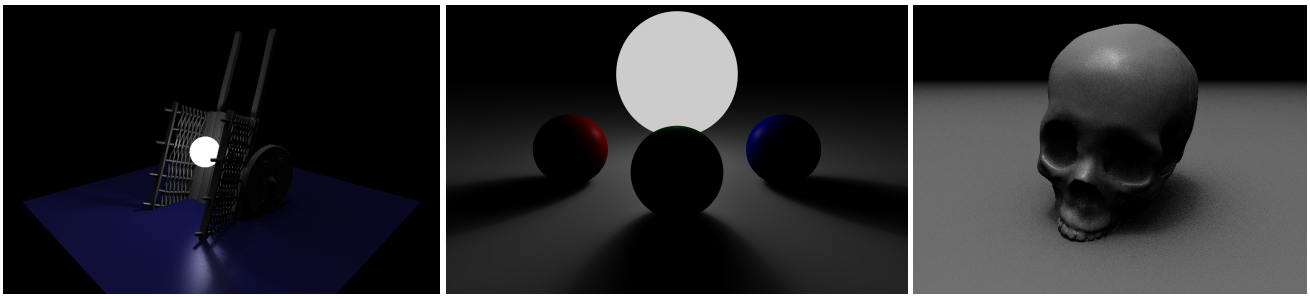
Haste: A Highly Parallel CUDA Monte Carlo Ray Tracer

Bob Somers *
Computer Science Dept.
College of Engineering
California Polytechnic State University

Chris Gibson †
Computer Science Dept.
College of Engineering
California Polytechnic State University



Abstract

We implement and evaluate a full Monte Carlo GPU-based renderer using CUDA, demonstrating the vast improvements gained when running naturally parallel processes such as ray tracing and shading on the Tesla series NVIDIA GPUs. We explain how our implementation builds off of existing algorithms, but goes beyond naïve single-bounce rendering and achieves a high level of parallelism despite its recursive nature. We go into detail where the CPU's responsibilities end and where the GPU's begin. We explain the bugs we ran into, the lessons learned during the design process, and where this project is expected to go next.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing I.3.1 [Computer Graphics]: Hardware Architecture—Parallel Processing I.6.8 [Simulation and Modeling]: Types of Simulation—Monte Carlo

Keywords: ray tracing, GPGPU, Monte Carlo rendering

1 Introduction

Ray tracing is a common technique for generating images based on a three-dimensional scene by tracing the path of light through pixels on a user-defined plane (such as an image plane defined by camera basis vectors). This technique is capable of a high level of realism, while remaining reasonably efficient.

By design, every ray sent out in the initial ray cast is independent. That ray may intersect with objects in a scene, cast out rays in order to determine the shading state, and recurse as the light is reflected or refracted through the objects in the scene that it hits. (Figure 1)

Ray tracing, at its core, is an approximation of how rays of light would behave in reality. There are, however, many drawbacks to typical Whitted ray tracing. First off, the rays cast to check if a particular point is in shadow or not take a very naïve approach to shading. The diffuse value of the light is either applied or it is not. Additionally, indirect light and caustics are not taken into account, which contributes a large majority of light in reality.

*e-mail: rsomers@calpoly.edu
†e-mail: cgibson@calpoly.edu

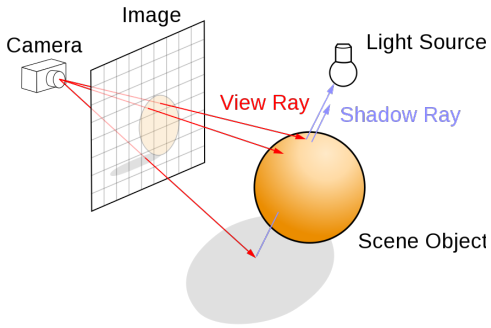


Figure 1: *Ray tracing involves casting rays out of the eye, computing object intersections, and shading those points with respect to the lights.*

The Monte Carlo method of ray tracing involves taking a sufficient number of random samples in order to statistically estimate the the amount of light contributed at any given point. This is broken into two parts, the direct light and indirect light contributions as opposed to the traditional model which simply sets a base ambient value.

With recent advances in computing, particularly in GPU hardware, many have sought to take advantage of the high degree of parallelism that such architectures offer. Enter CUDA, a framework developed by NVIDIA for their next-generation GPUs.

CUDA allows developers to take advantage of the hundreds of cores available in their graphics cards with standard C code. This amount of parallelism is highly useful for a variety of applications such as physical simulations, graphics, financial estimation models, and video encoding to name a few. As long as the algorithm can be broken down into smaller, iterative units, it can take advantage of the CUDA framework.

In this paper, we show how our implementation takes the common recursive approach to Monte Carlo ray tracing and turns it into an iterative one for CUDA. Generalizing the ray casting algorithm so that all rays are treated the same (be they reflected, refracted or shading rays) allows us to implement a flexible framework in CUDA to intersect and shade all rays cast into a scene.

The technique that we present is similar to the layer based approach

first mentioned in [Segovia et al. 2009], but with additional improvements to make it more flexible. Specifically, we show that storing more information with each ray (such as contribution factor, and its destination pixel and layer) allows us to avoid large memory-hungry ray batching and treats all rays equally, simplifying the implementation significantly.

2 Related Work

Ray tracing on the GPU has been an area of heavy research recently with work similar to ours but with some minor variations. Most implementations have made drastic assumptions about particular parts of the rendering process, such as avoiding ray bouncing (ignoring the recursion problem) or avoiding Monte Carlo (ignoring the memory problem).

2.1 Seminal Work

Turner Whitted's research [Whitted 1980] is considered to be the fundamental underpinnings of ray tracing. Although the idea of casting rays from the eye into the scene was developed a few years earlier by Arthur Appel, Whitted was the first to follow this idea recursively, developing the basic algorithms that allow us to do reflection, refraction, and shadows with ray tracing. (Figure 2)

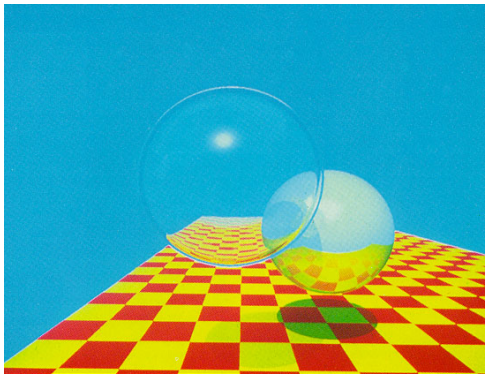


Figure 2: *Whitted's renders were the first to demonstrate reflection, refraction, and shadows.*

2.2 Iterative Layer-Based Approach

A significant hurdle in this project was rewriting the traditional ray tracing algorithm to be iterative. An iterative, layer-based approach is presented in [Segovia et al. 2009] that shows how one of the most difficult problems with rewriting the algorithm can be avoided using a stack of image buffers for each level of recursion.

In sections 4.1 and 4.2, we explore the parts of their work that apply to Haste, as well as the adjustments we made to enhance the flexibility of their approach.

2.3 NVIDIA OptiX

OptiX is a generic ray tracing engine built using CUDA for NVIDIA's GeForce and Tesla series GPUs. OptiX was designed to be as generic and flexible as possible, to accelerate the task of casting rays for a variety of scientific purposes, not just image rendering. Using OptiX involves writing subprograms to test intersections of objects in the scene, as well as compute shading values (if desired) when intersections occur. [Parker et al. 2010]

3 Lighting Model

Before we could begin porting the ray tracing algorithm to CUDA, we needed to nail down exactly how we were going to compute shading for the objects in the scene. For Haste, we opted to go with the Phong lighting model, which we were both familiar with. The Phong model attempts to capture the way light interacts with a surface by breaking it up into three distinct components.

- The **ambient** component is light that reaches the surface by being reflected off of other objects in the environment. In real-time pipelines, this is usually a constant flat color and is somewhat of a hack.
- The **diffuse** component is light that reaches the surface directly from a light and then is scattered in many directions due to the surface's microscopic roughness. This is the primary contributor for most objects.
- The **specular** component is light that reaches the surface directly from a light and then reflects around the same angle due to the surface's smoothness. It is typically identified as a "shiny spot", and is dependent on the viewer's location relative to the object and the light.

In addition to these three, there is one additional contributor that is typically left out of the Phong equation in real-time pipelines.

- The **emissive** component is light that is emitted by the surface itself rather than reflected from other lights or the environment.

While most real-time pipelines ignore this term since lights and objects are handled separately, our global illumination approach requires us to consider this term as well. In fact, Haste makes no distictions between lights and objects. Lights are simply objects with an emissive component greater than zero.

Thus, our lighting equation, shown in (1), determines the illumination (color) I at a point p on the surface of an object.

$$I_p = E_p + A_p + D_p + S_p \tag{1}$$

Where E_p, A_p, D_p, and S_p are the emissive, ambient, diffuse, and specular components respectively.

3.1 Diffuse Lighting

The diffuse component, shown in (2), is a sum of all the contributions from each light m. Each light's contribution is function of the surface's diffuse reflection constant k_d, the direction vector $\vec{L_m}$ from p to the light, the normal vector $\vec{N}$ of the surface at p, and the diffuse color i_d of the surface and the light.

$$D_p = \sum_{m \in \text{lights}} k_d \left(\vec{L_m} \cdot \vec{N}\right) i_d \tag{2}$$

3.2 Specular Lighting

Similarly, the specular component, shown in (3), is a sum of all the contributions from each light m. Each light's contribution is a function of the surface's specular reflection constant k_s, the reflection vector $\vec{R_m}$ of the light vector about the surface normal at p, the view vector $\vec{V}$ from the p to the viewer, the specular power α (shininess), and the specular color i_s of the light.

$$S_p = \sum_{m \in \text{ lights}} k_s \left(\vec{R_m} \cdot \vec{V}\right)^{\alpha} i_s \tag{3}$$

3.3 Monte Carlo

The problem with (2) and (3) is that if we only take one sample per light, we treat our lights as point lights instead of area lights. This causes us to lose our soft phenomena, such as soft shadows. In reality, p may be able to see part of the light, while some of the light is obscured.

To simulate this, we use a Monte Carlo approach and uniformly sample across the volume of the light. For each sample j in n samples, we generate a random point p_L inside the volume of the light and use that point to calculate the light direction vector L_m. This leads to the following adjustments to (2) and (3) respectively:

$$D_p = \sum_{m \in \text{ lights}} \left[\sum_{j=1}^{n} \frac{1}{n} k_d \left(\vec{L_m}\left(p_L\right) \cdot \vec{N}\right) i_d \right] \tag{4}$$

$$S_p = \sum_{m \in \text{ lights}} \left[\sum_{j=1}^{n} \frac{1}{n} k_s \left(\vec{R_m}\left(p_L\right) \cdot \vec{V}\right)^{\alpha} i_s \right] \tag{5}$$

3.4 Ambient Lighting

As mentioned above, in real-time graphics pipelines the ambient term is somewhat of a hack. It's flat color value is usually used to prevent an object from fading completely to black in areas where its diffuse contribution is small. The constant flat color is a local approximation of what actual ambient lighting would contribute.

However, in a global illumination system (such as Haste), we are not concerned with real-time performance, but rather rendering accuracy. Thus, we can accurately compute the ambient contribution, again using a Monte Carlo approach.

The ambient component, shown in (6), is a sum of all the contributions of each ambient sample taken. For each ambient sample, we generate a random ray in the hemisphere normal to p and find the closest object it intersects with. We call this point p_A. For that point, we compute the simple (non-Monte Carlo) diffuse and specular lighting at that point, D_{p_A} and S_{p_A}, explained in (2) and (3). In addition, each ambient sample relies on the surface's ambient reflection constant k_s.

$$A_p = \sum_{j=1}^{n} \frac{1}{n} k_a \left(D_{p_A} + S_{p_A}\right) \tag{6}$$

3.5 Emissive Lighting

Lastly, the emissive component, shown in (7) is rather straightforward. It simply consists of the surface's emissive constant k_e and the emissive color i_e of the surface. There is no need to sample this component with a Monte Carlo approach.

$$E_p = k_e \, i_e \tag{7}$$

4 Parallelizing the Algorithm

Graphics in general, especially ray tracing, is one of the classic problems that is said to be "embarassingly parallel". We quickly found, however, that this is usually touted by people who have never attempted to implement a highly parallelized ray tracer. While the problem indeed has an enormous amount of exploitable data-level parallelism, there are significant hurdles in rewriting the algorithm itself to work on highly parallel hardware, such as NVIDIA's CUDA platform.

4.1 Eliminating Recursion

The crux of the problem is that the traditional ray tracing algorithm itself is recursive, with successively bounced rays contributing color in a cumulative fashion. Color is summed from the furthest bounce inwards for each pixel as rays are successively popped off the stack, like so:

$$\text{pixel} = \text{ray}_0 \cdot R_0 \left(\text{ray}_1 \cdot R_1 \left(\text{ray}_2 \ldots R_{n-1}\left(\text{ray}_n\right)\right)\right) \tag{8}$$

where R_{n-1} is the fraction of light from ray_{n-1} that was bounced in the direction of ray_n.

CUDA, at this time, does not have support for recursive function calls. Thus, the algorithm needed to be rewritten iteratively. However, this must be done carefully, because naïvely refactoring the standard ray tracing algorithm does not take this recursive summing into account.

[Segovia et al. 2009] presents an interesting approach which uses a series of layer buffers to effectively give you a stack of final image buffers. The rays are traced in "batches", where each new batch draws into the next layer buffer down. The layer buffers are then summed in a weighted fashion such that the final sum matches the recursive approach.

Our approach is similar, but with some minor modifications. The Segovia approach is limited because all objects in the scene have the same R_n, since the layers are summed after the fact. In Haste, the rays themselves contain additional state, discussed in more detail in section 4.2.

One of those pieces of state is the ray's "contribution factor", which is multiplied against all shading calculations for that ray. When a ray spawns a reflected or refracted ray, the new ray has its contribution factor set to that of its parent, multiplied by the reflection or refraction constant for that surface, like so:

$$C_n = C_{n-1} \cdot R_{n-1} \tag{9}$$

where C_n is the contribution factor for ray_n.

with this modification, the correct color summing is performed on a per-object basis rather than a per-layer one. Thus, the layers should be summed using a flat sum instead of the weighted sum used in the Segovia approach.

4.2 Streaming Stateful Rays

In [Segovia et al. 2009], rays are traced in batches, with each batch corresponding to a new layer of "recursion". Since Haste uses a Monte Carlo method for rendering, it is definitely conceiveable that the number of rays in successive layers could grow wildly out of control, potentially using up all available memory on the GPU.

To combat this, we encode additional state information in each ray so that it knows exactly what pixel and layer it contributes to. This allows us to trace rays in arbitrary batch sizes to tune for memory efficiency as well as kernel launch size. With one thread allocated per ray, our additional state allows us cast more total rays than the maximum possible launch size.

A structure to hold a typical stateless ray would look something like listing 1.

```
typedef struct Ray {
    float3 origin;
    float3 direction;
} Ray;
```

Listing 1: *A traditional stateless ray structure.*

Our new stateful ray structure stores additional information along with the origin and direction, as seen in listing 2.

```
typedef struct Ray {
    float3 origin;
    float3 direction;
    float contrib; // contribution factor
    uint32_t layer; // contributing layer number
    ushort2 pixel; // contributing pixel coordinates
    bool unibounce; // single bounce ray?
} Ray;
```

Listing 2: *A stateful ray structure.*

With this additional information, we are no longer dependent on tracing rays in batches per layer, nor are we limited to a constant R_n for all objects. Each ray is independent and stores enough data to figure out which pixel and layer buffer to write its contribution into.

This allows us to constantly stream input rays into the GPU and extract output rays (such as reflected and refracted rays) from the ray tracing kernel. The output rays are pushed into a ray queue that stores pending rays. When the GPU is ready for a kernel launch, we extract a packet of rays from the queue (one ray per GPU thread), copy the packet to the device, and launch the kernel. This process continues until the ray queue is depleted.

5 Squashing Bugs

Many bugs arose during the development of Haste. Some were architectural, based on issues in our initial implementation decisions, while others were simply typos, but were deviously hidden by the complexity of the system.

5.1 Normal Interpolation

One major bug that appeared turned out to be entirely unrelated to CUDA. Although normal interpolation within triangles seemed to work well, polygonal meshes (such as the Stanford bunny) would show seams at the shared edges of triangles, as seen in Figure 3. This lead to a thorough investigation with the assumption that the models were being loaded incorrectly or that the normals were being modified incorrectly. The issue turned out to be a simple arithmetic bug in the calculation of the interpolated normal.

While investigating the normal interpolation bug, a modification to the normal and vertex parsing code caused some rather unexpected results, as seen in Figure 4.

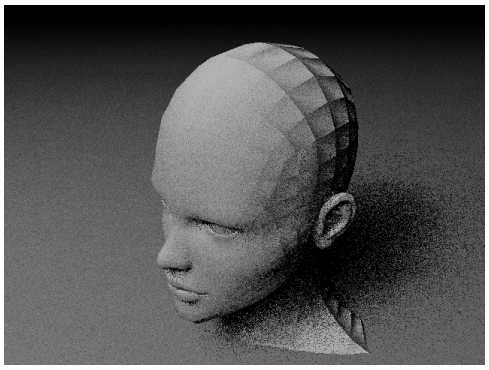


Figure 3: *Model with incorrectly interpolating normals.*

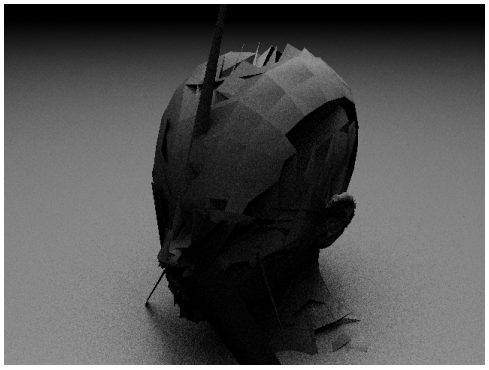


Figure 4: *Head model ruined by incorrect vertex parsing.*

5.2 Bounding Volume Hierarchy Traversal

There are many spatial data structures that can speed up intersection tests in a ray tracer. We chose to implement one of the most common, a bounding volume hierarchy (BVH). The basic idea is that objects in the scene are grouped into axis-aligned bounding boxes (AABBs) that are then grouped spatially in a binary tree structure (Figure 5). Because of the spatial grouping, if a ray does not intersect a bounding box node in the tree, it is guaranteed to not intersect either of the two subtrees or any objects contained therein. Thus, we can reduce the number of intersection tests from $O(n)$ to $O(\log n)$.

This is a big win for performance. For example, in a scene with 30k triangles, instead of testing each ray against all 30k objects we can reduce this down to roughly 16 intersection tests. The only downside is that the intersection test code becomes more branchy, which is bad for the CUDA architecture.

A significant hurdle in implementing the traversal code is, not surprisingly, eliminating recursion. Traditionally both BVH construction and traversal algorithms are recursive. Refactoring them to do an interative traversal is not trivial, because it's not simply a matter of traversing the tree. Since the point of building the BVH is to selectively *not* traverse large chunks of the tree, there is some logic at each node that needs to make decisions about where to head next based on the traversal history.

Our first approach was stackless, and involved using parent pointers and a previous record to determine both the location in tree and where to head next. This code is still buggy, and at this point, will probably be scrapped.

Since we know the depth of the tree at construction time (which

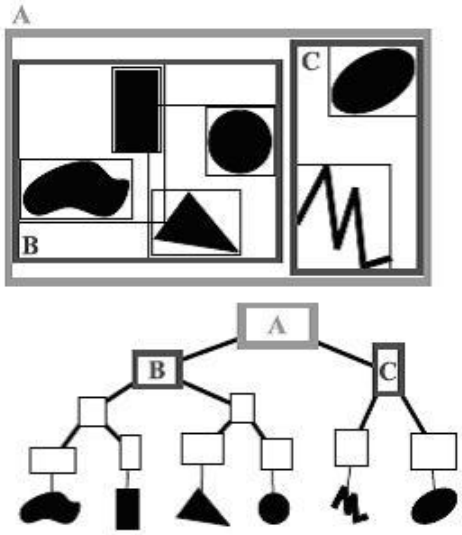


Figure 5: *An example of a two-dimensional bounding volume hierarchy.*

happens on the CPU), we know how much memory to allocate for a traversal stack. We will likely refactor the traversal algoroithm to use a stack-based iterative traversal. This is a high priority item on our future work agenda.

6 Results

Results can be viewed at the end of this paper. Figures 6, 7 and 8 are all pictures generated by our renderer.

7 Future Work

Although we have generated some striking images, there is still much to be expanded on for Haste to be considered complete. These include general improvements to the base code (which will continue on the master branch) as well as specialized improvements for our Masters theses (which will happen on separate branches).

7.1 General Improvements

Although making certain assumptions from the start helped us avoid many issues that naïve GPU ray tracers run into, many issues exist that will likely be improved in the near future.

One example is the fact that the CUDA kernels are currently only supported by Fermi architecture, which limits what hardware the software can run on. This is not a trivial issue, as some of our algorithms assume certain features and operations are available in the hardware when running. The most important operation we rely on is atomic addition of floating point numbers, which is not available in pre-Fermi architecture GPUs.

Additionally, our current method of transferring ray packets to and from the GPU is inefficient, reallocating the device-side memory for each packet sent. This can be very costly. An improvement beyond simply allocating device-side memory once would be to setting up CUDA streams, allowing the GPU to be performing three tasks at once: Pulling rays from the CPU, testing ray-object intersections and shading, and pushing intersections off the GPU.

The CUDA kernel currently implemented is a one-kernel-fits-all direct-lighting shader. When more complex scene objects (like volumes) come into play, this will not be a viable option. The intersections, shading and volume integration may all have to be split into separate kernels in order to take full advantage of the SIMD architecture that CUDA offers.

7.2 Distributed Rendering of Massive Scenes

A current limitation of all GPU ray tracers is that all the scene geometry must fit into the GPU's memory. This is a severe limitation for adoption of GPU rendering in the entertainment industry, since it is not uncommon to render scenes with many large meshes, thousands of textures, and volumetric effects.

Haste is highly parallel on a single machine, but we would like to extend it to be highly parallel across a cluster of machines. Specifically, we'd like to parallelize across the distribution of scene geometry, allowing a distributed version of Haste to render scenes too large to fit on a single machine.

7.3 High Performance Volume Rendering

Future improvements will include the generation, lighting, and rendering of volumetric data within scene files. Fluid and smoke simulations within the CUDA kernel will significantly speed up simulation times. Although parallel volume ray tracing is more complex due to its naturally large size requirements, there are many algorithms to help mitigate the memory issues.

Acknowledgements

We'd like to thank Dr. Zoë Wood for teaching the Advanced Rendering class and sparking our interest in ray tracers. In addition, her constant prodding to stop building hacky workarounds and go with a fully Monte Carlo solution was, as usual, right on target.

We'd also like to thank Dr. Chris Lupo for his ability to procure obscenely powerful supercomputers for us to play around with, as well as his insightful lectures that always seemed to end with the punchline: "It depends".

References

PARKER, S. G., BIGLER, J., DIETRICH, A., FRIEDRICH, H., HOBEROCK, J., LUEBKE, D., MCALLISTER, D., MCGUIRE, M., MORLEY, K., ROBISON, A., AND STICH, M. 2010. Optix: a general purpose ray tracing engine. *ACM Trans. Graph. 29* (July), 66:1–66:13.

SEGOVIA, A., LI, X., AND GAO, G. 2009. Iterative layer-based raytracing on cuda. In *Performance Computing and Communications Conference (IPCCC), 2009 IEEE 28th International*, 248 –255.

WHITTED, T. 1980. An improved illumination model for shaded display. *Commun. ACM 23* (June), 343–349.

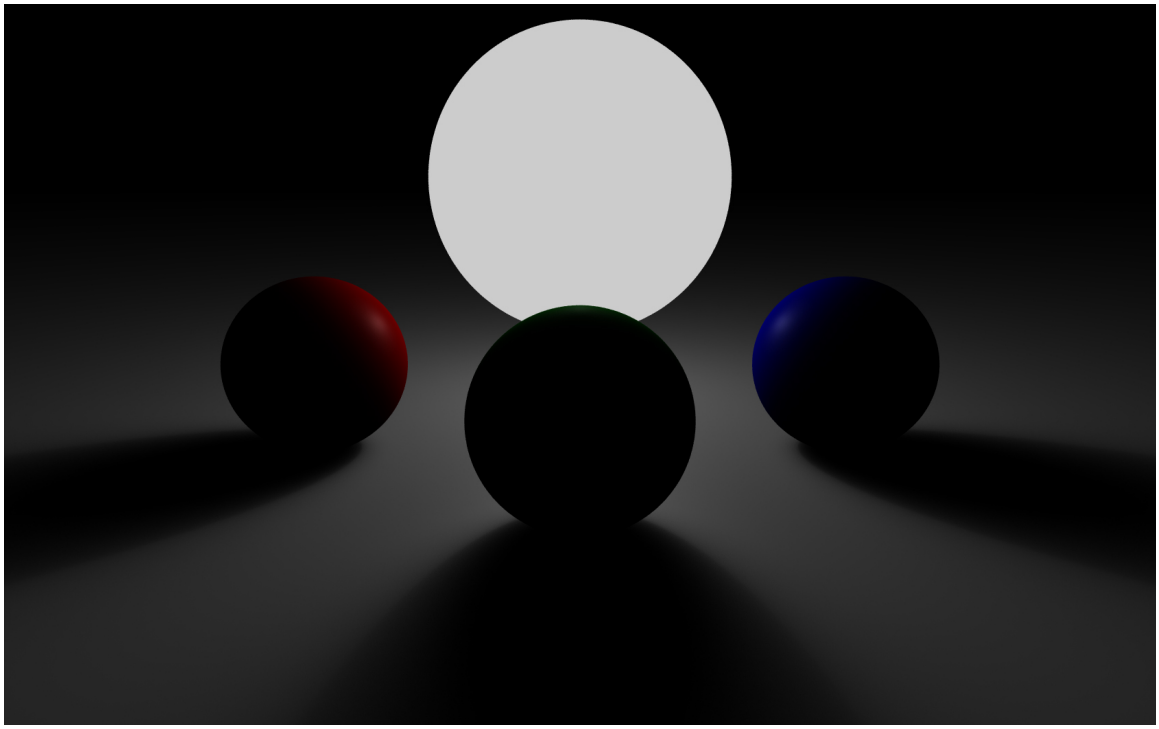


Figure 6: *Three spheres casting soft shadows from an area light.*

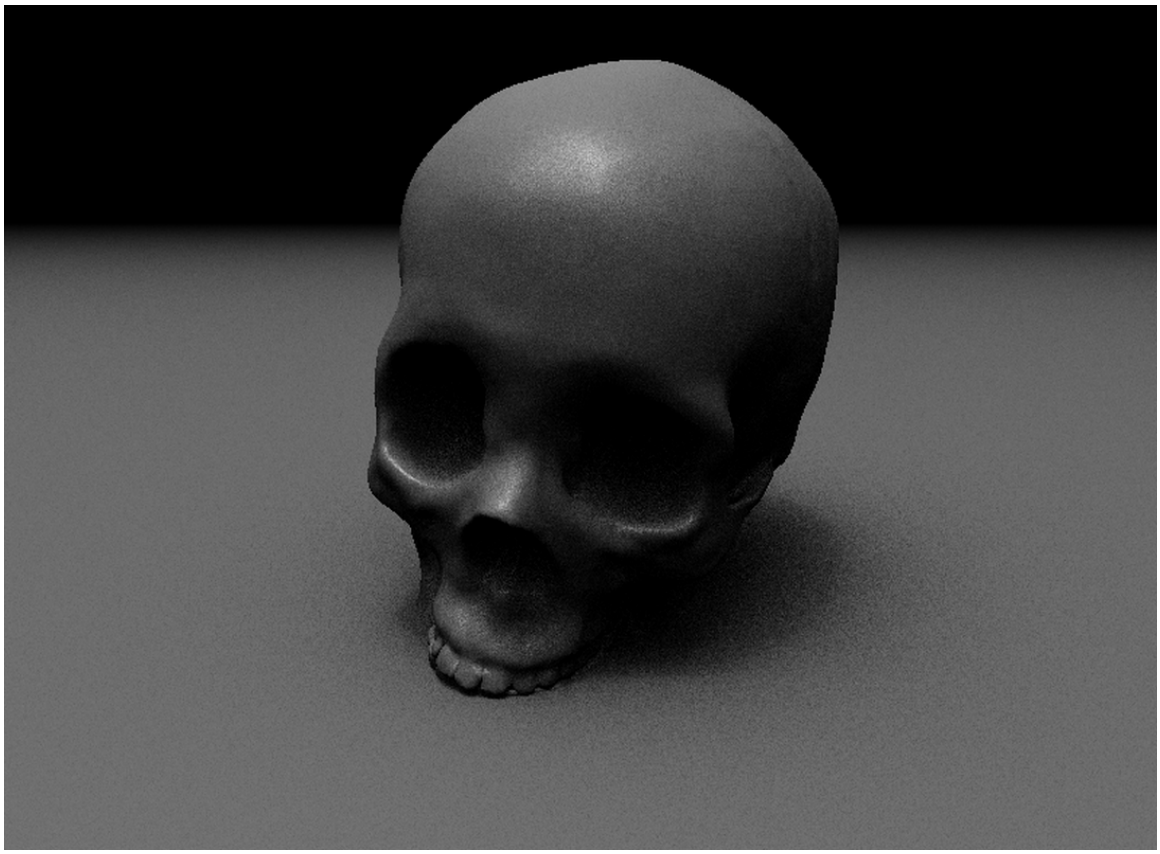


Figure 7: *Skull model rendered with an area light.*

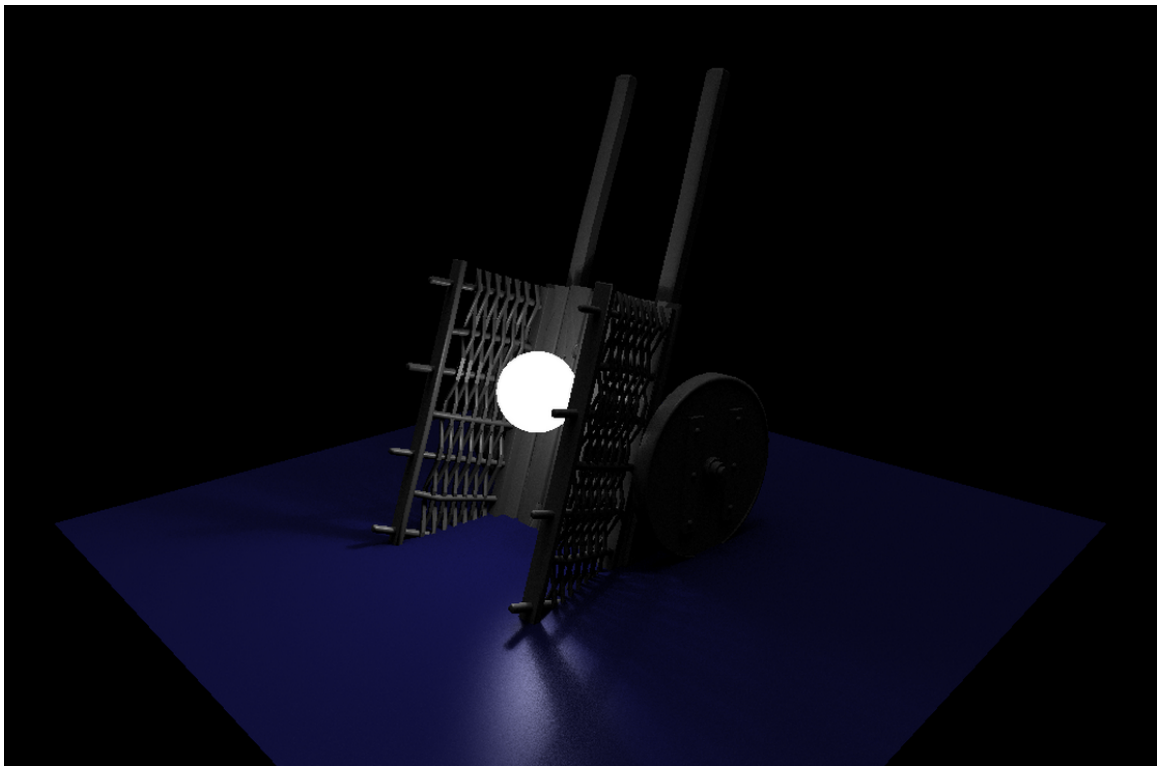


Figure 8: *Cart model rendered showing the soft shadows cast with a small area light.*