juliacon

RayTracer.jl: A Differentiable Renderer that supports Parameter Optimization for Scene Reconstruction

Avik Pal[1]

[1]Indian Institute of Technology Kanpur

ABSTRACT

In this paper, we present RayTracer.jl, a renderer in Julia that is fully differentiable using source-to-source Automatic Differentiation (AD). This means that RayTracer not only renders 2D images from 3D scene parameters, but it can be used to optimize for model parameters that generate a target image in a Differentiable Programming (DP) pipeline. Our renderer is a complete general purpose renderer which means that unlike most previous work, we don't make any changes to the renderer to make it differentiable. Additionally we interface our renderer with the deep learning library Flux for use in combination with neural networks. In this paper, we also demonstrate the use of this differentiable renderer in rendering tasks and in solving inverse graphics problems using gradients.

Proceedings of JuliaCon

Keywords

Julia, Differentiable Programming, Automatic Differentiation, Inverse Graphics, Differentiable Rendering

1. Introduction

Rendering is a technique of generating photo-realistic or non-photo-realistic 2D projections from 3D objects. As such there are several algorithms for rendering complex scenes. One of the most popular techniques for photo-realistic rendering is ray tracing. However, for real-time rendering algorithms like rasterization are used.

Ray Tracing is a technique in computer graphics for rendering 3D graphics with complex light interactions. In this technique, rays are traced backward from the eye/camera to the light source(s). The ray can undergo reflection and refraction due to interactions with the objects in its path. This technique, however, is very computationally expensive and hence difficult to do in real-time. But since ray tracing leverages the properties of the materials of the objects in the scene, a natural extension to the rendering problem would be to extract the exact properties of the materials, lighting, etc. given an image of the scene. Calculating analytic gradients for every single parameter of the scene is a very tedious process and prone to errors. This has made it a difficult task to present a general gradient based inverse rendering method. As such, there is only one framework in our knowledge, redner [9], which has been able to do so by using analytic gradients. However, we bypass this problem by using AD. Another major limitation that exists in case of differentiability is the presence of geometric discontinuities, like the edge between two triangles. This makes the renderer non-differentiable at those points. Unfortunately this is still one of the limitations (Section 6) of RayTracer.

Rendering is a computationally expensive technique, and so it is generally done in static languages like C++. This makes it very time expensive to develop the software. Also, most languages lack the support of the state of the art automatic differentiation tools like Zygote [4], Jax [6], etc., which are generally implemented for high-level languages like Julia and Python. As such, it is difficult to develop differentiable renderers in those languages and then interface with popular deep learning software. The AD softwares present in other languages like CasADi [1] don't seamlessly integrate with packages, which means one needs to rewrite the software to use specific AD tools.

In this paper, we explore the idea of differentiability through a renderer, by leveraging the AD in Julia [2]. We present a fully general renderer capable of handling complex scenes and able to differentiate through them. We don't rely on analytic gradients but use source-to-source AD to generate efficient gradient code in the backward pass. Our renderer contains very little code for integration with Zygote and hence in theory we can plug in any other AD software written in Julia.

2. Differentiable Ray Tracing

There are several photo-realistic renderers available which contain a vast amount of implicit knowledge. Differentiation allows such renderers to make use of gradients to learn the inverse mapping from an image to its parameter space. We can then use this knowledge in combination with any machine learning / deep learning models to train them in a fast and efficient manner. Experiments in [3] demonstrate the effectiveness of incorporating differentiable renderers in deep learning pipelines by achieving State of the Art results in WIDERFace Hard dataset.

However, as usual, it is difficult to compute efficient derivatives from a production-ready renderer, typically written in a performance language like C++. This provides the primary motivation

Fig. 1: Utah Teapot Render from three different views. The camera definition shown in Listing 1 can be easily modified to generate all these views.

towards the development of RayTracer.jl. We develop an entire general purpose ray tracer in a high-level numerical computation language. The presence of strong automatic differentiation libraries like Zygote.jl make it trivial to compute efficient derivatives from the renderer.We present the performance gains we get on using Zygote as compared to Central Differencing in Section 5.2.

RayTracer.jl [10] is a package for Differentiable Ray Tracing written to solve this particular issue. It relies heavily on the source-to-source automatic differentiation package, Zygote, for computing gradients with respect to arbitrary scene parameters. This package allows the user to configure the location of objects, lights and a camera in the scene. This scene is then interpreted by the renderer to generate the image. RayTracer.jl is naturally interfaced with the deep learning library Flux [5], due to the common AD backend, for use in more complex differentiable pipelines.

3. Scene Rendering

Our approach to differentiable rendering is based on first creating a general purpose renderer and then make use of efficient AD tools. Hence, at its core RayTracer is a fully featured renderer. It contains functionalities for both raytracing and rasterization. Unlike most prior work in differentiable rendering we do not make performance compromises in the forward pass (rendering) to allow gradient computation.

RayTracer gives users a lot of control to the user over the scene they want to render. The user controls the lighting in the scene, the shape and materials of the objects and the camera configuration.

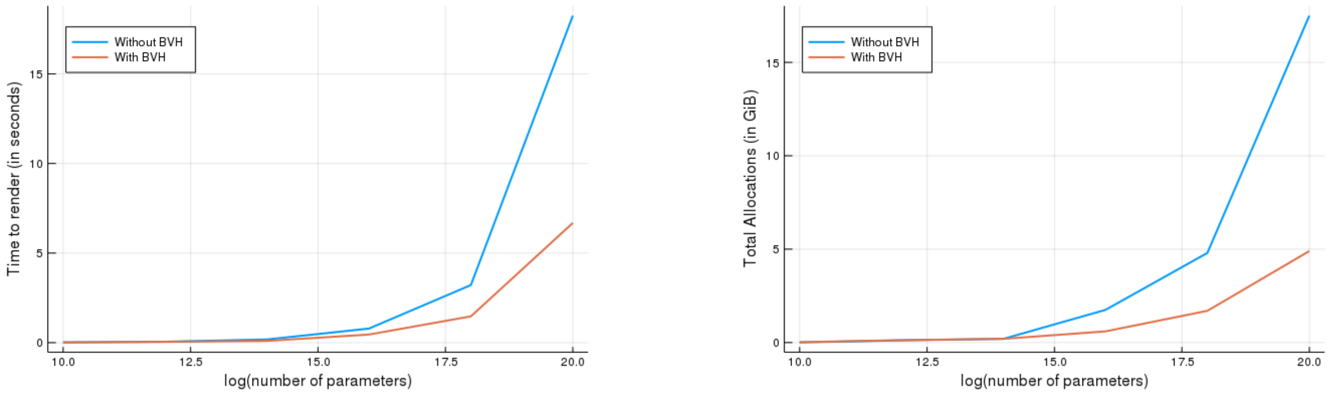
In this part we will demonstrate the general pipeline for defining a 3D scene using RayTracer.jl and then rendering it. We are going to render the popular Utah Teapot model. We need to specify the 3D model in the form of a Vector of Objects. We can do it manually for custom scenes or we could simply load it from a wavefront object (obj) file (support for additional file formats can be provided via MeshIO.jl). Apart from the scene vector, we need to specify the camera configuration and the configuration of light(s). We summarize the entire code to render the teapot in Listing 1.

```
# Screen Size
screen_size = (w = 512, h = 512)

# Camera Setup
cam = Camera(
    lookfrom = Vec3(1.0f0, 10.0f0, -1.0f0),
    lookat = Vec3(0.0f0),
    vup = Vec3(0.0f0, 1.0f0, 0.0f0),
    vfov = 45.0f0,
    focus = 1.0f0,
    width = screen_size.w,
    height = screen_size.h
)

origin, direction = get_primary_rays(cam)

# Scene
scene = load_obj("teapot.obj")

# Light Position
light = DistantLight(
    color = Vec3(1.0f0),
    intensity = 100.0f0,
    direction = Vec3(0.0f0, 1.0f0, 0.0f0)
)

# Render the image
color = raytrace(
    origin,
    direction,
    scene,
    light,
    origin,
    2
)
```

Listing 1: Rendering the Utah Teapot Model

4. Inverse Rendering

The rendering problem is to project 3D scene parameters to form an image in the 2D plane. Inverse Rendering problem is just the



(a) Performance Benchmarks **(b)** Memory Allocation Benchmarks

Fig. 2: Comparison between scenes rendered with and without BVH

opposite: generating a mapping from the 2D image back to the parameters of the 3D scene.

One of the primary motivations of RayTracer.jl is to solve the inverse graphics problem. Since the renderer can be used to compute the gradient with respect to any arbitrary parameter (as long as it is differentiable), we can then use any gradient based optimization technique to optimize on that parameter. This allows us to propose a generalized Algorithm 1 which is capable of optimizing any differentiable parameter.

Algorithm 1: Gradient Based Optimization of Scene Parameters

Result: Optimized set of Camera Parameters
Initial Guess of Parameters;
while *not* ***converged*** *or iter < max_iter* **do**
  gs = gradient (x → mean_squared_loss (render_image (x), target_img), params);
  for *param in params* **do**
    update! (optimizer, param, gs[param]);
  end
  if *loss < tolerance* **then**
    converged = True;
  end
end

5. Experiments

In this section we showcase our differentiable renderer in some benchmarking and toy inverse rendering problems. Using the following experiments we demonstrate the use of gradients obtained via AD to recover the camera, material and lighting parameters for a scene. In the inverse rendering experiments we make use of the Adam optimizer as described in [8]. We interface the raytracer with Flux to use these optimizers. As an alternative, we have also tested the functioning of our package with the optimizers present in Optim[1].

5.1 Accelerating the Rendering using Acceleration Structures

To accelerate the rendering process we support an acceleration structure, Bounding Volume Hierarchy (BVH)[7]. We follow the exact same API for ray tracing using these accelerators. In this case instead of passing a Vector of Objects we wrap it in a BoundingVolumeHierarchy object and pass it. So in order to use this, we would have to change the scene variable in Listing 1 to BoundingVolumeHierarchy(load_obj("teapot.obj")).

In this section we provide a comparison between the performance gains and memory allocation benefits of using BVH. We use a mesh of 137 triangles centered at the origin as the scene. We increase the screen size and hence increasing the total number of pixels (and therefore primary rays) in the scene[2].

We present the benefits of using BVH in Figure 2. We are able to get to reduce the total allocations (Figure 2b) and also get a significant performance boost (Figure 2a)[3]. Note that we only get exponential benefits in terms of memory and performance when the the number of pixels are of reasonable size, for example in case of 128×128 or better resolution images. For smaller images, using BVH might end up slowing down the rendering process.

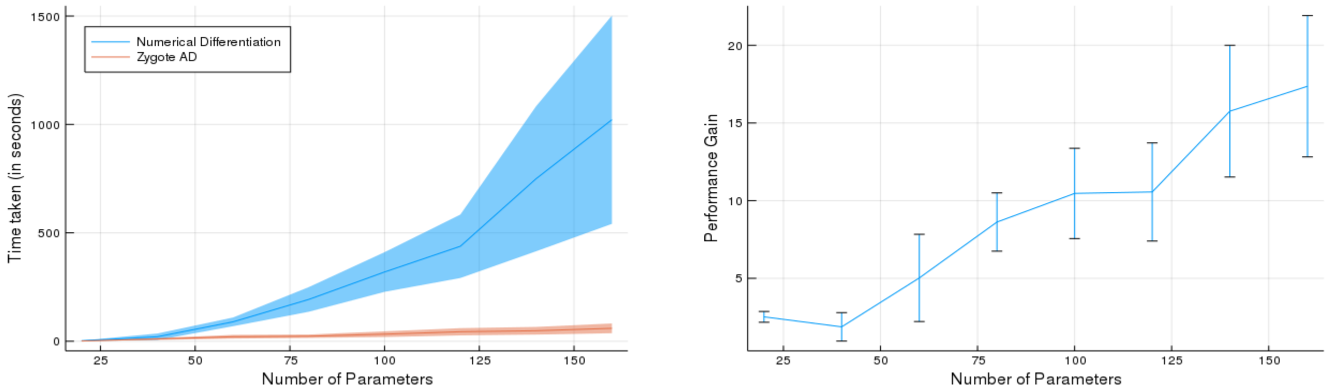
5.2 Comparison with Finite Differencing

In this section, we compare the use of our source-to-source AD backend with Central Finite Differencing, that is built into the package for gradient testing purposes. For comparing the two differentiation techniques, we run five independent trails and present the mean runtimes (with the standard deviation) in Figure 3.

[1] We provide an example at optim_compatibility.jl

[2] Even though it might seem that increasing the number of objects in the scene would be a better metric for comparison, it is immensely difficult to make that comparison. This is primarily because the same scene in a different configuration will have a different render time

[3] We provide the code for reproducing the experiment in performance_benchmarks.jl



(a) Time taken for the Backward Pass **(b)** SpeedUp when using AD vs Numerical Differentiation
Fig. 3: Comparison between Automatic Differentiation and Numerical Differentiation

We fix the position of the camera and light and randomly generate the triangles present in the scene. Every new triangle added to the scene, adds 20 additional parameters with respect to which we must compute the derivatives. We use the mean squared loss function to compute the scalar loss and backpropagate.

Figure 3b shows that we get significant speedups for a reasonable number of parameters. The nature of speedup shown suggests that it is nearly infeasible to use numerical differentiation when the number of parameters exceed 50. In most realistic applications of differentiable mesh rendering involving neural networks we will have atleast thousands of parameters. In such cases, our AD based solution will be able to compute the derivatives in reasonable time.

5.3 Calibration of Camera Parameters

In this experiment we start with the image of a rectangle (Listing 2) under some configuration of the Camera model (Listing 3). Since RayTracer supports only two primitive shapes - Spheres and Triangles, we need to triangulate the rectangle.

```
scene = [
    Triangle(
        Vec3( 20.0,  10.0, 0.0),
        Vec3( 20.0, -10.0, 0.0),
        Vec3(-20.0,  10.0, 0.0),
        Material(
            color_diffuse = Vec3(0.0, 1.0, 0.0))),
    Triangle(
        Vec3(-20.0, -10.0, 0.0),
        Vec3( 20.0, -10.0, 0.0),
        Vec3(-20.0,  10.0, 0.0),
        Material(
            color_diffuse = Vec3(0.0, 1.0, 0.0)))
]

light = PointLight(
    Vec3(1.0, 0.0, 0.0),
    100000.0,
    Vec3(0.0, 0.0, -10.0)
)
```

Listing 2: Configuration of the Scene for Experiment 5.3

```
camera_target =
    Camera(
        Vec3(0.0, 0.0, -30.0),
        Vec3(0.0, 0.0,   0.0),
        Vec3(0.0, 1.0,   0.0),
        90.0,
        1.0,
        screen_sizeCamera...
    )
```

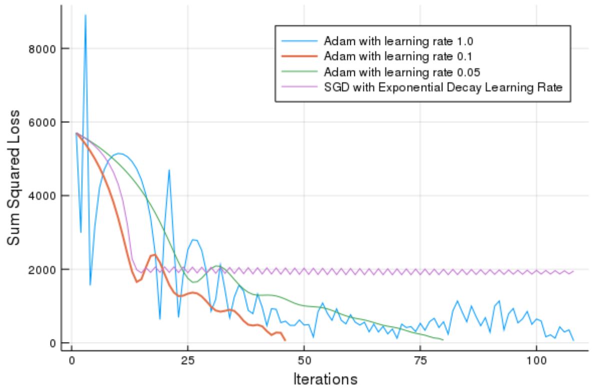
Listing 3: Camera Parameters to be Reconstructed

```
camera_guess =
    Camera(
        Vec3(5.0, -4.0, -20.0),
        Vec3(0.0,  0.0,   0.0),
        Vec3(0.0,  1.0,   0.0),
        90.0,
        3.0,
        screen_size...
    )
```

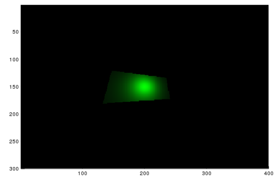
Listing 4: Initial Guess of the Camera Parameters

Our aim is to reconstruct the image of this rectangle (Figure 4c) by modifying the focus and the location of the camera. We assume that all the other parameters of the model, like the light configuration (Listing 2), position of objects, etc. are known apriori. We use algorithm 1 for optimizing the parameters. We make an initial guess of the parameters and initialize the camera (Listing 4).

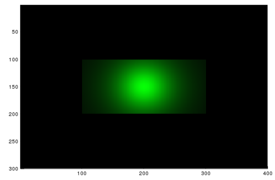
As our loss function, we use the mean squared difference between rendered and target images, each with 300×400 pixels having fractional RGB values. We minimize loss with the Adam optimizer, with learning rate 0.1, and declare the optimization to have converged if loss falls below 100 (where the initial loss is 5705.98). Figure 4 shows the optimization steps. We present the various learning rates and optimizers we experimented with in Figure 4a.



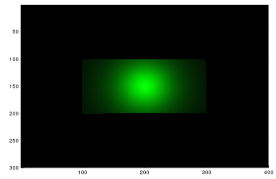
(a) Loss Values over the Light Configuration Optimization Process



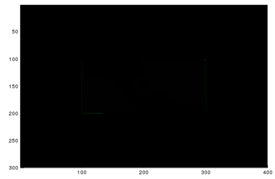
(b) Image with uncalibrated camera



(c) Image to be reconstructed



(d) Image obtained after the optimization



(e) Difference between reconstructed and target image

Fig. 4: Calibration of Camera Parameters to reconstruct Image 4c from Image 4b

5.4 Optimizing the Light Source

In this experiment we describe our solution to the inverse lighting problem. This problem involved predicting the configuration of the light source(s) in the scene given a target image (Figure 5b). All the other information about the geometry and surface properties of the objects and the camera configuration are already known. Listing 5 describes the known configurations.

```
screen_size = (w = 128, h = 128)

camera = Camera(
    Vec3(0.0f0, 6.0f0, -10.0f0),
    Vec3(0.0f0, 2.0f0,  0.0f0),
    Vec3(0.0f0, 1.0f0,  0.0f0),
    45.0f0,
    0.5f0,
    screen_size...
)

scene = load_obj("tree.obj")
```

Listing 5: Configuration of the Scene for Experiment 5.4

The object in our scene is a tree. We start with an arbitrary lighting (Listing 7) condition and then iteratively improve the lighting using Algorithm 1. We present the loss curve and the images generated during the optimization process in Figure 5.

```
light_target = PointLight(
    Vec3(1.0f0, 1.0f0, 1.0f0),
    20000.0f0,
    Vec3(1.0f0, 10.0f0, -50.0f0)
)
```

Listing 6: Target Lighting Conditions

```
light_guess = PointLight(
    Vec3(1.0f0, 1.0f0, 1.0f0),
    1.0f0,
    Vec3(-1.0f0, -10.0f0, -50.0f0)
)
```

Listing 7: Initial Guess of Lighting Conditions

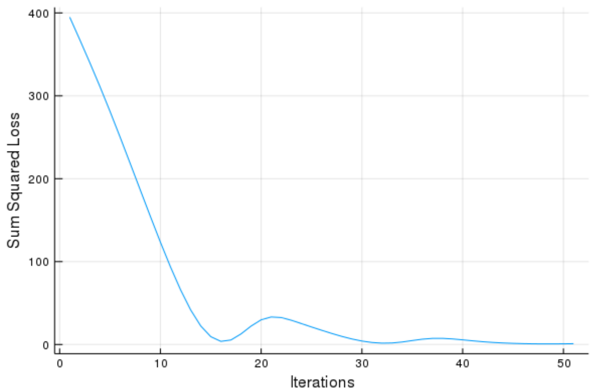
5.5 Retrieving Color of Materials

RayTracer.jl can also be used to recover the properties of the material of a mesh. For this experiment we shall use the same tree mesh from Experiment 5.4. We are going to optimize the diffuse color of the mesh. The position of the camera and the lighting conditions are mentioned in Listing 8.

The major difference of this optimization from the prior experiments is that the range of values the color can take is constrained between 0.0 and 1.0. Hence, after every step of the optimizer we need to clamp the value of the diffuse color for each mesh triangle.

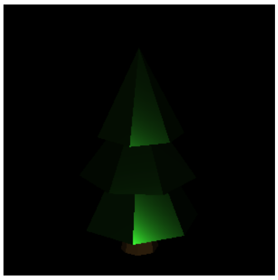
We use the sum squared loss function. We minimize the loss with Adam optimizer, with a learning rate of 0.05. The converge of the model is quite fast and we get a good approximation of the paramters just after a single optimizer step (Figure 6c).

```
screen_size = (w = 400, h = 300)

light = PointLight(
    Vec3(1.0f0),
    1000000.0f0,
    Vec3(0.15f0, 0.5f0, -10.5f0)
)

cam = Camera(
    Vec3(-2.0f0, 2.0f0, -5.0f0),
    Vec3( 0.0f0, 1.7f0,  0.0f0),
    Vec3( 0.0f0, 1.0f0,  0.0f0),
    45.0f0,
```



(a) Loss Values over the Light Configuration Optimization Process

(b) Image to be reconstructed

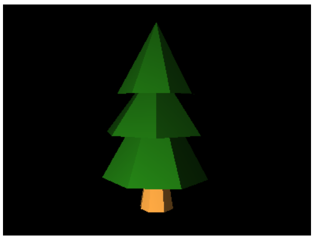


(c) Initial Guess of Lighting Parameters

(d) Image produced after 10 iterations

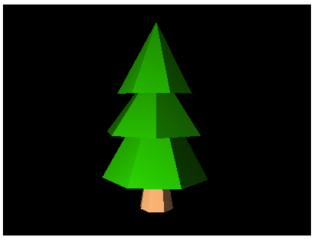
(e) Image with converged parameters

Fig. 5: Optimization of the lighting conditions to reconstruct Image 5b from Image 5c

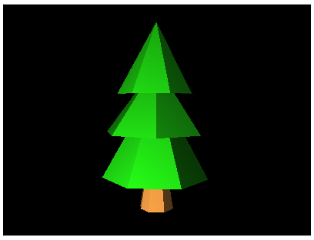


(a) Image to be reconstructed

(b) Initial Guess of the Materials



(c) Image produced after just 1 iteration



(d) Image obtained after 35 iterations

Fig. 6: Optimization of the materials of the mesh to reconstruct Image 6a from Image 6b

```
    1.0f0,
    screen_size...
)

scene = load_obj("tree.obj")
```

Listing 8: Configuration of the Scene for Experiment 5.5

6. Current Limitations

Despite the success of our approach in solving a variety of inverse graphics problems, we fail to deal with problems that are non-differentiable in nature. One such instance would be estimating the proper geometry of an object given an image. Such problems are non-differentiable due to the large number of discrete choices in the position of the triangle vertices. Hence, trying to optimize such parameters generally causes them to diverge. The other cases which we can't handle properly are secondary lighting (shadows) and global illumination. Most of these cases can be dealt with, similar to the way proposed by [9], but that leads to a significant slow-down to the rendering of the scene.

7. Conclusion

In conclusion, we have showed how Julia can be leveraged to build differentiable systems. We have presented which to the best of our knowledge is the first differentiable renderer which uses source-to-source AD. We have used a set of toy examples to demonstrate the ability of our renderer to reconstruct scenes (which are differentiable) from only a single image. This also shows that this renderer can be used in differentiable programming pipelines which involve image generation.

8. References

[1] Joel A E Andersson, Joris Gillis, Greg Horn, James B Rawlings, and Moritz Diehl. CasADi – A software framework

for nonlinear optimization and optimal control. *Mathematical Programming Computation*, In Press, 2018.

[2] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[3] Jiankang Deng, Jia Guo, Yuxiang Zhou, Jinke Yu, Irene Kotsia, and Stefanos Zafeiriou. Retinaface: Single-stage dense face localisation in the wild. *CoRR*, abs/1905.00641, 2019.

[4] Michael Innes. Don't Unroll Adjoint: Differentiating SSA-Form Programs. *CoRR*, abs/1810.07951, 2018.

[5] Michael Innes, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal, and Viral Shah. Fashionable Modelling with Flux. *CoRR*, abs/1811.01457, 2018.

[6] Matt Johnson, Roy Frostig, Dougal Maclaurin, and Chris Leary. JAX: Autograd and XLA. `https://github.com/google/jax`, 2018.

[7] Timothy L. Kay and James T. Kajiya. Ray Tracing Complex Scenes. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 269–278, New York, NY, USA, 1986. ACM.

[8] Diederik Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations*, 12 2014.

[9] Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. Differentiable Monte Carlo Ray Tracing through Edge Sampling. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)*, 37(6):222:1–222:11, 2018.

[10] Avik Pal. RayTracer.jl. `https://github.com/avik-pal/RayTracer.jl`, 2019.