5-2022

Explorations in Distributed Ray Tracing and Photometry of Large Scenes

Connor W. Weisenberger
Trinity University, cweisenb@trinity.edu

Explorations in Distributed Ray Tracing and Photometry of Large Scenes

Connor Weisenberger

Abstract

This work encapsulates three explorations into different implementations of distributed ray tracing, that is to say, ray tracing that has been distributed across multiple machines. Our goals lie in the rendering of scenes with more geometry than can fit within the memory of a single computer, so we focus on the distribution of memory. Ultimately, this work discusses a Spark standard (or classical) distributed ray tracer, a Spark photometric distributed ray tracer, and a single-machine Akka Typed photometric ray tracer with some basis for future distribution. Individual timing results for each ray tracer are included, but they cannot be compared due to differences in their generation. Qualitative comparisons between the ray tracers and their approaches are made, and recommendations are given to future researchers in this niche.

Acknowledgments

Dr. Mark C. Lewis of the Trinity University Department of Computer Science, for keeping this train on the tracks, even when the train was actively bouncing from the track onto nearby rocks and then back, and also on fire. His actions are what allowed this work to begin, to continue, and to end, and without him all of this would've been entirely impossible.

My family, for encouraging me whenever I feel like its all hopeless, and for providing a cool, rational sounding board when I need it most.

My friends, for listening to me rant about this work even as they dealt with their own struggles and tasks.

The Trinity Computer Science Department, for the unending support and resources they provide.

Explorations in Distributed Ray Tracing and Photometry of Large Scenes

Connor Weisenberger

A departmental senior thesis submitted to the
Department of Computer Science at Trinity University
in partial fulfillment of the requirements for graduation
with departmental honors.

April 1, 2005

Thesis Advisor

Department Chair

Associate Vice President
for
Academic Affairs

Explorations in Distributed Ray Tracing and Photometry of Large Scenes

Connor Weisenberger

Contents

List of Tables

List of Figures

Chapter 1

Background

1.1 Ray Tracing

Ray tracing is the process of intersecting some rays with some geometry to create an image. Further specifics vary depending on the implementation, but in the standard implementation rays are cast from the "eye" of a metaphorical viewer out in all directions within their field of vision and checked for intersections against some geometry. Rays that intersected the geometry then generate new rays from the point of that intersection towards the lights. Whether these rays collide with geometry before they would hit the light determines whether the ray's origin point is lit. All of that information (intersection point, object of intersection, light visibility) is then brought back together and used to determine the color of the pixel that matches that point where the original ray from the eye hit the geometry. If that original ray doesn't hit the geometry, we know that pixel (where the ray was fired from) will be the background color.

There are also many opportunities for expansion on this basic ray tracing model (reflections, refraction, motion blur in animated scenes, many others that are more complex).

One simpler example is reflections (allowing rays to reflect off of the first few (exact number up to the user) geometries they hit, often based on the reflectiveness of the hit surface). These expansions are what give ray tracing it's notoriety for realism and detail, but they aren't the focus of this research. This research is primarily motivated by a desire to test out and learn about potential frameworks for a future, more visually complex ray tracer that could be used to generate visually interesting and realistic images of large numbers of bodies in space.

Ray tracing is interesting for graphics research because depending on the implementation, it can model a number of impressive visual details (complicated reflections, glare, shadows, etc) due to the relatively realistic way in which it creates images (compared to rasterization, the currently dominant way of generating images). Storing the geometry for ray tracing is more memory intensive - objects outside of the scene can still interact with the image and thus must be included in every computation. We use a `KDTree` to offset some of the actual computational cost, but all of the geometry is still represented in memory [6]. This memory reliance, along with long computation times for scenes with complex geometry or lighting behavior, seems to be the primary roadblock to more widespread use of ray tracing [9].

This memory limitation is part of what makes the distribution of ray tracing interesting. Ray tracing is an "embarrassingly parallel" problem, in that it is easy to distribute each ray as they don't mutate the scene, but as a memory-heavy computation it benefits greatly from distribution. Distribution introduces some inefficiency in the form of connection latency, but as scenes get larger the benefits of distribution grow to outweigh the costs of sending data between machines.

1.1.1 Photometric Ray Tracing

One such realistic implementation of ray tracing is photometric ray tracing. The primary difference between photometric ray tracing and the standard ray tracing described above is that in photometric ray tracing, the rays originate from the light sources. You may recall that in the standard model, the rays originated from the "eye". The photometric rays continue from the light sources into the scene, where they are checked for intersections with the geometry. If a collision occurred, rays are cast from the point of collision to the eye of the camera, to determine whether or not this point is visible to the viewer. This process is very similar to the standard process, but executed in a slightly different order. This difference in order makes photometric ray tracing more similar to the real behavior of light than standard ray tracing. Photons that start at the light sources mimic the behavior of photons in the real world, and thus more complex, real world lighting behaviors are easier to simulate.

Another key difference is that in the standard model, at least one ray must be fired for each pixel in the scene. This generates some unrealistic bias in the image because points that may not actually receive all that many photons from their light sources still get the same treatment as those that do. In some ways, the photometric approach often ends up supersampling the lit pixels to create softer shadowing and edges while not hitting other pixels at all. The standard approach ensures every pixel is hit, but supersampling must be explicitly performed. This difference, along with easier scatter calculations for light refraction, give photometry the edge in image quality.

One more thing to note about photometric ray tracing is that, generally, more rays are required to generate a photometric ray traced image than a standard ray traced image. This is again because there is no one ray to one pixel guarantee in the photometric process. Thus, high quality photometric images are created by throwing repeated batches of photons into the scene until a predetermined number of photons or satisfactory visual quality is reached. Standard images are often done in one “take” because it is easy to know ahead of time how many rays will be required for a certain image.

1.2 Distribution

We have observed that standard approaches to ray tracing have memory issues with large geometry. Inevitably, it is necessary to ray trace scenes with a large amount of geometry, and sometimes the memory representation of this geometry is larger than can fit within a single computer’s memory. This is where the desire to distribute the ray tracing process originates.

One note here is that “distributed ray tracing” is an overloaded term. It refers here to the process of distributing ray tracing over a number of machines, but in other works it can refer to a specific method of ray tracing that better captures certain visual phenomena [10].

This research and all three of these ray tracers were specifically motivated by a prior collaboration between Dr. Lewis and the American Museum of Natural History, where a number of high quality renderings of large sections of the rings of Saturn were needed [14]. These renderings were first done by stitching together smaller cells of the image - breaking the scene up into smaller pieces and working on them sequentially. This worked, but was slow and often tested the limits of memory even on a strong machine. Thus, it was decided that there should be an analysis of possible methods to distribute this process, specifically the memory load of large geometry.

The general idea of distribution is straightforward: when a single machine is too slow or otherwise unfit (likely not enough memory), multiple computers, also known as a cluster, are used together to perform computations that would otherwise be impossible or much slower. In a distributed environment, there are multiple semi-autonomous machines with their own memory that pass information between themselves, often to solve a large problem. Two methods of distribution explored in this work are Spark and Akka, both of which have support for multiple languages but are utilized in this work with Scala [11], [3].

Other distributed ray tracers do exist [18], [20], [7]. They vary in performance, purpose, and implementation quite considerably. Many are interactive ray tracers, which render the scene "on-the-fly" in response to user movement of the camera [15], [7]. Some are designed to work with specific hardware, often referred to as "tightly coupled parallel approaches" [18]. The use cases of distributing the ray tracing process vary. Some are hyper-performant, dedicated pairings of hardware and software designed to crunch rays as fast as possible, others to improve the efficiency of already existing processes by utilizing underused machines [18]. There is a distinction among distribution, which is whether the distribution is performed for memory's sake or computation's sake. Interactive ray tracers tend to be computationally distributed. It actually seems that distributing the memory involved in ray tracing is at least slightly rarer than distributing the computation, but this isn't a survey paper and we cannot prove that. The size of the scenes involved also varies with ray tracer. Some use distribution to simply spread the computational load as opposed to our memory-centric distribution.

1.2.1 Spark

One such implementation of this idea of distribution is called Spark. The Apache Foundation, the maintainers of Spark, describe Spark as a "multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters". Spark provides a number of implementations of their distribution model, but the one primarily used here is the Resilient Distributed Dataset (RDD). The RDD is a data structure that stretches across all the worker nodes in your cluster where elements can be operated on in parallel. Scala Spark provides a number of Scala Collections Library-esque functions for RDDs, so using them is very intuitive and similar to standard collections logic. One important note here is that these functions are almost entirely collection-wide, in that they operate on every element of the collection. This and more about Spark will be elaborated on later, but Spark's requirement of collection-wide operation is a key difference between it and Akka.

1.2.2 Akka

Another approach to distribution and parallelism is explored within Akka. Akka provides a framework for distributed message passing. What this means in practice is that the programmer defines some behaviors (called actors in Akka Classic) that dictate the responses to these messages. Some major benefits of this are that the system is quite resilient (not as out-of-the-box resilient as Spark is, but the functionality is present) as there are few if any single points of failure. Akka provides functionality to define actions to occur should a Behavior become unresponsive or otherwise fail, which also improves resilience. Again, these features are supported by Akka but do require some explicit setup by the programmer, which is more than Spark requires.

Another benefit of Akka is that the behaviors (actors) and their messages are lightweight. Akka boasts that a single machine can handle 'up to 50 million [messages per second]' and that 2.5 million actors can fit into a gigabyte of heap space [3]. This surely scales with the sizes of the messages and actors, as well as the specifications of the machine, but Akka is certainly performant.

One major change occurring in the Akka space at the time of this work is the move to

Akka Typed. This will be further discussed in Section 4.1, but the general theme of the changes is a move to more explicit type definition, particularly in regards to what messages an Actor or Behavior receives. Lightbend, the sponsors of Akka, currently suggest that new projects are created with Akka Typed [2]. There is still support for Akka classic, and the two can even be intermingled in the same project, but it certainly seems that Akka Typed is the future of Akka. As an example of this, all of the documentation readily available on Lightbend's page is for Akka Typed. The classic documentation is still available for now, but is relegated to it's own section, away from the current documentation [1].

1.3 Geometry

The geometry used in this simulation was provided by [17]. We use a single timestep, which contains the data for roughly 2.3 million spherical particles. In scenes with more than one of these timesteps, referred to henceforth as "panels" or "geometry files", we tile them such that side by side they align with the edges of the image. It is important that the image is entirely filled in for proper tests because in the standard model of rendering rays are still cast into empty space and that results in less computation than a filled image. The photometric model doesn't have this issue, but to aid in comparisons between the two we use the same geometry tiling scheme.

Each file has a 1:10 aspect ratio, so we can make square tiled images that entirely fill the frame for scenes with $10n^2$ panels of geometry. The first of these is 10 (1 row of 10), then 40 (2 rows of 20), then 90 (3 rows of 30). None of our attempts at any scenes with 90 or more tiles have successfully completed, whether due to cluster size or issues at scale remains to be seen.

One interesting note about the geometry panel (data from Saturn ring simulations) is that it features a moonlet, roughly in the center, which has a “propeller” structure of surrounding particles (that S-shaped feature extending from the moonlet) [19]. These propeller structures are the primary visible characteristic of the moonlets, so it’s neat that we can see them in our renderings of the geometry tiles as well (they are quite visible in Figure 3.2).

Another interesting note is that all of our scenes make use of perfect spheres to represent the ring particles. This is viewed as tolerable because each sphere is so relatively tiny compared to the entire scene, but adding additional geometry functionalities is on the list of potential future work.

1.4 Cluster Configuration

Trinity University has a cluster of 8 machines referred to as the Pandora Cluster. A single machine in this cluster has an Intel®Xeon®CPU E5-2683 v4 @ 2.10GHz CPU (32 cores) and 16 GB of RAM.

The Spark machines utilize the cluster with one driver node and 7 worker nodes which actually contain the RDDs. The cluster itself does require some configuration for Spark, but once configured it can easily accept jobs from a single machine and also includes a web UI for viewing the details of both active and past jobs.

While not expressly used in this work, the Akka cluster ray tracers have one `FrontendNode` and seven `BackendNode`s. The frontend nodes handle leading the backend nodes, the image itself, and the distribution of geometry amongst the backend nodes, whereas the backend nodes handle the geometry and the intersections (the most memory and computationally intensive tasks).

Chapter 2

Distributing Ray Tracing with Spark

2.1 Spark

Spark is introduced in Section 1.2.1. Anybody who is familiar with Spark outside of this paper might have the idea that Spark is for big SQL-esque operations, as this is likely high on the list of its most common uses, but it is certainly not limited to that role. Traditional, business-class applications of Spark likely make use of Spark `Dataframe`s, which support columnar data (and are more readily applied to database-adjacent use cases than `RDD`s). This research makes use of `RDDs` (Resilient Distributed Datasets), which are slightly lower-level than their `Dataframe` counterparts. `RDD`s function very similarly to other collections (in Scala) and have similar member functions.

`RDD`s essentially spread a collection out over a cluster, broken apart into "partitions" that can be worked on by their owner machine without disturbing or inhibiting other machines. When the `RDD` is needed for some work, that work is distributed out to each machine, and

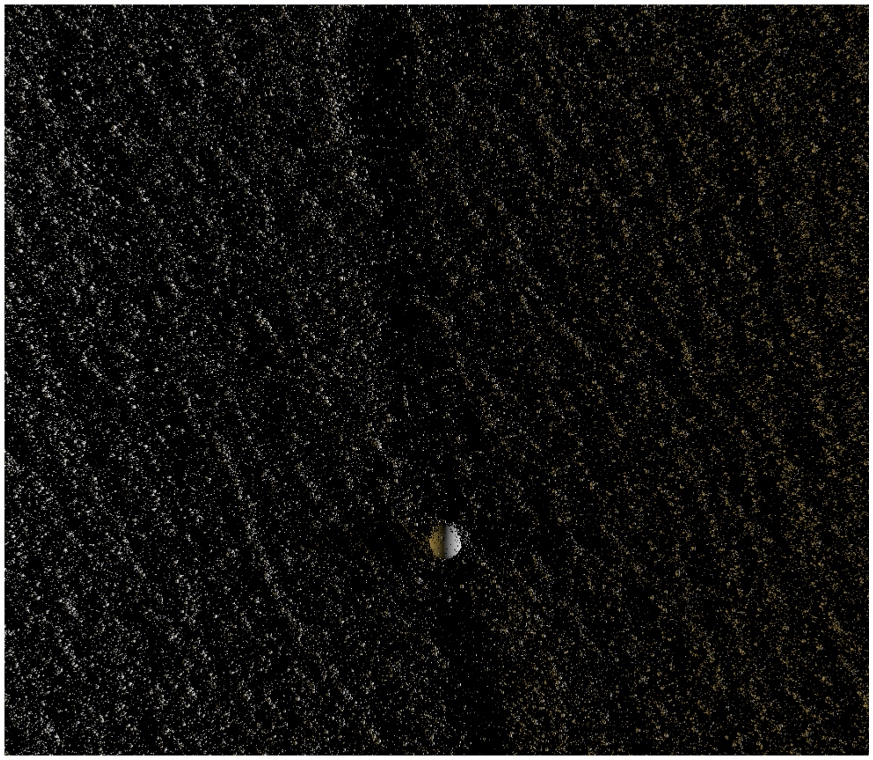

Figure 2.1: Zoomed in one panel rendering generated by the standard Spark Ray Tracer in [8]. The moonlet is the larger sphere in the bottom center.

more specifically to each executor, that contains any of the `RDD`'s partitions.

Each partition exists solely on a single machine. They are the atomic unit of the RDD. Two to three partitions per CPU core is often best [4]. In both this standard Spark ray tracer as well as the photometric Spark ray tracer, we assign each partition one file of geometry, stored as a `KDTree`.

2.2 Methods

This Spark standard ray tracer was our first step in a larger journey towards rendering ray traced images of scenes with more geometry than can fit within the memory of a single

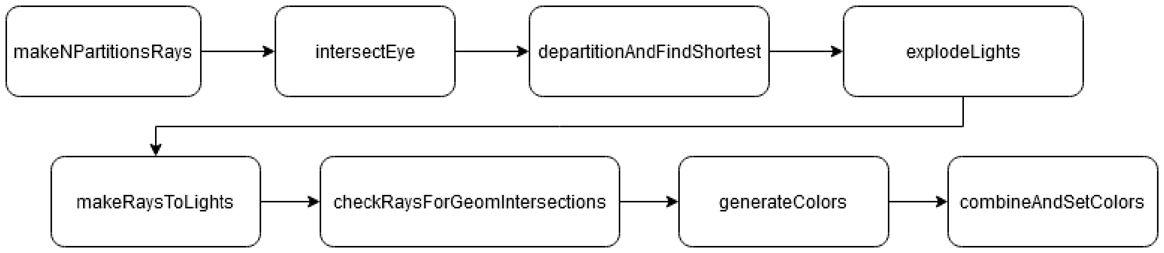

Figure 2.2: Diagram depicting the functional sequence of our Spark implementation of the standard ray tracing process.

machine. This is the only ray tracer we designed that utilized the standard process, and that was primarily because of our focus on distribution: we were more concerned with whether we could distribute the ray tracing process efficiently with Spark at all than we were with anything more than basic “extra work” done on the ray tracing.

As this was the first iteration, it started truly from scratch. We had access to Dr. Lewis’ `SwiftVis2` library for `Geometry` and `Ray` data types as well as some supporting functions (notably a function that intersects rays with geometry), but the large majority of the code was built from the ground up. The resulting iterations were also smaller steps than the later ray tracers, particularly the Akka photometric ray tracer. In the case of the Spark photometric ray tracer, this was because we were less experienced with the first ray tracer than the second (perhaps obviously). In the case of the Akka photometric ray tracer, this was also in part because Akka simply required more work to create a ray tracer than Spark (and thus the steps were necessarily larger).

For a detailed explanation of the intermediate iterations, please consult our prior publication entirely dedicated to this work, which discusses each iterative step [8]. In this work, we will only discuss the final product.

The final version of the Spark distributed ray tracer, from a one thousand foot view, distributes each panel of our geometry to each executor node in the cluster, then sends any generated rays to each section of geometry to determine intersection and whether points are lit, and thus how the pixels should be colored.

Describing the process more specifically, the initial rays are created from the camera such that they go outwards from the eye through the correct "pixel" in the screen. A copy of each of these rays is sent to every machine in the cluster to check for collisions against that machine's geometry. This sounds memory inefficient, but the memory requirements of the geometry scale much faster than the memory requirements of the rays, so this is viewed as a tolerable solution. After all, the thing that often exceeds the memory of a single machine (and that we are thus trying to distribute) is the geometry. It is important to remember that in some form, the rays must be checked in some way for collisions against every geometry (we use a KDTree to mitigate the cost of this somewhat, but at the core level it is still true: we must send every ray to every KDTree of geometry and thus if the geometry is spread across machines, the rays must be duplicated.) Each intersection is then stored as a Scala `Option` type of IntersectData, which includes information about where and when the intersection occurred. This information is then used to determine which intersection occurred first (and is thus the only significant intersection). From here, we want to determine if that point of significant intersection is going to be lit by the scene's lights. To do this, we create rays from each point towards the lights in the scene, duplicate them and send them to every panel of geometry, then check those rays for intersections and take the significant one the same way we did when finding the original intersections. If an intersection is found, we know the original point of intersection is shaded, if none is found we know the point is lit. This information is then used to determine how the appropriate pixel should be colored.

We'll now discuss in a closer to code fashion exactly how this ray tracer functioned. If you wish to actually view the code yourself, you should be able to find it on Dr. Lewis'

Github account (MarkCLewis) in the SparkRayTracer repository [22]. One note about this Spark ray tracer that is also true of the photometric Spark ray tracer is that it was designed very functionally. The entire process is done by passing the results of some functions into the inputs of others with very little outside mutation, and this was a benefit to the development and debugging of these ray tracers. The same cannot be said of our Akka implementation, though only because Akka simply doesn't work in the same fashion.

The `render` function is passed the geometry (an `RDD[KDTreeGeometry[BoundingSphere]]`) the scene's lights (List[PointLight]), the actual image to be drawn to and it's size, and the number of desired partitions (one for each file of geometry). It begins by calling `makeNPartitionsRays` with information about the number of partitions and how the camera is positioned and oriented. This is all the information required to generate rays from the camera into the scene, and then duplicate those rays for as many partitions as required by the application. This function outputs an `RDD[(Int, (Pixel, Ray))]`.

From there, the duplicated rays are joined with the geometry that has been passed in so that the intersections can be performed. Both of these `RDD`s contain tuple data types where the first element is an integer, the partition number. The result of this is that each geometry is paired with a copy of every ray. The datatype of this operation is `RDD[(Int, ((Pixel, Ray), KDTreeGeometry[BoundingSphere]))]]`.

Once the join is complete, the intersections can begin. The joined ray-geometries are passed into the `intersectEye` function, which performs the intersection between each ray and each `KDTree` of geometry. The output of this operation is an `RDD[(Int, (Pixel, (Ray, Option[IntersectData])))]`, where the leading `Int` is again the partition number, with it's associated pixel, and then the ray that was cast and it's associated `Option[IntersectData]`. The `IntersectData` contains information about the location and time of the collision, and the `Option` that contains it represents the possibility that there was no collision.

At this point, we have as many potential intersections of each ray as there were partitions of geometry. Some of their associated `Option` types may be empty, but there are invariably still cases where a ray hit geometry in multiple panels, and thus we need to determine which of the potential collisions should've happened first (and thus been most significant).

2.3 Results

Number of Geometry Panels	Time [sec]
10	105 ± 9
40	166 ± 5

Table 2.1: Benchmark timing results generated by the Spark standard (non-photometric) Ray Tracer on the Trinity University Pandora Cluster in [21].

Table 2.1 contains timing results generated by this Spark-distributed standard ray tracer running on the Pandora Cluster. Timing results for both 10 and 40 panel images (full square tilings of our geometry) were generated from the average of five runs and include a one standard deviation margin of error.

It is interesting to note that this ray tracer scales sub-linearly as the number of geometry files is increased. The exact cause of this behavior isn't known, but is attributed generally to the idea that Spark is more efficient when it's operating on larger data (within the bounds of the cluster's memory).

Chapter 3

Distributing Photometric Ray Tracing with Spark

This work is discussed specifically and with more focus in [21].

3.1 Photometry

Photometric ray tracing could be generalized as “regular ray tracing, but in reverse”. This isn’t quite correct (particularly for more realistic lighting behaviors), but it is true that in photometric ray tracing the rays begin with the light sources, which is often where the process ends for standard ray tracing. In photometry, rays are fired from the light sources into the scene, checked for collisions against the geometry, and then checked for visibility from the camera at the point of the intersection with geometry.

One key difference between photometric ray tracing and standard ray tracing is that the standard ray tracing process knows how many rays it needs to fire for a ”complete” image before it starts - the photometric process does not. This is because in the photometric process (with abstract geometry) it isn't possible to know which pixel a ray will correspond to until essentially the entire process for that ray is completed. In contrast, the standard ray tracer can fire exactly as many rays as there are pixels (or more if desired). The standard ray tracer is able to know exactly which pixel an outgoing ray will correspond to.

This difference is significant - it has performance implications (negative for the photometric process), but also allows for more realistic lighting behavior. One example of this is in diffuse lighting "bounces", which are impossible in a standard ray traced application [13].

Another distinction, less significant than the prior but still important, is that photometric ray tracing can be easily done in "batches" (and it often makes sense to do so, at least in our experience). If the geometry is quite large, it can be more time efficient to fire a large batch of photons, see how the image comes out, and then repeat if you'd like more detail or if you feel that not all of the appropriate geometry has been illuminated. Batching is certainly possible in a standard ray traced application (imagine generating two 1 ray per pixel images and then averaging their pixels together to get something akin to anti-aliasing), but likely less frequently implemented because standard ray tracers know they'll have a completed image when all of their rays are fired.

One final distinction, and perhaps the most important of them all, is that photometric ray tracing is much less efficient than its standard counterpart. This is because many of the rays fired in the photometric process yield collisions which are either completely obscured by closer geometry or are off-screen, and thus more rays are required than in the standard process. More rays simply means more computation, and thus more time to compute.

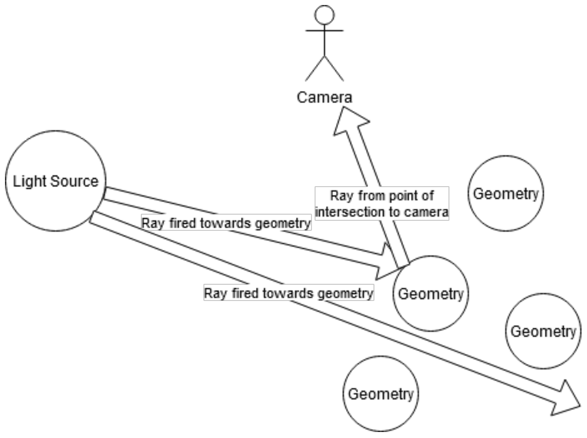
Camera
Light Source
Ray from point of intersection to camera
Geometry
Ray fired towards geometry
Ray fired towards geometry
Geometry
Geometry
Geometry

Figure 3.1: Diagram depicting the photometric ray tracing process.

3.2 Methods

This ray tracer was also created with an iterative development process, although the first iteration was actually the final version of the standard spark ray tracer. A lot of that code was modular and much of the logic and structure of the standard ray tracer could be shared with the photometric ray tracer.

The general process we followed in the photometric Spark ray tracer was thus fairly similar to the standard ray tracer. The geometry is created and setup identically. All of the scene parameters (camera location, coordinate system information, all the information associated with how the geometry coordinate system is related to the actual output image—also known as the view) are also .

The photometric Spark renderer is also similar to the standard Spark ray tracer in that it was designed relatively functionally. It follows the same style of using functions to generate the inputs of other functions with little outside mutation, and we reap the same rewards as in the standard Spark renderer (easier code maintenance, readability, expandability, modularity).

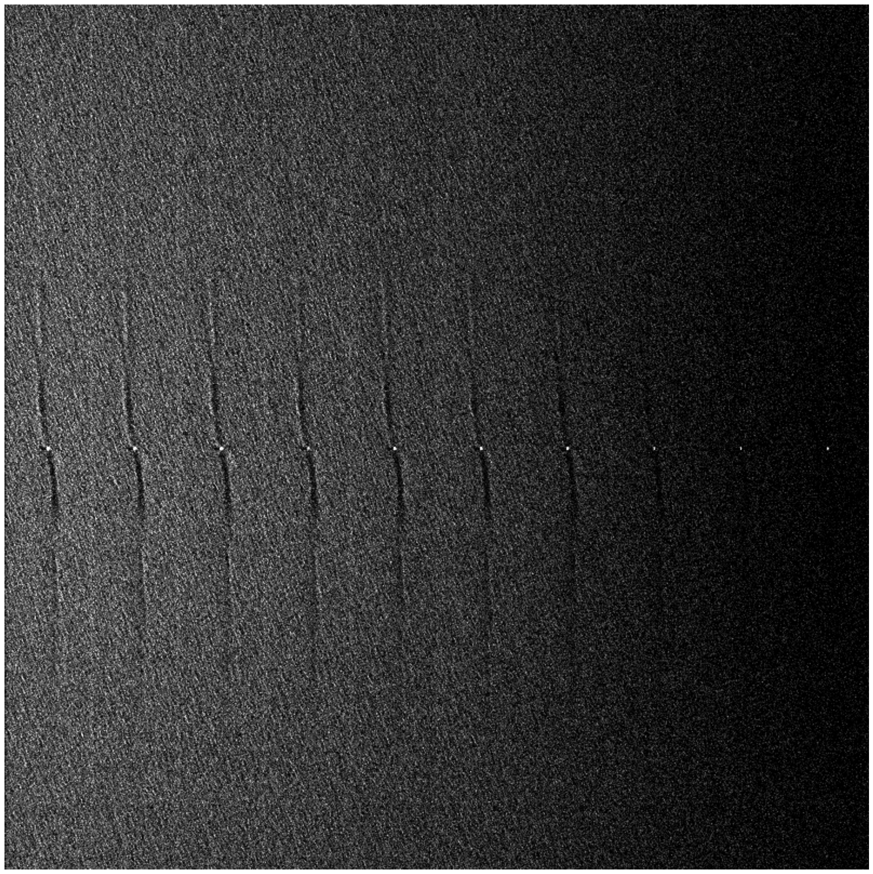

Figure 3.2: 24 million photon rendering of 10 panels generated in [21] by the photometric Spark ray tracer.

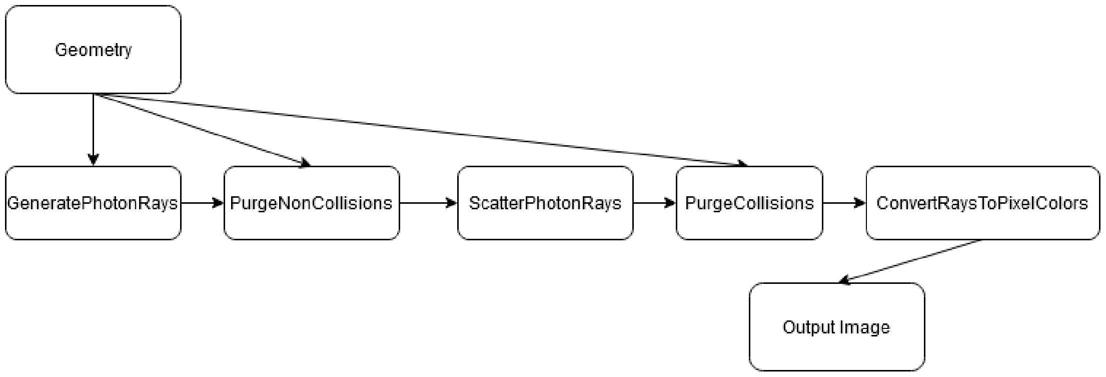

Figure 3.3: Diagram depicting our Spark implementation of the photometric ray tracing process.

This ray tracer is unique among the three discussed here in that it was specifically set up to run batches of photons. The Akka Photometric Ray Tracer could be expanded to include this functionality but it doesn't currently have it.

The Spark photometric ray tracer, in it's final form, begins by creating the `RDD` of Geometry. It's type is `RDD[(Int, KDTreeGeometry[BoundingSphere])]`, where the leading integer is a numeric key that determines what partition of the `RDD` the `KDTree` resides in.

This geometry, along with the lights, the image, the view, the size, and the number of partitions, is passed into a `Render` function. The render function begins a while loop that controls the batching. Within that while loop, five functions are called to do the photometric ray tracing process for the current batch.

The first of these functions is called `generatePhotonRays`. It accepts as arguments the lights in the scene as well as the `RDD` of geometry, and returns an `RDD[ColorRay]`, where `ColorRay` is a case class containing a color (of the light) and a ray. This RDD is generated by creating a number of rays from the light to each KDTree of the geometry. This approach is slightly different than the one in the Akka Photometric Ray Tracer, which randomly fires rays into the entire scene. This approach guarantees a certain uniformity of ray distribution amongst the panels of geometry. We haven't noticed a difference in output images, but it is possible that this approach is slightly more consistent with output images.

The output `RDD[ColorRay]` of `generatePhotonRays` is passed into a function called `purgeNonCollisions` along with the geometry RDD and the number of partitions. The goal of this function is to intersect the rays with each KDTree of Geometry and remove any rays that missed the geometry entirely. It is important to remember that even though we fired rays into specific `KDTrees` of the geometry, they still must be checked for intersections with the entire geometry. Thus, they are first duplicated for each of the `KDTree[Geometry]`, and then joined with the geometry such that we have an `RDD[(Int, (KDTreeGeometry[BoundingBox], ColorRay))]`, where the leading `Int` determines the partition, and a copy of each ray is tupled with each geometry.

At this point, the intersection can be performed, and non-collisions are filtered out. At this point, all that remains are the actual collisions, but we still might have rays that have collided with geometry in multiple `KDTrees`, and thus we need to see if there are multiple collisions, and which would've occurred first. Once this is determined, we discard the unnecessary rays and all that we have left is an `RDD[(ColorRay, IntersectData)]` that contains only rays that have collided with the geometry and even then only the significant collisions.

Then, the result `RDD` of the prior function is passed into yet another function, called `ScatterPhotonRays`, along with the location of the eye, or camera in the 3D rendering space. This function exists to first identify what quantity of light would've actually scattered towards the eye with that angle of collision, this is expressed as a percentage. Then, a `ColorRay` is generated containing a ray from the point of collision to the eye and the color as scaled by the scatter percentage. These ColorRays are generated in an RDD, which the function outputs as an `RDD[ColorRay]`.

At this point, the rays that collided with geometry before reaching the eye need to be removed. To accomplish this, we pass the prior function's `RDD[ColorRay]`, the geometry, and the number of partitions to the `purgeCollisions` function. This function is very similar to the `purgeNonCollisions` function, but not quite similar enough that we felt it necessary to abstract them to one generalized function.

This `purgeCollisions` function begins by grouping a copy of each ray with each panel of geometry, identically to `purgeNonCollisions`. Then, the intersections are performed. Again, we must now aggregate the duplicate rays sent to different geometry panels back into one place to determine which of them is significant. This time though, we aren't looking for one collision out of many to mark as significant, we're instead looking for duplicated rays where every duplicate missed it's panel of geometry. This is a slightly different bit of logic, but we essentially use an "or" folding operation on the duplicate rays to bubble any collision to the surface. This generates one "most significant" collision for each ray. If there was a collision, the ray is removed. If there was no collision, the ray is included in the output `RDD[ColorRay]`.

From here, we are left with an `RDD[ColorRay]` full of rays from points of light intersection with the scene towards the eye, that are unobstructed by geometry. All that remains to do is convert these rays to their pixel locations. This is a difference from the standard ray tracer, which already knows which pixel a ray corresponds to at this point. To do this, we pass the output RDD, the view, and the size of the image into the `convertRaysToPixelColors` function, which has a self-documenting name.

The `convertRaysToPixelColors` function essentially does some math to convert every `ColorRay` into a pixel location and a color. We won't discuss the actual math here, as it's pure ray tracing math, but checks are done to ensure that the ray is approaching from the front, and then that the pixel would be on the screen. This generates an `RDD[(Pixel,`

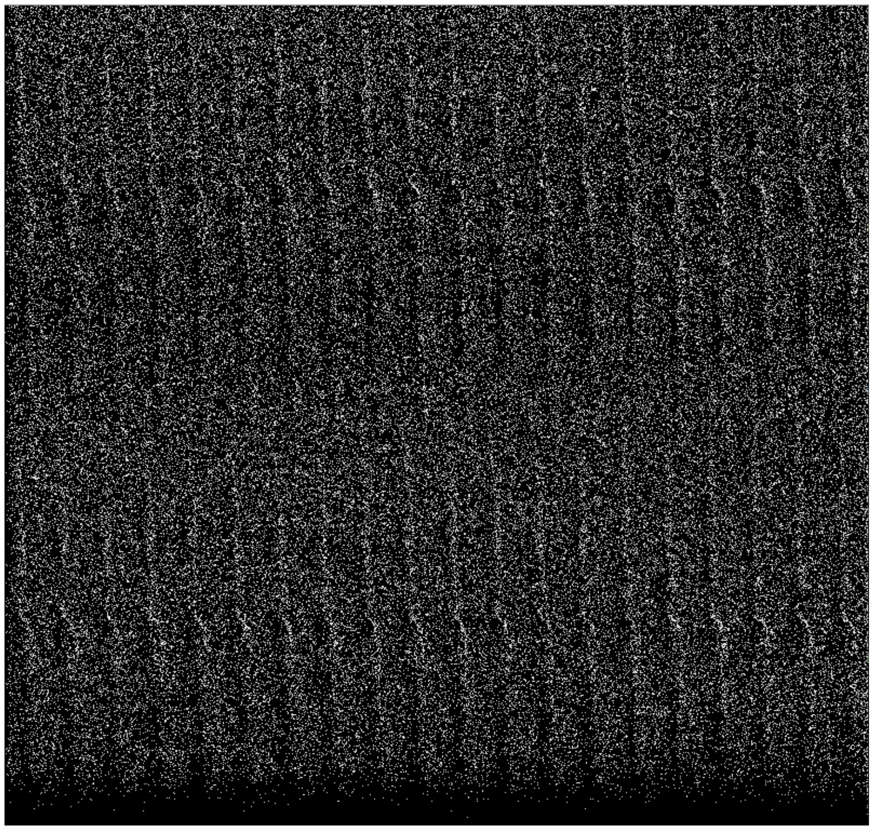

Figure 3.4: Rendering of 40 panels (2 rows of 20) created by the Spark photometric ray tracer.

`RTColor)]`, which will only ever be as large as the number of rays sent and is thus small enough to be “collected” (returned to single-machine, local memory). Once collected, these pixel-color tuples are iterated over and added to the image, and the process is complete, at least for this batch. At this point the user is prompted with whether they’d like another batch, and the ray tracer potentially begins this process again.

Total Number of Photons	Time [sec] (10 Panels)	Time [sec] (40 Panels)
4,000	657 ± 13	1038 ± 17
40,000	845 ± 15	1301 ± 10
400,000	1048 ± 20	1580 ± 29
4,000,000	1486 ± 31	2089 ± 26

Table 3.1: Benchmark timing results generated by the Spark photometric Ray Tracer in [21].

3.3 Results

Table 3.1 contains timing results generated by the Spark photometric ray tracer running on the Pandora cluster described in 1.4 at 10 panels and 40 panels of geometry, for [21]. The numbers displayed here were arrived at by averaging runtimes for five trials. A one standard deviation margin of error has been included.

It is again interesting to note that for both number of photons as well as number of geometry panels, the scaling is sub linear, and similarly to the standard Spark ray tracer, we don't have an exact cause of this but we suspect it to be a result of Spark being more efficient as the load is increased, up to a certain maximum where the limits of the memory begin to be tested.

Runtimes are also generally longer in similar photometric versions than standard, as more rays are required by the photometric renderer to achieve the same number of modified pixels. To be relatively confident that you've illuminated every pixel that should be illuminated, many many more rays are required than the standard ray tracer.

Chapter 4

Distributing Photometric Ray Tracing with Akka

A previous student worked quite extensively on creating an Akka Distributed Ray Tracer, and ultimately completed it [12]. This section is a continuation of his work, and works to bring his code up to the latest release of Akka such that other students or researchers may potentially continue upon it even further.

4.1 Akka

Kurt Hardee's prior ray tracer was written with a version of Akka now referred to as 'Akka Classic'. Akka has since released 'Akka Typed', which they suggest new projects should be started with [2]. The primary difference, although there are a few, between Akka Classic and Typed is that in Akka Typed the idea of an Actor has been replaced by a typed 'behavior'. What this means is that where you once had an Actor that could accept any message, you now have a 'behavior' that only accepts messages of a certain type. In practice, there is often a sealed trait for each Behavior that is extended by it's messages, which is passed as the type argument to the Behavior (to specify which message type it accepts).

This necessitates a large if not hugely structural change. The majority of the ray tracing logic in Kurt Hardee's original code could remain the same, but implementation details about how Actors were created, the structuring of the messages, and other associated details had to be changed. Every message that contained an ActorRef needed to be clarified (what kind of messages does that ActorRef accept?), and in some cases split into multiple messages or built with a polymorphic ActorRef message type if more than one type of ActorRef was previously being passed.

4.2 Methods

The primary goal in this section of the work was to create a distributed Akka Typed photometric ray tracer. This was a continuation of the work of a prior student, Kurt Hardee, who created a distributed Akka Classic photometric ray tracer [12]. We intended on essentially following the architecture he laid out but in Akka Typed, such that it could be built on further into the future when Akka Classic may not be as well supported. Ultimately, the ray tracer we created was a single-machine Akka Typed photometric ray tracer. I enjoy following an iterative development structure where multiple versions are created that build off of each other, and that is what was done here. The general process was as follows: first we created an Akka Typed photometric ray tracer that worked on small scenes of geometry, then we made it work with a single panel of geometry, and then we made it work with the full geometry (an arbitrary number of panels). The final step, which we began but did not complete, was to clusterize our existing single-machine Akka ray tracer. The conversion from a single machine application to a clustered application isn't terribly difficult in Akka, but it does require the creation of a number of new actors and interactions with system actors that didn't exist in Akka Classic (the receptionist, the subscriptions actor).

The problem of creating an Akka distributed photometric ray tracer for the large space scenes we desired was decomposed into three steps. First, to get an Akka actorized photometric ray tracer. This would be a single-machine application that would make use of Akka's actors to do the photometric ray tracing computation. Next, we wanted to get that working with our paneled geometry (much larger and structured slightly differently than the simple geometry). Finally, we intended to clusterize that process with Akka's Cluster functionality.

4.2.1 Initial Simple Geometry

In this step, we wanted to get the ray tracing logic Kurt Hardee had in his Akka Classic photometric ray tracer correctly working and running in Akka Typed. We were able to cut some corners as a result of having only a single 'panel' with a relatively very small amount of geometry, but a large portion of the code generated in this step would remain viable in the later steps. To keep things concise, we will only offer a detailed explanation of the workings of the Initial Simple geometry version, with notes on the changes required for the arbitrary number of geometry panels version.

When this was completed, we were able to feel more confident about the accuracy of the ray tracing code. This is one of the primary benefits of the iterative programming cycle. If you end each iteration with a satisfactory degree of confidence in the output code, you can sometimes limit the scope of future bugs (i.e. at the end of this step we felt confident that the ray tracing was being executed as desired, and that our output images matched our expectations. When we had bugs, we could guess that they were introduced in some form by code generated in the current iteration. It is also much harder to identify problems

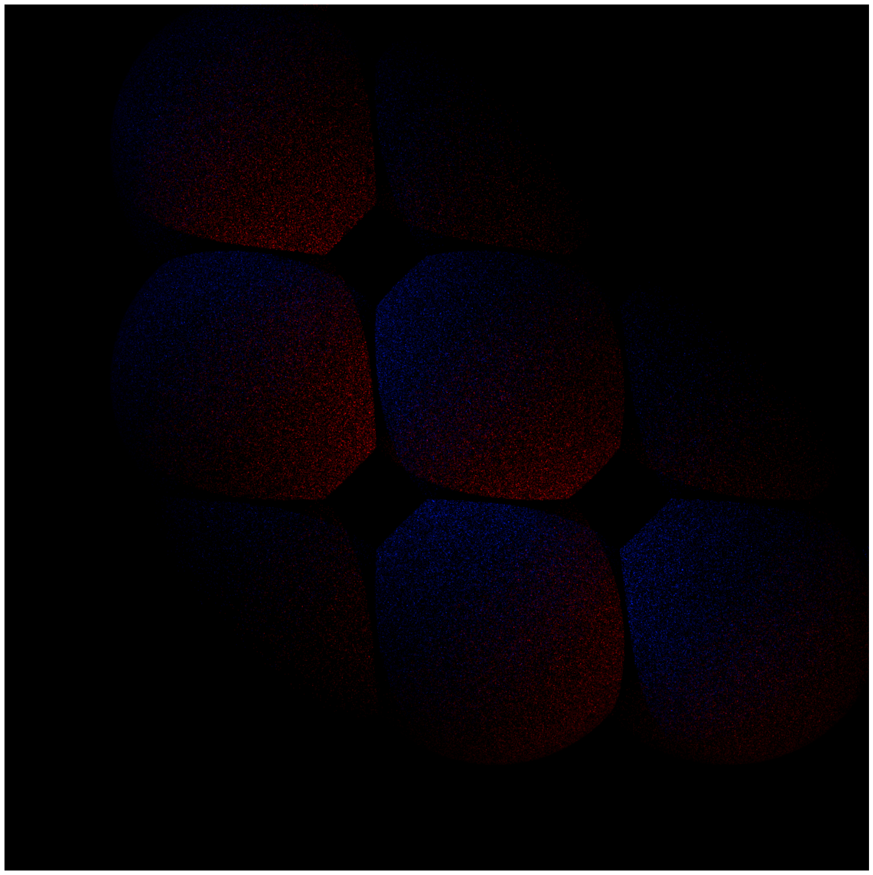

Figure 4.1: Akka Typed photometric ray traced rendering of simple geometry ("grid spheres" specifically) with red and blue lights.

in output images that contain millions of very small geometries. Development strategies aren't the primary focus of our research, but what we chose to do could be an interesting note for any future thesis students who read this work.

To be more specific, the geometries used in these simple geometry scenes were often slightly random, though later tests made use of some non-randomly placed geometries. We find that randomly placed geometry (within some limited space viewable by the camera) is effective for testing. The randomness helps to test more geometry configurations than a

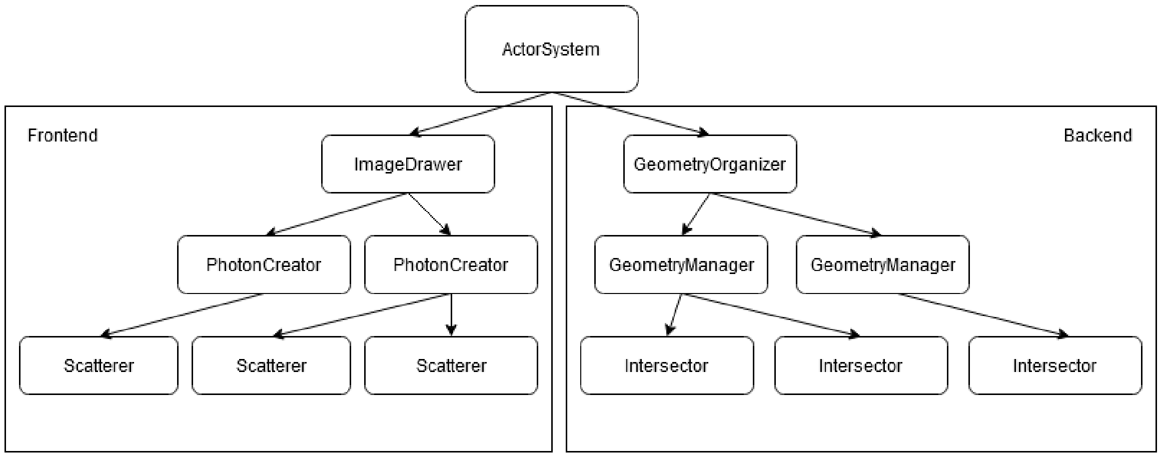

Figure 4.2: Actor hierarchy diagram for the Akka photometric ray tracer

non-random solution would, and when combined with non-random, preset geometry (like at the edges of the scene), makes for a good testing scene. Accurately rendering scenes like the one described here were the primary focus of the Simple Geometry iteration.

The primary difference between this version and the next were the view configuration and the lack of loading in a real geometry file. We separated these steps because we felt it important to have a working baseline of photometric ray tracing before we increased the geometry size. If we experienced bugs while getting the simple geometry version up and running, we could easily identify them, as each sphere was very large.

4.2.2 Single Geometry Panel

Completing this step required updating a number of files (or completing partial updates started in [12]) to Akka Typed. At the most basic level, this meant looking at Kurt's untyped code, taking out the actual bits of logic, and putting them into a new Akka Typed wrapper. Often, this was fairly straightforward, but occasionally the designs used in the untyped version weren't directly compatible with the model provided by Akka Typed and thus had to be modified. In general, we attempted to follow the same ray tracing procedures that Kurt used in the untyped version, and when we had to modify them, we created something as close to his original designs as we could in Akka Typed. The primary benefit of this was that the actual design of the code was already known to be working and viable, which removed some potential for error from our own endeavor.

The final product of this iteration begins by creating a `GeometryOrganizerSome`. This is the actor that will be managing the geometry. To be created, it requires the geometry for the scene (it also accepts an argument referred to as the `intersectResultMaker`, which is used for polymorphism with some of our messages).

An `ImageDrawer` is also created. This will handle the actual image itself as well as the creation of photons, which will be done by the `ImageDrawer`'s `PhotonCreator` children. Creation of the `ImageDrawer` requires the lights, the image, and information about the view and it's orientation in our 3D space.

On initialization, the `GeometryOrganizer` creates a `KDTree` of the geometry and assigns it to a `GeometryManager`. This step isn't expressly needed in this iteration, but for multiple panels of geometry it will be critical.

Each `GeometryManager` accepts `CastRay` messages and spreads them among a `Router` of `Intersectors`, which will check for intersections within whatever geometry the manager owns.

When the `ImageDrawer` is created, it first sends a message to the `GeometryOrganizer` to identify the bounds of the geometry. The `GeometryOrganizer` provides this information by combining the bounding boxes of all it's Geometry Managers into one larger box. When the `ImageDrawer` receives the bounds message, it remembers them and then sends itself a message to start ray tracing.

When this message is received, `PhotonCreators` are generated for each light source in the scene and sent their `Render` message. The light sources have an associated number of photons they emit, and this is passed into the PhotonCreator as well as information on the bounds of the geometry, the view, and the image (only used for it's width and height, could be optimized away).

When the `PhotonCreators` receive their `Render` message, they create as many rays as their associated light tells them to. These rays are from the light source into a random location within the bounds of the geometry. Each ray is then sent to the GeometryOrganizer to be distributed amongst the managers. This message containing the ray is called `CastRay`, and contains a reference to the `PhotonCreator` who is sending the message, an ID, and the ray itself.

When the `GeometryOrganizer` receives that `CastRay` message, it checks with it's managers to identify which of their outermost bounding boxes intersects the ray. If a manager's outermost bounding box doesn't intersect the ray, we know it is safe to not waste time computing that ray, as it definitely won't connect with any of the geometry in that manager. This decision-making process is referred to in [16] as the "Some" approach. It is listed there with the "All" and "Few" approaches, and performed the best in benchmarks. The primary difference between these approaches lies in which managers are sent which rays. "All" is perhaps self-explanatory in that every manager gets every ray. This is similar to our Spark implementations, but Akka can be more efficient. "Some" does what is described above, only sending the ray to the managers who have some possiblity of finding a real intersection. "Few" places the most burden on the `GeometryOrganizer`, requiring it to identify which collision should happen first, and then sending rays to geometry as it is determined they are necessary. Converting the "All" and "Few" modes to Akka Typed is something that we're leaving for future work, if it ever needs to be done.

Getting back to the ray tracing at hand, the managers that are found to potentially intersect with the ray are sent yet another `CastRay` message. It contains all the same data, but also a reference to the organizer itself.

When the `GeometryManager` receives a `CastRay` message, that message is piped directly into the `Router` of `Intersector`s mentioned earlier.

When one of the `Router`'s `Intersector` receives a `CastRay` message, it sends the `GeometryOrganizer` a message, referred to in the code as a `RecID` message (`RecID` short for received intersect data), with the original sender of the ray (the `PhotonCreator`, the ray's id, and the result of the intersection between the geometry and the ray.

When the `GeometryOrganizer` receives a `RecID` message, it doesn't immediately know if it should act on it. The `GeometryOrganizer` can only act on the results of the intersections once each `GeometryManager` has sent back the intersections for the ray it was assigned. Before this point, we don't know if missing intersections will be significant. So, it checks the id of the ray and consults with how many `RecID` messages it should've received for that ray. If not all have been returned, it continues to wait. If all have been returned though, it determines which of the potential collisions is the "real" collision. This intersection is then sent back to the `PhotonCreator`.

When the `PhotonCreator` receives this `IntersectResult`, it creates a `Scatterer` child actor to determine if that point is visible from the camera. When the `Scatterer` actor is created, it sends the `GeometryOrganizer` a `CastRay` message (same message type sent before in the PhotonCreator), but this time the ray is from the point of intersection to the eye, and the `Scatterer` is passed as the actor to send the final results back to. This triggers the same process executed before for the first collision check, but with a different ray.

When the `Scatterer` receives the `Option[IntersectResult]`, it only cares if it is empty. If it is not empty, that means the point is somehow obscured by some other geometry. If it is empty, that means the line between the camera and the point is unobstructed. If, further still, the point is actually within the bounds of the image, the `Scatterer` sends the `PhotonCreator` a `SetColor` message which contains the relevant pixel location as well as the color.

The `PhotonCreator` passes this message along to the `ImageDrawer` in the form of an `UpdateColor` message, which contains all the same data except the color is modified to include any prior coloring for that pixel.

When the `ImageDrawer` receives the `UpdateColor` message, it does so, and the pixel is added to the pixel buffer. When the pixel buffer reaches a certain size (100), it is pushed to the image.

This represents the end of one full process of Akka photometric ray tracing. Obviously, many of the processes described above are happening many many times and all at once, but this is the general flow they follow.

4.2.3 Arbitrary Number of Geometry Panels

The primary changes required in this iteration from the previous were to stop passing the geometry into the `GeometryOrganizer` (requires all of the geometry being on one machine at one time, incompatible with our goals), and to add some code that automatically places the view for any number of geometry panels.

The solution to the first problem was to modify the `GeometryOrganizer` to accept not the geometry itself, but how many panels of geometry were required. The `GeometryOrganizer` then downloads each file and gives it to it's associated `GeometryManager`. This could be further refined by having the `GeometryManager` download the file itself, but this first solution does solve the issue of having the geometry all in one place. As they aren't used at the same time, the garbage collector should be able to remove each panel after it is passed to the `GeometryManager`.

Prior to this version, the view was static. In this version, we introduced code (pulled from prior projects) [8], [16] to generate a view that places the panels in the 10*n panels in a row, n panels in a column configuration that generates our filled, square images.

With these two changes, we had a functioning single machine, Akka Typed, photometric ray tracer. The only thing missing is actually distributing it within a cluster. An interesting note is that in Spark there is no idea of a “single-machine” Spark program. Any Spark program can, with very little modification back and forth, run on any Spark compatible cluster as well as on any single machine. This is not the case with Akka, and thus some explicit clusterization must be performed.

4.2.4 Clusterization

This section discusses the process of clusterizing an Akka typed application. We did not finish this clusterization, but we believe the groundwork is there for future students to efficiently finish it

The Akka clusterization process is very different from its Spark counterpart. With Spark, essentially all of the cluster configuration is done on a by-hardware basis - i.e. you configure one cluster, it runs as many Spark programs as you want, with minimal configuration of those Spark programs. Akka, on the other hand, requires that the application itself (or an associated config file) is configured, and then cluster execution can be performed.

One other interesting difference between Spark and Akka is that clustered execution in Spark is done from one terminal on one computer. There is some ‘leader’ machine that accepts the command to begin execution, and then that leader machine takes care of setting up the workers. With Akka, the program must be run on each machine independently (those programs then look for each other and begin computation when ready). This also means that Akka users don’t necessarily have to rigidly follow the ”leader with workers” setup provided by Spark (though many certainly do).

As this work continues on work done by a previous student [12], where he successfully created an Akka Classic Distributed Ray Tracer, we feel confident that the general designs for clusterization expressed in that previous work are sound and could work in Akka Typed, with the required changes for the update. Our failure to clusterize the Akka Typed version is not due to any fundamental design issue, but rather a lack of time.

In the implementation begun and planned here, based off of Kurt Hardee's implementation, there would be a FrontendNode and a BackendNode. The FrontendNode is similar in function to a standard leader machine, and the BackendNode is similar to a worker machine. The FrontendNode creates actors for drawing the image as well as creating/organizing the geometry. The FrontendNode also looks for BackendNodes and begins the computation when they're all connected. The BackendNodes create n GeometryManager actors, one per geometry panel assigned to that machine (by the FrontendNode), and their associated child actors (Intersectors).

From there, the process moves very similarly to the single-machine version (the abstract geometry variation). The FrontendNode and BackendNode actors serve almost as Cluster-wrappers around the original code, and thus there isn't much modification to the actual logic. One important change, if not to the 'logic' but to the configuration, is the requirement for data to be serializable. Scala's case classes are Serializable by default, but if there is any need for more efficient serialization, there will be some configuration for that (specific to the serialization library).

Ultimately though, we were not able to complete the clusterization process within the required timeframe, and it is thus left as future work. There is some groundwork laid, and this planned structure of how clusterization should work is still very viable - but actually implementing it and testing it would've taken more time than we had available.

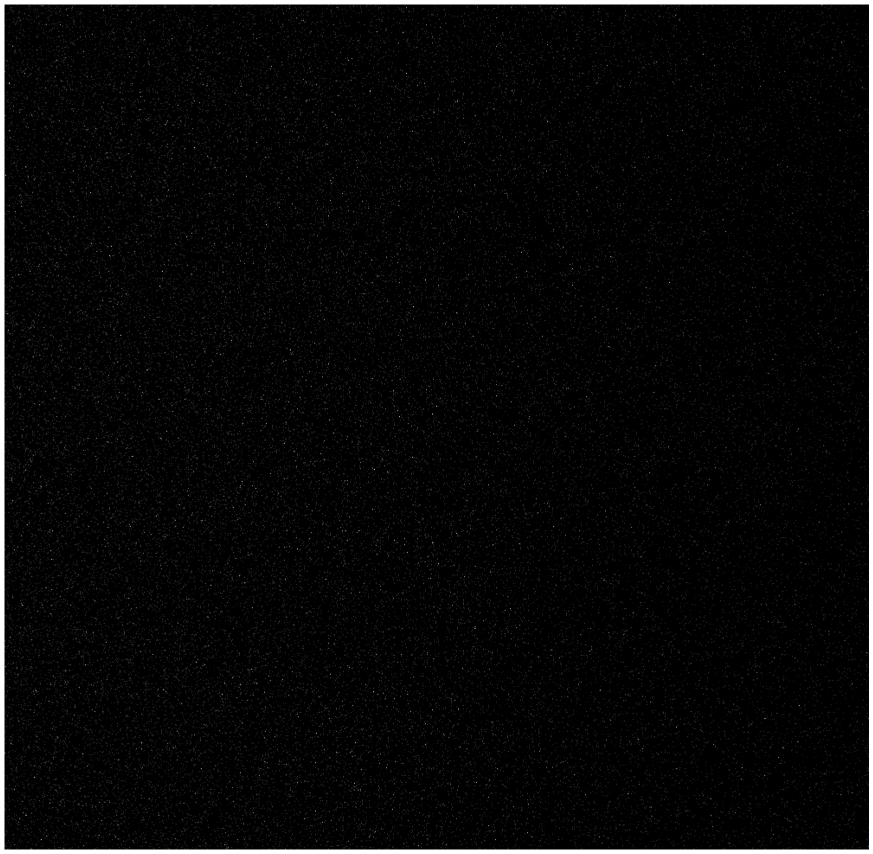

Figure 4.3: Akka Typed photometric ray traced rendering of 40 panels and 400,000 photons generated when creating the timing results seen in Table 4.1.

4.3 Results

Number of Geometry Panels	Time [sec]
10	108 ± 2
40	931 ± 43
90	DNF

Table 4.1: Timing results for the Akka Typed photometric ray tracer at varying numbers of geometry and 400,000 photons on a single machine in the Trinity University Pandora Cluster. DNF (did not finish) indicates the program ran out of memory and crashed.

The results contained in Table 4.1 are the runtimes for the Akka photometric ray tracer on a single machine, specifically Pandora02, one of the machines in the Pandora Cluster at Trinity University. They were created with samples of five runs, and include a margin of error of one standard deviation. It is important to note that these timing results are almost entirely incomparable with the results of the Spark sections, as those results are for distributed processes, and this ray tracer currently only runs on one machine.

One of the 40 panel images generated while creating timing results for Table 4.1 is shown in Figure 4.3. The moonlets aren't visible, but this is because each moonlet and it's associated propeller structures are quite small in a 40 panel image.

The geometry used in these tests is described in Section 1.3. Briefly, the test geometry is comprised of a number of distinct "panels" which are tiled such that they can create a square image for certain numbers of panels (notably 10, 40, and 90).

Chapter 5

Comparisons, Conclusions, and Future Work

5.1 Differences in Implementation

The primary differences in implementation between our Spark and Akka ray tracers can be divided into two camps: differences in the code required, and differences in the amount of control. Akka gives the developer more minute control over the transfer of data and messages, but this comes at a cost: the developer has to write the code to send and handle all of those messages.

This obviously isn't a fatal flaw with Akka. Frameworks do require some code to be written to use them, but the difference is significant. The single machine photometric Akka project uses at least nine explicitly referenced code files, with 6 different actors, weighing in at roughly 503 lines of relevant code. Clusterization would add at least two more actors and another layer of messages. For comparison, the photometric Spark project uses 2 primary code files and only 5 sequentially applied functions for the entire ray tracing process, and is roughly 400 lines long. The “lines of code” difference isn’t a fantastic metric, as the files aren’t machine-formatted, but the Spark version is already clusterized and certainly still smaller than the single machine Akka version. It’s also much less complex and easier to understand than the Akka version. Wrapping your head around five functions performed on `RDD`s likely takes less time than understanding the cadre of actors and their associated messages in the Akka version. This is a more subjective and qualitative observation, but it is our observation.

Another distinction is seen in the clusterization process. Akka requires explicit clusterization. A standard Akka (Classic or Typed) application does not run on a cluster [5]. A standard Spark application does. This could be a significant difference for prospective users of either framework because the clusterization process, while entirely feasible, does represent additional work that must be done before true completion. The importance of this difference is, in some ways, documented within this work. The Akka photometric ray tracer was not clusterized successfully. That wasn’t even a concern with the Spark ray tracers.

This is somewhat subjective, but we believe there is some core truth here in that Akka requires more work than Spark to get off the ground, and that future students working on future ray tracers should perhaps factor this into their calculations when deciding which distribution frameworks are more promising. Akka offers the promise and potential of additional control, but additional control isn’t very useful when your ray tracer still isn’t clusterized.

5.2 Recommendations for the Future

Contained within this thesis is discussion of three ideas for a distributed ray tracer and their associated implementations. It is difficult to make direct quantitative comparisons about their quality in this purpose (not everything can be boiled down to timing results even if we had that data), but what certainly can be said is that Akka allows for a finer degree of control than Spark, at the cost of increased complexity and quantity of code.

Thus, it would seem prudent to recommend that any endeavors towards a “final” ray tracer that will meet the original goals of these works be created in Akka, but it isn’t so straightforward. In an environment where at least a fourth but probably more of the student researchers leave (graduate) every year, there could be some real advantages to a Spark implementation. The amount of time required for a student to become “up to speed” on a prospective Spark ray tracer is probably less than that for the Akka ray tracer, assuming relative inexperience with both, and that could mean a significantly higher productivity for any students “picking up” the research of prior students.

Another recommendation, this one with more confidence, is that future researchers working on this project should probably implement a photometric model. It is more inefficient than the standard model, but as it mirrors the behavior of real light, it more easily supports scientifically realistic lighting behavior more easily than standard ray tracing.

5.3 Future Work

This thesis represents only a few steps towards the greater aims and needs of Dr. Lewis. Ultimately, Dr. Lewis needs a ray tracer capable of rendering scenes with very large amounts of geometry (more than can fit within the bounds of a single computer), and ideally with some additional features like animation support, light refraction through dust (practically impossible from a memory perspective to represent dust with the spheres used in this work), support for cloud computing resources.

An expansion to any of these ray tracers that is very interesting is cloud compatibility, or the ability to run on cloud machines and ideally with geometry data hosted in the cloud. We have experimented with this before, specifically with the Spark photometric ray tracer described in [21]. We were able to get that ray tracer running on AWS cloud resources, but we never took it further than a proof of concept. Taking this further would be excellent future work, because it would allow for a larger exploration of timing results and scaling behavior, and for simply larger scenes.

A full comparison between the Akka typed distributed photometric ray tracer and the Spark distributed photometric ray tracer must still be completed. This may be completed shortly after this paper is published, but in the event that it is not, it should be relatively straightforward for a future student to do. The only required tasks remaining are to finish the clusterization process started here with Akka Typed, and then compare that ray tracer with the Spark photometric ray tracer.

Another small bit of work that remains to be done is to add batching of photons to the Akka typed photometric ray tracer. Currently, it does the calculations for the initial quantity of photons with no option to add more or to run them in sequential batches. This isn't a major change, and a procedure to render more than one batches of photons was implemented in [21] to generate more visually interesting and realistic images. An implementation of photon batching in essentially any form will allow the Akka Typed photometric ray tracer to render images with more photons than it currently can.

Bibliography

[1] Actors • akka documentation.

[2] Akka classic • akka documentation.

[3] Akka home page.

[4] Rdd programming guide, Mar 2021.

[5] Cluster usage • akka documentation. https://doc.akka.io/docs/akka/current/typed/cluster.html, Feb 2022.

[6] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[7] James Bigler, Abe Stephens, and Steven G Parker. Design for parallel interactive ray tracing systems. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 187–196. IEEE, 2006.

[8] Erica Cater, Connor Weisenberger, and Mark C. Lewis. Distributed ray tracing of large scenes using spark. In *2020 International Conference on Computational Science and Computational Intelligence*, volume 7. IEEE CPS (+ IEEE Xplore, Scopus, ...), 2020.

[9] Per H. Christensen, Julian Fong, David M. Laur, and Dana Batali. Ray tracing for the movie 'cars'. *2006 IEEE Symposium on Interactive Ray Tracing*, 2006.

[10] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *Seminal graphics*, page 77–85, 1998.

[11] Apache Software Foundation. Apache spark™- what is spark. `https://databricks.com/spark/about`.

[12] Kurt D Hardee. *A Study in Akka-based Distributed Ray-tracing of Large Scenes.* Digital Commons @ Trinity, 2021.

[13] Marc. Raytracing / photon mapping.

[14] American Museum of Natural History. Planetarium show: Worlds beyond earth — amnh. `https://www.amnh.org/exhibitions/permanent/hayden-planetarium/worlds-beyond-earth`.

[15] Steven Parker, William Martin, Peter-Pike J. Sloan, Peter Shirley, Brian Smits, and Charles Hansen. Interactive ray tracing. *Proceedings of the 1999 symposium on Interactive 3D graphics - SI3D '99*, 1999.

[16] Elizabeth M. Ruetschle, Kurt D. Hardee, and Mark C. Lewis. Distributed ray tracing of large scenes using actors. In *Proceedings of the Symposium on Parallel & Distributed Computing (CSCI-ISPD).* IEEE Conference Publishing Services, 2020, accepted.

[17] Miodrag Sremčević, Jürgen Schmidt, Heikki Salo, Martin Seiß, Frank Spahn, and Nicole Albers. A belt of moonlets in saturn's a ring. *Nature*, 449(7165):1019–1021, 2007.

[18] Kelvin Sung, Jason Loh Jen Shiuan, and A.L. Ananda. Ray tracing in a distributed environment. *Computers & Graphics*, 20(1):41–49, 1996.

[19] Matthew S. Tiscareno, Joseph A. Burns, Matthew M. Hedman, and Carolyn C. Porco. THE POPULATION OF PROPELLERS IN SATURN's a RING. *The Astronomical Journal*, 135(3):1083–1091, feb 2008.

[20] Will Usher, Ingo Wald, Jefferson Amstutz, Johannes Günther, Carson Brownlee, and Valerio Pascucci. Scalable ray tracing using the distributed framebuffer. *Computer Graphics Forum*, 38(3):455 – 466, 2019.

[21] Connor Weisenberger and Mark C. Lewis. Distributed photometric rendering of large scenes using spark. In *2021 International Conference on Computational Science and Computational Intelligence*, volume 8. IEEE CPS (+ IEEE Xplore, Scopus, ...), 2021.

[22] Connor Weisenberger, Mark C. Lewis, and Erica Cater. Spark ray tracing. `https://github.com/MarkCLewis/SparkRayTracer`. Accessed: 2012-04-22.