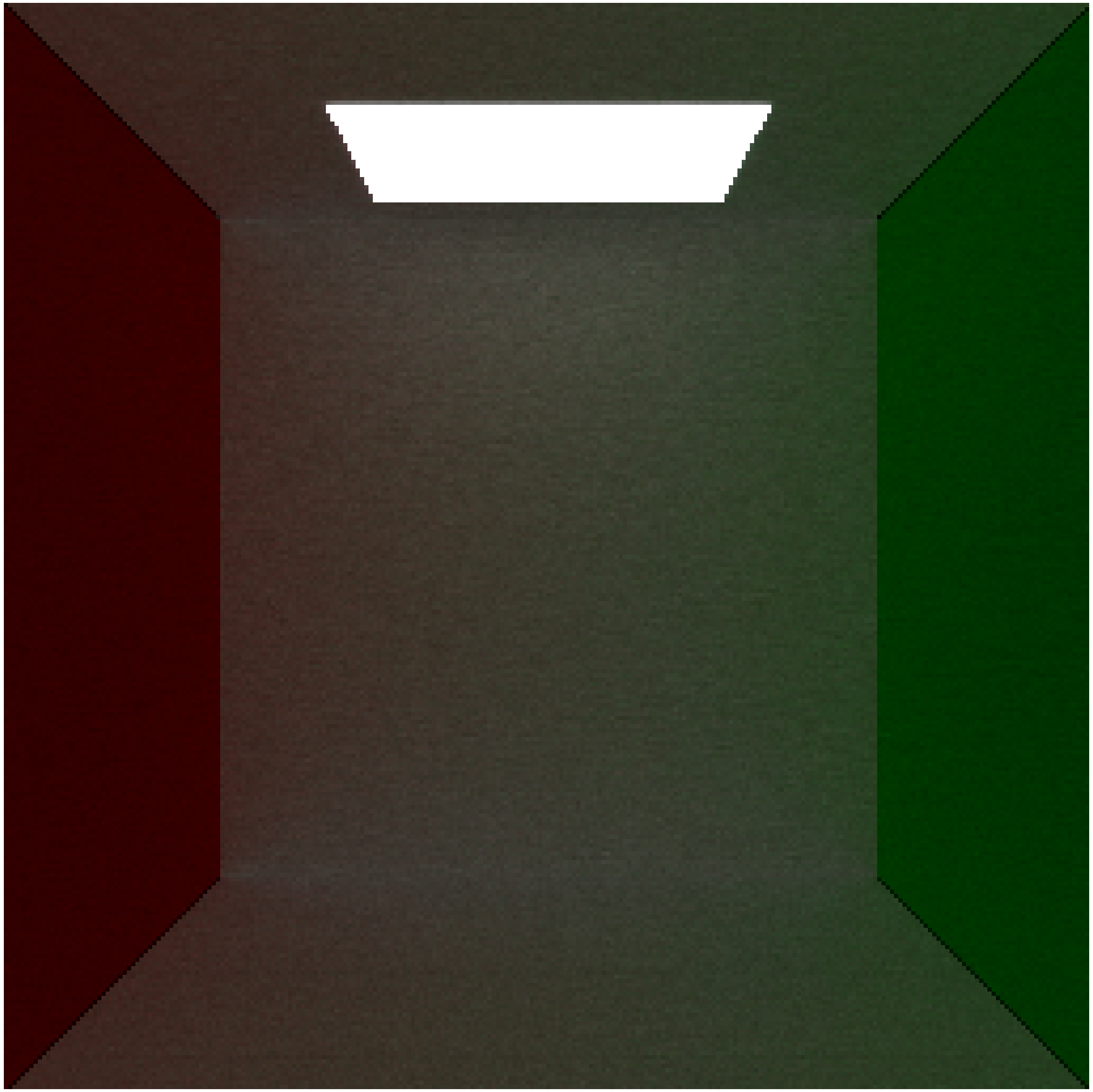
# Results (256×256, 200 iter.)



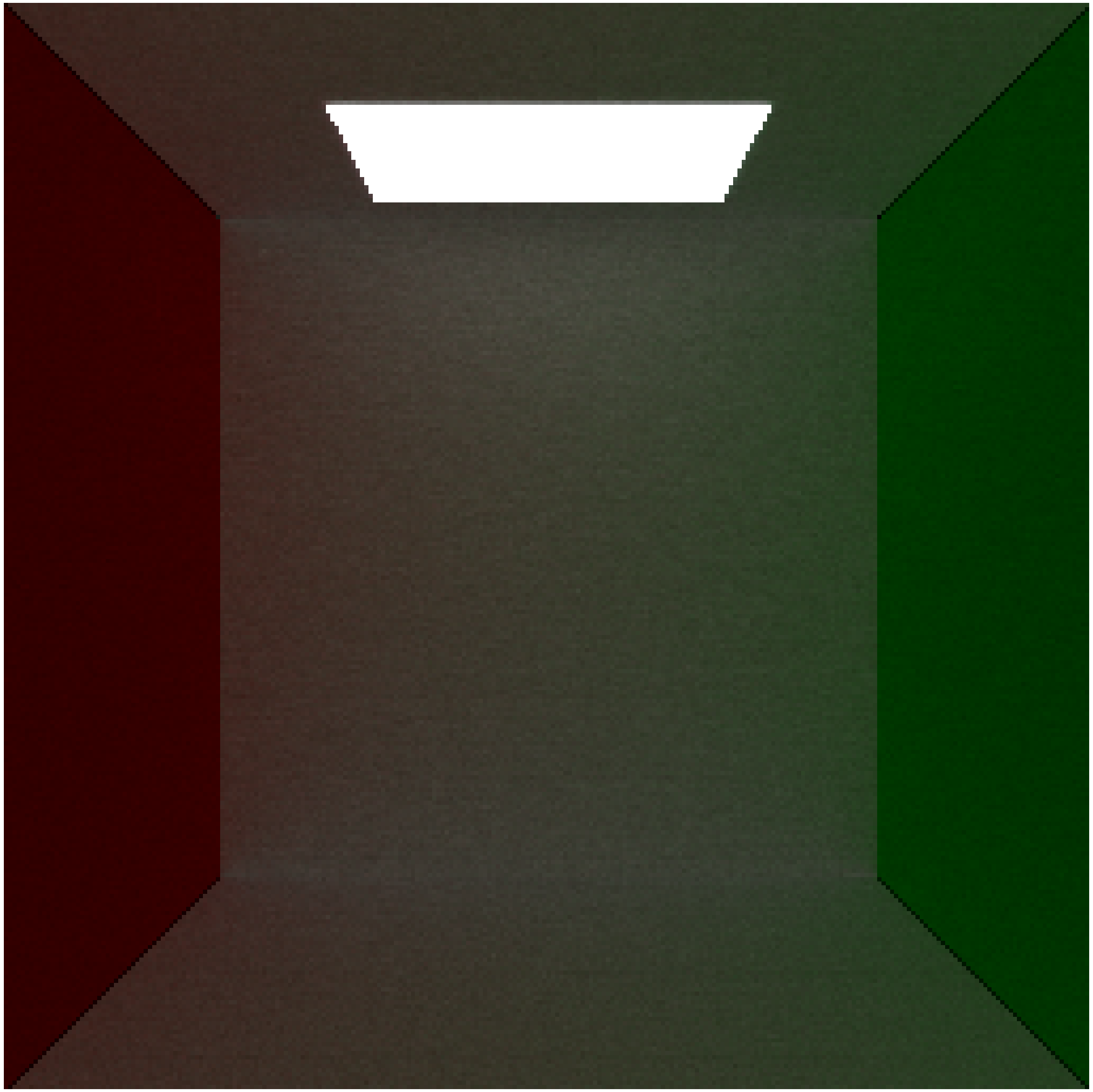
# Results (256×256, 400 iter.)



# Results (256×256, 600 iter.)



# Results (256×256, 800 iter.)



Notice the global illumination...

# Future Ideas

- Total global queue takes a lot or memory, resolve rays via batches.
- Utilize shared memory, can probably handle 1000 rays in there.
- Try out different other approaches besides the random ray bounce.

That is all... thanks!