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. We interface our renderer with the deep learning library Flux for use in combination with neural networks. We demonstrate the use of this differentiable renderer in rendering tasks and in solving inverse graphics problems.

Proceedings of JuliaCon

Keywords

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

1. Introduction

Rendering is a technique of generating photorealistic or non photorealistic 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 backwards from the eye/camera to the light source(s). The ray of 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. Unfortunately ray tracing is non-differentiable at some points and calculating the analytic gradients is a very tedious task. This has made it a very difficult task to present a general inverse rendering method. As such there is only one framework in our knowledge, redner[8], which has been able to do so by using analytic gradients.

Rendering being a computationally expensive operation 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 state of the art automatic differentiation tools like Zygote [4], Jax [1], 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.

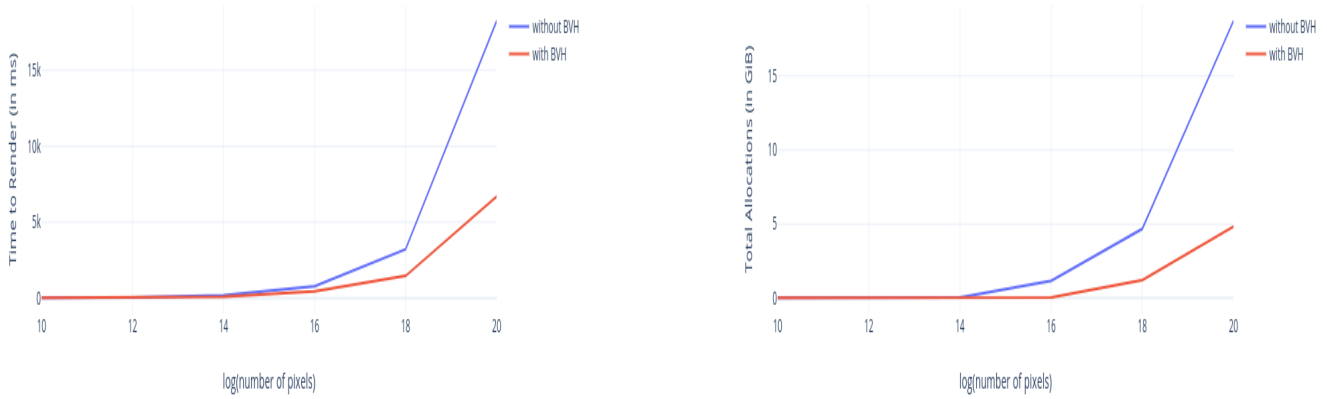
In this paper, we explore the idea of differentiability through a renderer, by leveraging the AD in Julia[2]. We present a fully general renderer capable of handling complex scenes and able to differentiate through them. We don't rely on analytic gradients but use source-to-source AD to generate efficient gradient code in the backward pass.

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. This knowledge can then be used 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 incorporate 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 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.

RayTracer.jl [9] 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.



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

Fig. 1: Comparison between scenes rendered with and without BVH

Fig. 2: Utah Teapot Render from three different views. The camera definition shown in Listing 1 can be easily modified to generate all these views.

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 for the differentiability part. Hence, at its core RayTracer is a fully featured renderer. It contains functionalities for both raytracing and rasterization. Additionally, 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.

3.1 Accelerating the Rendering Process

To accelerate the rendering process we have acceleration structures. Currently only one such acceleration structure, Bounding Volume Hierarchy [6], is supported. 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 to *BoundingVolumeHierarchy(load_obj("teapot.obj"))*.

Clarifications and Reproducibility

We show the benefits of using Acceleration Structures in Figures 1a and 1b. As can be seen from the plots we not only get a good

performance gain (Figure 1a) in terms of speed, we also end up allocating much less memory (Figure 1b on using Bounding Volume Hierarchy.

3.2 Rendering the Utah Teapot

Clarity and Grammer

In this section we shall render a very popular model in computer graphics: the Utah Teapot. We simply load the model from a wavefront object (obj) file. Support for other types of files is trivial through the MeshIO.jl package with defines file reader for common mesh object files. Once the scene is loaded the next step is to configure the other elements in the scene - the camera and the light(s). Script 1 illustrates the code for rendering the teapot model.

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

# Camera Setup
cam = Camera(
    Vec3(1.0f0, 10.0f0, -1.0f0),
    Vec3(0.0f0),
    Vec3(0.0f0, 1.0f0, 0.0f0),
    45.0f0,
    1.0f0,
    screen_size...
)

origin, direction = get_primary_rays(cam)

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

# Light Position
light = DistantLight(
    Vec3(1.0f0),
    100.0f0,
    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

More clarity in writing

Just like rendering can be thought of as the projection of 3D objects into a 2D plane, the inverse rendering problem can be described as just the opposite. It is the mapping of the 2D image back to the parameters of the scene.

As we have mentioned previously, one of the primary motivations of RayTracer.jl is for solving the problem of inverse graphics. Being able to compute gradients wrt any scene parameter, means that we can optimize that parameter. The optimization algorithm is pretty straight forward and is general enough to work for any arbitrary parameter. We describe the algorithm in 1.

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(params) **do**
        img = rendered image with params;
        loss = mean_squared_loss(img, target_img);
    end;
    for *param in params* **do**
        update!(optimizer, param, gs[param]);
    end
    if *loss* < *tolerance* **then**
        converged = True;
    end
end

5. Experiments

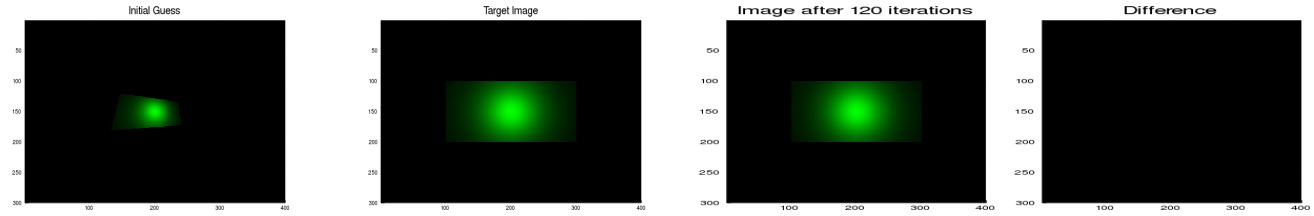
In this section we showcase our differentiable renderer in some toy inverse rendering problems. Using the following two experiments we demonstrate the use of gradients obtained via AD to recover the camera and lighting parameters fr a scene. In both the experiments we make use of the Adam optimizer as described in [7]. We interface the raytracer with Flux to use these optimizers. As an alternative, we have tested the functioning of our package with the optimizers present in Optim.

5.1 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.

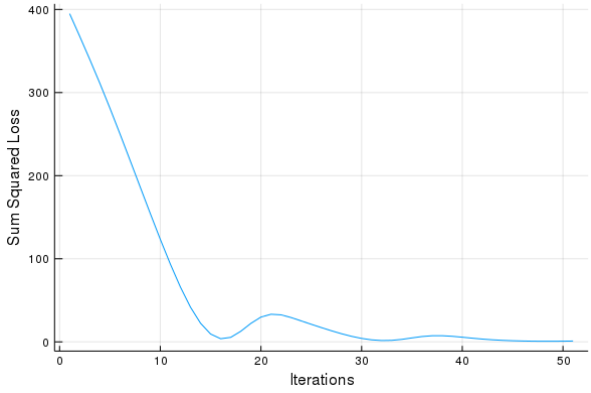
```
scene = [
    Triangle(
        Vec3( 20.0,  10.0, 0.0),
        Vec3( 20.0, -10.0, 0.0),
        Vec3(-20.0,  10.0, 0.0),
        Material(
            color_diffuse = Vec3(0.0, 1.0, 0.0))),
    Triangle(
        Vec3(-20.0, -10.0, 0.0),
        Vec3( 20.0, -10.0, 0.0),
        Vec3(-20.0,  10.0,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

Listing 2: Configuration of the Rectangle for Experiment 5.1



(a) Image with uncalibrated camera **(b)** Image to be reconstructed **(c)** Image obtained after optimization

Fig. 3: Calibration of Camera Parameters to reconstruct Image 3b from Image 3a



(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. 4: Optimization of the lighting conditions to reconstruct Image 4b from Image 4c

```
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_size...
    )
```

Listing 3: Camera Parameters to be Reconstructed

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

Listing 4: Light Configuration

Our aim is to reconstruct the image of this rectangle (Figure 3b) 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 4), 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 5).

```
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 5: Initial Guess of the Camera Parameters

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 0.05, and declare the model to have con-

verged if loss falls below 10 (where the initial loss is in the order of 600). Figure 3 shows the optimization steps.

5.2 Optimizing the Light Source

More details to reproduce the results

In this experiment we describe our solution to the inverse lighting problem. In this case we assume the knowledge of the geometry and surface properties of the objects, camera position and the target image of the scene.

For this experiment we try to optimize the lighting for a scene containing a tree. We are given the target image with proper lighting conditions. We start with an arbitrary lighting and then iteratively improve the lighting using algorithm 1.

We present the images generated during the optimization process in Figure 4.

6. Current Limitations

Write this section highlighting the problem for non differentiable parts

7. Conclusion

Needs a rewrite using the comments in the review

In conclusion, we have presented how julia can be leveraged to build differentiable systems. Integrating them with the machine learning pipeline can make an end-to-end differentiable pipeline. Using this pipeline we can define differentiable programming algorithm on it to solve the problem. As expected making the pipeline differentiable allows us to exploit the huge amount of implicit knowledge stored in the system.

8. References

[1] Jax. https://github.com/google/jax, 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] 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.

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

[8] 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.

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