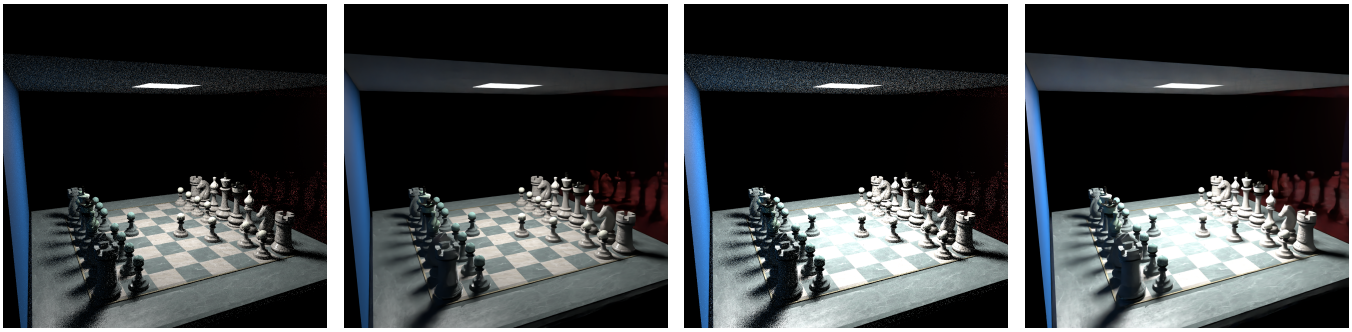
CUDA-Accelerated Neural Pathtracing

Shashank Anand
shashank.anand@eecs.berkeley.edu
UC Berkeley
Berkeley, CA, USA

Yongye Zhu
yongye.zhu@berkeley.edu
UC Berkeley
Berkeley, CA, USA

Yifei Dai
yifeidai@berkeley.edu
UC Berkeley
Berkeley, CA, USA

Seyoung Samuel Kim
sam2025@berkeley.edu
UC Berkeley
Berkeley, CA, USA



ReSTIR GI raw dim | **ReSTIR GI denoised dim** | **ReSTIR GI raw bright** | **ReSTIR GI denoised bright**

Figure 1: Dim and bright renderings of CornellChess at 1200x1200 resolution.

Abstract

We present a fully-featured, CUDA-accelerated, one-sample-per-pixel path tracer that combines ReSTIR GI's spatio-temporal reservoir resampling with OIDN GPU-accelerated denoising to generate stable, noise-free indirect illumination and reflections in complex glTF scenes. We implement a lightweight GLTF loader, and support the complete base GLTF feature set, in addition to a number of extensions. We utilize the PBR rendering model, supporting textures, normal maps, roughness maps, metallic maps, and emission maps. We also support rendering multiple frames at different camera angles in one go, taking full advantage of the temporal resampling step. On an RTX 2000 GPU, we render a single 400x400 frame of a scene with 1.5 million primitives simulating 10 bounces of light, in 30ms, producing ≈ 33 frames per second. We render a high fidelity 1200x1200 frame in approximately 200ms, achieving near-realistic quality.

The website for our work is at: https://cal-cs184-student.github.io/hw-webpages-fpsg-webpage/final_report/index.html

1 Introduction

Graphics as we know them would be impossible without GPUs: they are ubiquitous in accelerating real-time rendering, offering exponential speedup over their CPU counterparts. The NVIDIA RTX 4090 [10] achieves 82.6 TFLOPS of FP32 throughput and a 443.5 GPixel/s rasterization rate. GPUs enabling real-time rendering of millions of triangles at 60 fps or the ray tracing of billions of rays per second. However GPU hardware designs, firmware, and drivers remain almost entirely closed-source and proprietary, severely limiting transparency into their low-level behavior. The advent of RISC-V and Chipyard [2] have led to a positive shift in research with focus on open source hardware and realistic full-system FPGA accelerated evaluation. Open source GPU designs are essential for academic research on novel graphics algorithms and performance optimizations.

Radiance, a proposed open source heterogeneous GPU, is currently under development at the SLICE lab. A key feature of the GPU is decoupled accelerator orchestration: allowing special purpose units such as ray tracing cores and matrix multiplication units to be decoupled from the SIMT core, thereby allowing them to operate asynchronously while freeing up the SIMT cores to handle other instructions. The Radiance programming model enables programmers to interface with these decoupled accelerators. Radiance is inspired by Vortex [14], a RISC-V open source GP-GPU designed for FPGAs.

Latency from raytracing at high samples-per-pixel (SPP) is infeasible for real-time applications, and real-time raytracing is achieved by rendering at very low SPP and denoising the resulting image [6, 11]. This paradigm shift is observed in both open source and enterprise rendering engines: blender supports low SPP rendering and denoising and NVIDIA offers denoising capabilities in the OptiX framework [4]. ReSTIR GI [9] is a novel method for rendering better images at very low SPP (1 spp), improving the quality of the denoised image. ReSTIR GI is also supported in Unreal Engine 5 [5].

Conference acronym 'XX, Woodstock, NY
2018. ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

Intel Open Image Denoise (OIDN) [1] is an open source denoising library, with pre-trained networks. OIDN is one of the supported denoising backends in blender.

The goal of the overarching project is to build a state of the art end-to-end neural raytracing pipeline to evaluate our Radiance GPU. In this paper, we describe our CUDA implementation of this neural raytracing pipeline. Future work will build the RISC-V and Radiance ports of this code.

2 Background

2.1 CUDA

CUDA (Compute Unified Device Architecture) is NVIDIA's parallel-computing platform and programming model introduced to expose the massive data-parallel power of modern GPUs. It extends standard C/C++ with minimal language additions. Thousands of light-weight threads are organized into thread blocks, which in turn form a grid. At hardware level, threads are scheduled in warps (typically 32 threads) that execute in lock-step under the single-instruction, multiple-thread (SIMT) paradigm. Each instruction in a CUDA kernel is executed once by each thread in parallel, where threads are identified by thread specific registers such as the thread idx. As all threads execute the same instruction, branches cause thread divergence where execution is serialized among groups of threads that take the same branch, incurring a severe performance penalty. Therefore it is imperative to optimize kernels to minimize thread divergence.

2.2 Denoising

Denoising in path tracing is the process of improving the fidelity of low SPP renders by utilizing a neural network. Denoisers can integrated with the path tracer in interesting ways. The naive method is to launch separate kernels for the denoiser, utilizing the final output of the path tracer. It is also possible to tightly couple the denoiser with the path tracer by fusing their kernels, but this requires careful synchronization across dependent pixels.

2.3 ReSTIR

Reservoir-based spatiotemporal importance resampling (ReSTIR) [3] is a Monte Carlo framework that maintains a small "reservoir" of candidate lighting samples per pixel and iteratively updates it via weighted reservoir sampling to approximate the rendering integral with far fewer evaluations than traditional path tracing. The algorithm is built upon Resampled Importance Sampling [13].

2.4 ReSTIR GI

ReSTIR for Global Illumination [9] extends the ReSTIR algorithm to multi-bounce path tracing by reusing indirect lighting samples across both space and time, attempting to improve the quality of computed multi bounce radiance by reusing neighbouring samples, and samples from the previous frames.

2.5 glTF

glTF (GL Transmission Format) is an open, royalty-free specification by the Khronos Group designed for efficient transmission and loading of 3D scenes and models. It represents assets in JSON (.gltf) with optional external or embedded binary buffers (geometry, animations, skinning) and textures, or as a single binary package (.glb). glTF scenes express materials as a linear combination of a dielectric and a specular term (which in itself are computed using a combination several factors such as metallicity, roughness, etc), enabling the rendering of realistic effects.

3 Scope & Challenges

The goal of the overraching project is to build an end-to-end neural graphics evaluation framework for the Radiance GPU. This requires the following milestones:

(1) CUDA implementation of ReSTIR GI
(2) CUDA implementation of OIDN
(3) RISC-V implementations of ReSTIR GI and OIDN

item 1 and item 2 are essential to allow us to have a baseline for application study, verification and evaluation. Having a baseline CUDA implementation greatly simplifies the process of porting to RISC-V. It also enables us to identify points of optimization using the decoupled accelerator orchestration mechanism of Radiance. Additionally, the ISA of radiance is a modified version of the RISC-V ISA for which we have built a compiler. Luckily, item 2 is available out of the box with the OIDN library. Therefore, the scope of this project is to implement item 1, while future work will tackle item 3.

Keeping these considerations, we devised the following requirements for our pathtracer:

(1) Low latency low SPP rendering followed by denoising
(2) Lightweight scene loading and support for complex scenes
(3) Highly simplified code base, with static linkage and minimal external library dependencies
(4) Packed, simple C style structs to represent the scene to interface with CUDA

Keeping these in mind, we began on the HW3 pathtracer framework, considering that it had minimal dependencies, and all dependencies were bundled with the project. However, we quickly found out that this is inadequate for our purposes.

The hw3 framework extensively uses polymorphic types. This does not work with CUDA, as CUDA kernels require the size of all data types to be known at compile time. Additionally, vtables are not copied over to the GPU side. Even if we were to copy over vtables, this would kill our performance as we would do additional memory accesses.

After extensive debugging and experimentation, we also came to realize that the Collada parser bundled with the provided framework is incomplete. Collada files exported with blender contain several tags, such as the triangles tag used to represent triangulated meshes, that are not implemented by the bundled Collada parser. This makes it impossible to handle complex scenes with varying geometry and diverse BSDFs.

Finally the GUI, half edge mesh structure, and several Collada helpers add a lot of functionality that is irrelevant for our use case of rendering images via the command line.

As a result of having to handle glTF scenes completely from scratch, we had to modify the scope of our project, and focus on getting a working and performant pathtracer, leaving the RISC-V implementation for the future.

4 Path Tracing

In this section we describe our journey. In chronological order each section describes the additions, modifications and deletions made to the path tracer. By the end of the section, we will have rewritten and reorganized almost every single aspect of the path tracer. A modern day ship of Theseus. Theseus' path tracer.

4.1 ReSTIR GI CPU

We implement the algorithm described in the ReSTIR paper first on CPU. ReSTIRGI aims to approximate the indirect lighting of each pixel more accurately. The algorithm is implemented as follows, per pixel and per frame.

(1) Create `initialSample` and `spatialBuffer`. Reuse `temporalBuffer` from previous frame
(2) Populate initial samples by raytracing one sample per pixel. We store the direct lighting (zero + one bounce) separately of the indirect lighting, as the following steps will sample neighboring values based on its indirect lighting component. We additionally also store the $fcos\theta$ value.
(3) Update `temporalBuffer` with current sample from indirect light part.
(4) Pick a number of neighbours randomly from a radius around the current sample and resample those `temporalBuffers` into the `spatialBuffer` provided they are not too geometrically dissimilar.
(5) Calculate the final radiance value according to:
`initialSample.direct+`
`spatialBuffer.fcos * spatialBuffer.indirect`

4.2 CUDA Conversion

Here, we convert all of our code to CUDA. This was easily the most difficult technical component, requiring a near total rewrite of the path tracer. Due to the sheer scale of conversion, changes were made in modules: a part of the code was identified, replaced, verified, and then we move on to the next part. A single big rewrite would have almost certainly made it impossible to debug issues. As a result, initial versions had two data types for primitives, lights and bsdfs: a host version and a CUDA version (prefixed with Cuda). For legacy and sentimental reasons we kept the CUDA name even though the host versions were eventually removed.

4.3 De-polymorphing

CUDA does not support polymorphic types, as the size of each type (concretely, the memory layout) must be known at compile time. vtables are not copied over to the GPU side. Several rewrites over the progress of this project converged the data types into `CudaPrimitive`, `CudaLight`, `CudaBSDF` and `CudaTexture`. A consequence of this is that the CPU version of the path tracer improved performance by over 2x. This is because resolving a function call to a polymorphic type requires looking up the vtable to find the relevant concrete function to call. Eliminating this lookup does wonders for performance.

```
struct CudaPrimitive {

  DEVICE bool intersect(Ray& r, CudaIntersection* i, Vector3D* vertices,
                        Vector3D* normals, Vector2D* texcoords, Vector4D* tangets) const;
  DEVICE bool has_intersect(Ray &r, const Vector3D * vertices, float &t) const;

  uint32_t i_p1, i_p2, i_p3;
  uint32_t i_n1, i_n2, i_n3;
  uint32_t i_uv1, i_uv2, i_uv3;

  int bsdf_idx;
};
```

Figure 2: Packed Cuda Primitive. GLTF only supports triangles.

4.4 RNG

Due to several difficulties with cuRand, CUDA's random number generator (strange banding at fp32, build system integration issues), we decided to implement our own random number generator (with copious assistance from AI). Our random state utilizes two 64 bit integers, with a large enough state space that correlation across threads is practically impossible. However 128 bits of state per thread is quite heavy, and this must be improved in the future (smarter cuRand usage and integration).

4.5 Data Movement

All scene data built has to be copied over to the GPU. This can be annoying if you have several composed objects: you need to cudaMalloc and cudaMemcpy with a temporary host-side container for the parent class, and then recursively repeat the process for the parent's parent. We flattened the structure as much as possible, with our deepest object being the BVH struct (2-level) which contained pointers to primitives and their vertices, normals and tangents.

4.6 Kernel Launch

Now, we split the functions into two kernels: `raytrace_temporal` and `spatial_sample`. Because spatial sample depends on neighbouring pixels being completed, we need to split it into a separate kernel. In the future we will investigate ways to perform smart kernel fusion. We generate grid and block dimensions (one pixel per thread), and launch the kernels appropriately.

4.7 glTF Loading

Due to the limitation of the Collada parser in the original path tracer to handle multiple meshes in a single node, we implement a custom integration of glTF parser to handle complex materials and meshes. Concretely, we use the existing glTF library, tinyGLTF [12] to parse a gltf file into a `Model` class. A `Model` contains a global array of vertices and an array of triangle meshes as well as all the texture and material data associated with the node, which we then use to construct our internal `CudaPrimitive` struct.

4.8 Texture Mapping

We support the full range of maps as defined in the glTF spec. We implement base color texture maps, normal maps, ORM (occlusion, roughness, metallic) maps and emission maps. These advanced maps allow us to model complex and irregular surface geometry. All textures are stored in a texture-type agnostic structure. Textures

are indexed and interpolated identically: at each intersection, compute uv coordinates, interpolate via barycentric coordinates, query texture. Textures are great because they let you approximate high triangle meshes with fewer triangles and accompanying textures.

4.8.1 Base Color Textures. These are the diffuse colors of each mesh, for example a chess board pattern. They are 4 channel RGBA textures.

4.8.2 Normal Textures. The RGB coordinates gives the x, y, z coordinates of the normal in tangent space. Tangent space is the 3D coordinate space described by the tangent, the bitangent and the normal vector. The tangent vectors are provided in the glTF file. We use the interpolated normal and tangent at the point of intersection to compute the TBN matrix, and transform the interpolated normal texture to obtain the "true" normal at that point.

4.8.3 ORM Textures. Occlusion, Roughness, Metallic are respectively contained in the R, G and B channels. These are used to vary the material's roughness and shinyness across a mesh.

4.8.4 Emisison Textures. These let you specify specific emissive regions on a mesh, for example led lights on electronics. The R, G, B values gives the strength and color of the emission.

4.9 PBR Material Modeling

Finally, we implement the PBR model for computing the f value of a point. We defer to the glTF spec [7], we implemented their equations almost one to one.

4.9.1 MIS. Now that we have real material modelling, we proceed to implement Multiple Importance Sampling. Instead of simply importance sampling lights, we also sample in the direction of the BSDF, each of those contributions weighed by their MIS weights.

$$W_l = \frac{pdf_l}{pdf_l \cdot pdf_{bsdf}} W_{bsdf} = \frac{pdf_{bsdf}}{pdf_l \cdot pdf_{bsdf}} \tag{1}$$

Where W_l and W_{bsdf} are the mixing weights of sampling lights and sampling bsdfs respectively. Sampling BSDFs is essential in cases of metallic BSDFs. This is because BSDF contributions are skewed in a narrow cone away from the angle of incidence, which we are unlikely to sample by purely importance sampling lights. To sample a BSDF, we randomly sample from either its dielectric or metallic component based on its metallic value. Figure 3 shows the appearance of strong specular highlights after implementing MIS.

At this point, we have implemented a full feature GPU accelerated rendering engine. Performance degrades quite severeley with the implementation of advanced BSDF sampling and textures. This is because advanced BSDF sampling causes a significant amount of thread divergence as sampling branch changes unpredictably based on hit material. Additionally texture sampling adds upto 4 memory accesses per material for each texture that it has. It also significantly increases the amount of scene data stores on the GPU, due to the texture images, tangents, etc, probably causing frequent cache misses. These are directions to optimize along in the future.

Figure 3: Top: Naive sampling. Bottom: Importance sampling. A significant difference in indirect lighting can be observed. Note that these are early stage low-res buggy renders.

4.10 Video generation

To implement video generation, the key is to implement camera movement inside path tracer to follow a directed path. For simplicity, we only implement the camera to move around a target in one horizontal circle. We also extend our path tracer to accept an additional argument to generate a designated number of images. For example, if the path tracer is asked to generate 100 images, it will generate these images around a pre-programmed circle. Internally, after we generate each images, we move the camera by $2\pi/num\_images$ in θ direction. Note that since the camera object lives inside the CPU memory when we modify it, we need to copy all the camera data back to GPU. For each image generation, we clear the image frame buffer and spatial reservoir buffer, and reuse the temporal sampling from the previous frame, which lets us sample across both time and space.

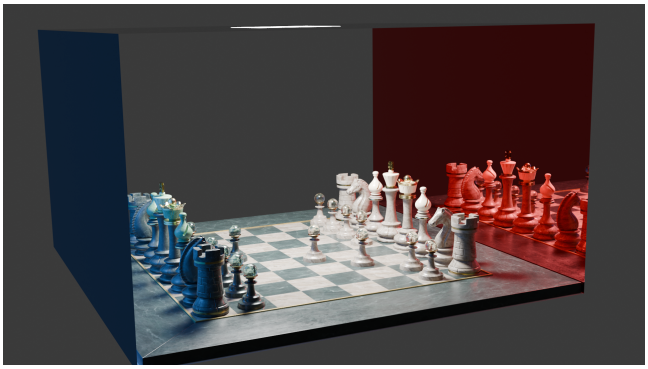


Figure 4: Blender rendering of CornellChess. Render at 1024 spp took 21 seconds.

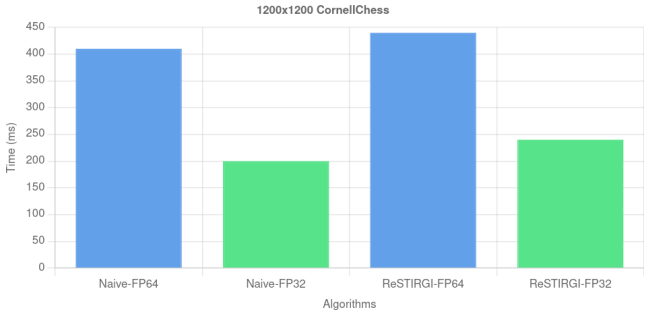


Figure 5: Performance comparison on rendering a 1200x1200 CornellChess at the same camera angle as Figure 4

After generating a specified number of images, we assemble the images into a video using `ffmpeg` with 30 frames per second.

4.11 Miscellaneous Optimizations

4.11.1 FP32 conversion. We swap all our variables to use fp32 (float) instead of fp64 (double). Ray tracing does not require the fidelity of fp64. The scene data is all provided in f32 anyway. We have to carefully ensure that constants (such as `EPS_D`) that are beyond the precision of fp32 are not used, because they will be truncated to incorrect values (such as `EPS_D` becoming 0).

4.11.2 Indexing into vertex array. We construct our `CudaPrimitive` to only contain vertices and normal indices into a global vertices, normal, and texture coordinate buffers instead of storing indices inside the struct. Doing that enables significant less data movement inside GPU (about 1/9 of the data). And since the existence of L1 cache in each streaming processor, all warps can access the same global array to exploit spatial locality.

4.12 Denoising

We utilize Intel OIDN to denoise our resulting 1spp noisy images. Building it with .png support on Fedora 42 was quite a bit of work, we had to build a lot of packages from source.

5 Evaluation

5.1 Setup

All evaluations are performed on an NVIDIA RTX2000 GPU. The evaluation system is running Fedora 42. The evaluation scene dubbed CornellChess (Figure 4) is a modification of an open source chess board scene [8]. The chessboard and the chess pieces have base color textures, normal maps and ORM maps

5.2 Performance

We compare performance across fp32 and fp64 version of the naive and ReSTIR GI algorithms in Figure 5. The naive algorithm performs 1spp sampling without utilizing ReSTIR GI. We observe that naive performs $\approx 7-16\%$ faster than ReSTIR GI. This is because ReSTIR GI performs additional temporal and spatial sampling steps over naive path tracing. Additionally the second kernel invocation adds an extra delay. Furthermore, fp32 is significantly more performant than fp64, giving an $\approx 50\%$ speedup over fp64.

5.3 Renderings

Figure 7 and Figure 6 show variously parameterized renderings of CornellChess comparing naive 1spp sampling, ReSTIR GI, and their respective OIDN denoised versions. The naive raw renders are observably noisier than the corresponding ReSTIR GI renders. The denoised renders of ReSTIR GI are sharper than the denoised naive renders, which look more smudged or creamy. The denoiser correct identifies the diffuse red color of the metallic wall. However, ReSTIR GI and both the OIDN denoiser do not perform well with mixed reflections. This could be potentially because spatial resampling incorrectly resamples the stronger diffuse component in mixed reflections from its neighbours, as the pixels that correctly capture the radiances of the reflected objects are few and far between at 1spp.

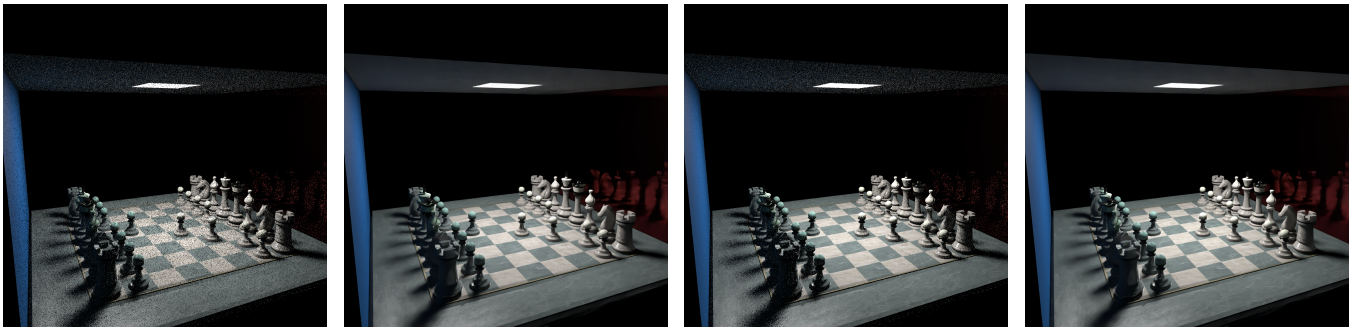
6 Conclusion

We presented a fully featured GPU accelerated neural path tracing framework for fast rendering and denoising. Our renderings come close to Blender's native GPU accelerated render of the CornellChess scene. We highlight that Blender is implemented using NVIDIA OptiX, while our engine uses barebones CUDA, with no RT cores involved whatsoever. Additionally we have not utilized any external library except tinyglTF, programming by hand the entire algorithm.

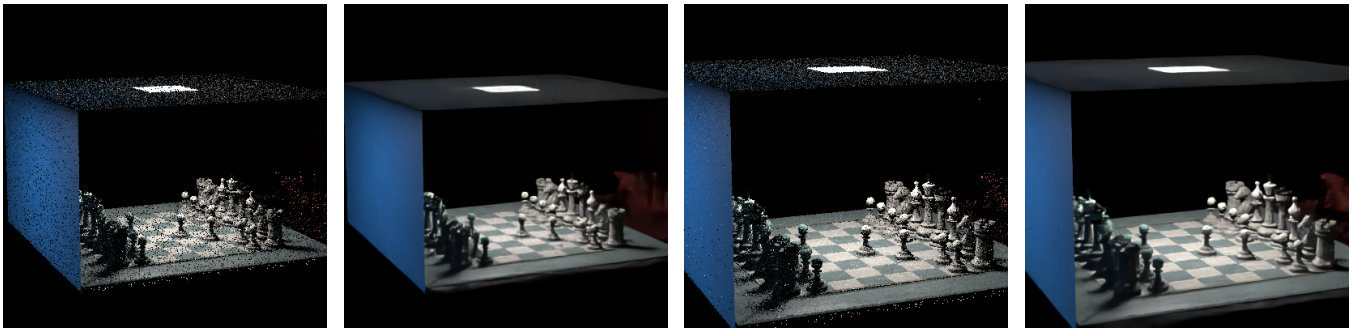
7 Future Work

Primarily we will focus on getting the RISC-V implementations completed next. Orthogonally there are several areas of optimization to speed up our path tracer. We could investigate utilizing advanced CUDA features, kernel fusion, and finally offloading BVH intersections to the RT cores.

Another avenue of exploration is utilizing supersampling methods similar to NVIDIA DLSS, AMD FSR, Intel XeSS. Because rendering at lower resolutions such as 400x400 is an order of magnitude faster than at 1200x1200, it would be beneficial to use these methods to upsample a low resolution path traced image.

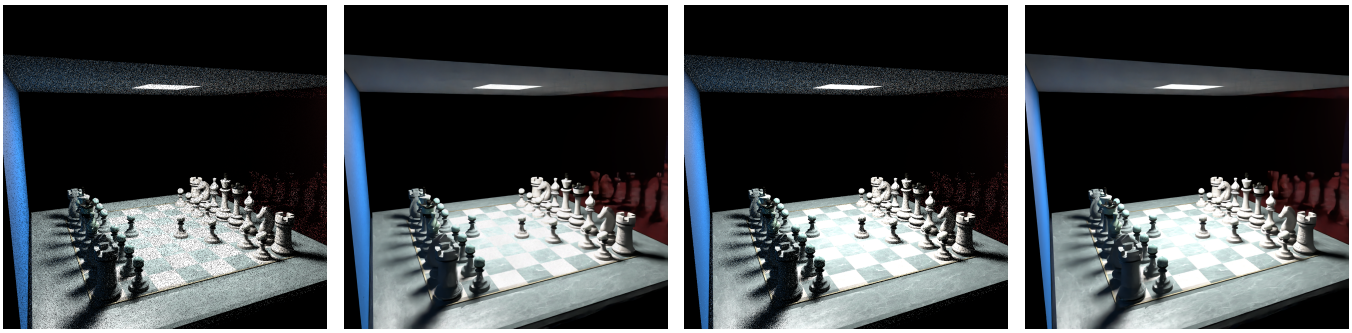


(a) Naive 1spp raw 1200x1200 (b) Naïve 1spp denoised 1200x1200 (c) ReSTIR GI raw 1200x1200 (d) ReSTIR GI denoised 1200x1200

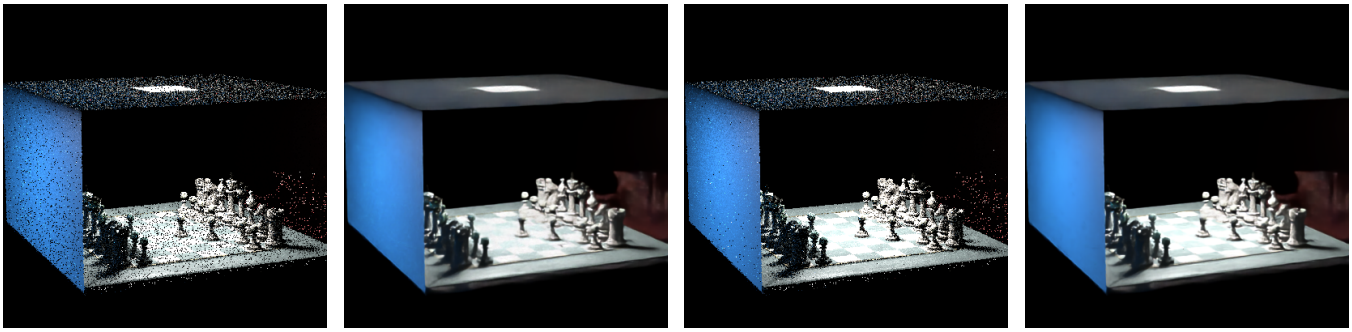


(e) Naive 1spp raw 400x400 (f) Naive 1spp denoised 400x400 (g) ReSTIR GI raw 400x400 (h) ReSTIR GI denoised 400x400

Figure 6: CornellChess with reduced brightness.



(a) Naive 1spp raw 1200x1200 (b) Naïve 1spp denoised 1200x1200 (c) ReSTIR GI raw 1200x1200 (d) ReSTIR GI denoised 1200x1200



(e) Naive 1spp raw 400x400 (f) Naive 1spp denoised 400x400 (g) ReSTIR GI raw 400x400 (h) ReSTIR GI denoised 400x400

Figure 7: CornellChess with high brightness.

Acknowledgments

We acknowledge the copious use of AI. ChatGPT, Gemini and Github Copilot were generously utilized throughout the course of this project. A significant portion of the code base has been debugged/refined with the use of AI. It is what helped us rapidly accelerate our workflow and build a full fledged path tracing engine in a matter of weeks.

References

[1] Attila T. Áfra. 2025. Intel® Open Image Denoise. https://www.openimagedenoise.org. (2025).

[2] Alon Amid et al. 2020. Chipyard: integrated design, simulation, and implementation framework for custom socs. *IEEE Micro*, 40, 4, 10–21. doi:10.1109/MM.2020.2996616.

[3] Benedikt Bitterli, Chris Wyman, Matt Pharr, Peter Shirley, Aaron Lefohn, and Wojciech Jarosz. 2020. Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 39, 4, (July 2020). doi:10/gg8xc7.

[4] Chakravarty R. Alla Chaitanya, Anton S. Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. 2017. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *ACM Trans. Graph.*, 36, 4, Article 98, (July 2017), 12 pages. doi:10.1145/3072959.3073601.

[5] gamegpu.com. [n. d.] Nvidia updates unreal engine 5. https://en.gamegpu.com/game/nvidia-updates-unreal-engine-5-new-support-restir-gi-for-realistic-lighting. ().

[6] AMD GPUOpen. 2025. Neural supersampling and denoising for real-time path tracing. https://gpuopen.com/learn/neural_supersampling_and_denoising_for_real-time_path_tracing/.

[7] The Khronos Group Inc. 2021. Gltf 2.0 specification. https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html#appendix-b-brdf-implementation. (2021).

[8] KhronosGroup. 2022. A beautiful game. https://github.com/KhronosGroup/glTF-Sample-Assets/blob/main/Models/ABeautifulGame/README.md. (2022).

[9] Yaobin Ouyang, Shiqiu Liu, Markus Kettunen, Matt Pharr, and Jacopo Pantaleoni. 2021. ReSTIR GI: Path Resampling for Real-Time Path Tracing. *Computer Graphics Forum*. doi:10.1111/cgf.14378.

[10] Tech Powerup. 2025. Rtx 4090 specs. https://www.techpowerup.com/gpu-specs/geforce-rtx-4090.c3889. (2025).

[11] Antoine Scardigli, Lukas Cavigelli, and Lorenz K. Müller. 2023. Rl-based stateful neural adaptive sampling and denoising for real-time path tracing. In *Advances in Neural Information Processing Systems*. A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, (Eds.) Vol. 36. Curran Associates, Inc., 66390–66407. https://proceedings.neurips.cc/paper_files/paper/2023/file/d1422213c9f2bdd5178b77d166fba86a-Paper-Conference.pdf.

[12] syoyo. 2018. Header only c++ tiny gltf library(loader/saver). https://github.com/syoyo/tinygltf. (2018).

[13] Justin Talbot, David Cline, and Parris Egbert. 2005. Importance Resampling for Global Illumination. In *Eurographics Symposium on Rendering (2005)*. Kavita Bala and Philip Dutre, (Eds.) The Eurographics Association. ISBN: 3-905673-23-1. doi:/10.2312/EGWR/EGSR05/139-146.

[14] Blaise Tine, Krishna Praveen Yalamarthy, Fares Elsabbagh, and Kim Hyesoon. 2021. Vortex: extending the risc-v isa for gpgpu and 3d-graphics. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO '21). Association for Computing Machinery, Virtual Event, Greece, 754–766. ISBN: 9781450385572. doi:10.1145/3466752.3480128.