juliacon

RayTracer.jl: A General Purpose Differentiable Renderer

Avik Pal[1]

[1]Indian Institute of Technology, Kanpur

ABSTRACT

With the rising popularity of deep learning techniques, being able to differentiate through arbitrary functions is becoming more and more important. These arbitrary functions contain a lot of implicit knowledge stored in them which when used in combination with a neural network can significantly speed up its training. In this paper we present a fully differentiable renderer, which is designed to be integrated in any 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 complex rendering tasks and show its use in solving inverse graphics problems.

Proceedings of JuliaCon

Keywords

Julia, Differentiable Programming, Deep Learning, Automatic Differentiation, Inverse Graphics

1. Introduction

In Computer Graphics, a popular technique for rendering photo realistic images is raytracing. 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. However, since ray tracing is non-differentiable at some points and calculating the analytic gradients is a very tedious task, it has been difficult to present a general inverse rendering method. As such there is only one framework in our knowledge, redner[7], which has been able to do so by using analytic gradients.

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. Background

Rendering is a technique of generating photorealistic 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 ~~is~~ 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). ~~In the path,~~ the ray ~~of light~~ can undergo reflection and refraction due to interactions with the objects in its path. ~~This method~~, however, is very computationally expensive and hence difficult to do in real time.

Rendering being a compute intensive operation is generally done in static languages like C++. This makes it very time expensive to develop the software. Also, these languages lack support ~~of~~ state of the art automatic differentiation tools like Zygote [3], Jax [1], etc which are generally implemented for 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.

3. 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 model to train them in a fast and efficient manner.

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 [8] 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

Summary of Comments on 10.21105.jcon.00031 (1).pdf

Page: 1

Author: snoeyink Subject: Sticky Note Date: 7/9/2019 2:37:21 PM

The package title mislead me: I was expecting a RayTracer, which solves a graphics problem, but the author's emphasis is on inverse graphics (a form of scene understanding.) I'd suggest adding to the paper title: RayTracer.jl: A Differentiable Renderer that Supports Parameter Optimization for Scene Reconstruction.

Author: snoeyink Subject: Sticky Note Date: 7/9/2019 2:41:32 PM

This paper introduces a raytracer built in Julia that allows the Zygote Automatic Differentiation tool to compute gradients of parameters. One can then optimize parameter values for scene understanding or reconstruction.

The examples shown are toy experiments of obtaining camera or light parameters to match a rendered image. They demonstrate that the combination works, which is good.

I would like to see some discussion of more interesting parameters, such as determining material properties from images. And more on limitations: determining geometry seems out of reach because there are so many discrete choices: the problem is not differentiable.

How fast is the gradient computation after AD? (How fast is the AD?) How do these grow with scene complexity?

Author: snoeyink Subject: Highlight Date: 7/9/2019 4:17:40 PM

The abstract really threw me for a loop -- it starts out like it is for a completely different paper. You will not be doing any experiments that demonstrate significant training speedups, so that should not be the start of the abstract.
I suggest something like this:

In this paper we present RayTracer.jl, a renderer in Julia that is fully differentiable via Zygote.jl. 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.

Author: snoeyink Subject: Highlight Date: 7/9/2019 4:05:49 PM

This is somewhat repetitive.

Author: snoeyink Subject: Inserted Text Date: 7/9/2019 4:24:13 PM

are
[better: avoid passive:: However, real-time rendering use algorithms like rasterization.]
You provide rasterization, too, so this way of presenting the background is strange. Most of this background is said elsewhere, so I'm not sure you need this section. Move the content into the next.

Author: snoeyink Subject: Cross-Out Date: 7/9/2019 4:25:24 PM

You haven't mentioned path yet, so "the path" confuses.

Author: snoeyink Subject: Cross-Out Date: 7/9/2019 4:25:02 PM

The ray is from the eye, so calling it a ray of light is confusing.

Author: snoeyink Subject: Highlight Date: 7/9/2019 4:17:56 PM

You don't demonstrate any complex rendering tasks.

Author: snoeyink Subject: Inserted Text Date: 7/9/2019 4:26:35 PM

was Technique -- don't change terms.
Ray tracing is computationally expensive

Author: snoeyink Subject: Highlight Date: 7/9/2019 4:30:32 PM

You are changing terms
computationally expensive = compute intensive
operation = technique?
And are you going back to general rendering or still referring to ray tracing?

Since rendering is computationally expensive, it is generally programmed in static languages like C++, making software development time expensive. Static languages also lack support for

Author: snoeyink Subject: Highlight Date: 7/9/2019 4:31:18 PM

antecedent of these (plural) is unclear, since you've only named one, C++.

Comments from page 1 continued on next page

juliacon

RayTracer.jl: A General Purpose Differentiable Renderer

Avik Pal[1]

[1]Indian Institute of Technology, Kanpur

ABSTRACT

With the rising popularity of deep learning techniques, being able to differentiate through arbitrary functions is becoming more and more important. These arbitrary functions contain a lot of implicit knowledge stored in them which when used in combination with a neural network can significantly speed up its training. In this paper we present a fully differentiable renderer, which is designed to be integrated in any 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 complex rendering tasks and show its use in solving inverse graphics problems.

Proceedings of JuliaCon

Keywords

Julia, Differentiable Programming, Deep Learning, Automatic Differentiation, Inverse Graphics

1. Introduction

In Computer Graphics, a popular technique for rendering photo realistic images is raytracing. 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. However, since ray tracing is non-differentiable at some points and calculating the analytic gradients is a very tedious task, it has been difficult to present a general inverse rendering method. As such there is only one framework in our knowledge, redner[7], which has been able to do so by using analytic gradients.

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. Background

Rendering is a technique of generating photorealistic 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 ~~is~~ 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). ~~In the path,~~ the ray ~~of light~~ can undergo reflection and refraction due to interactions with the objects in its path. ~~This method,~~ however, is very computationally expensive and hence difficult to do in real time.

Rendering being a compute intensive operation is generally done in static languages like C++. This makes it very time expensive to develop the software. Also, these languages lack support ~~of~~ state of the art automatic differentiation tools like Zygote [3], Jax [1], etc which are generally implemented for 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.

3. 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 model to train them in a fast and efficient manner.

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 [8] 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

Author: snoeyink Subject: Inserted Text Date: 7/9/2019 4:31:23 PM
for

Author: snoeyink Subject: Highlight Date: 7/9/2019 4:33:03 PM
What is the support for this claim? How does the non-differentiability of model-building decisions limit it?

Author: snoeyink Subject: Highlight Date: 7/9/2019 4:07:06 PM
You never come back to this issue. How does this affect what parameters you can optimize for?

Author: snoeyink Subject: Inserted Text Date: 7/9/2019 4:08:37 PM
Automatic Differentiation (AD)
[always spell out an acronym the first time you use it.]

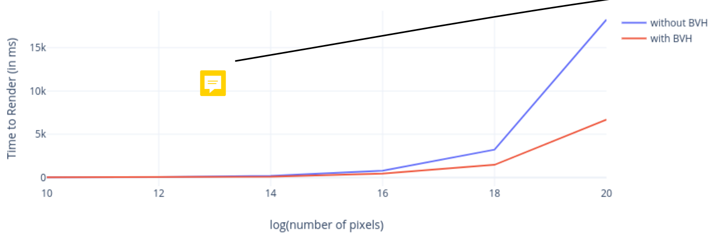


Fig. 1: Performance Gains on using BVH

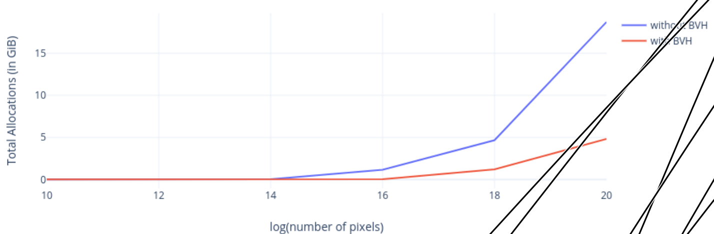


Fig. 2: Memory Allocation comparison between rendering with and without BVH

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 [4], due to the common AD backend, for use in more complex differentiable pipelines.

4. 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.

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. ~~It might be worthwhile to note that~~ we don't make ~~any sort of~~ performance compromise in the forward pass (rendering) to allow gradient computation.

4.1 Accelerating the Rendering Process

To accelerate the rendering process we ~~have acceleration structures. Currently only one such~~ acceleration structure, Bounding Volume Hierarchy [5], ~~is supported~~. We follow the exact same

Page: 2

Author: snoeyink Subject: Sticky Note Date: 7/9/2019 2:39:42 PM
These plots take far too much space for the small amount of data they contain.

Why is the number of pixels the measure? I'd expect it to be number of mesh elements.

Author: snoeyink Subject: Cross-Out Date: 7/9/2019 4:33:36 PM

Author: snoeyink Subject: Cross-Out Date: 7/9/2019 4:34:39 PM

Author: snoeyink Subject: Cross-Out Date: 7/9/2019 4:37:08 PM
This note does not seem to fit here, though, since this is about the user. Move to the end of the previous paragraph.

Author: snoeyink Subject: Pencil Date: 7/9/2019 4:36:07 PM

Author: snoeyink Subject: Inserted Text Date: 7/9/2019 4:35:43 PM
s

Author: snoeyink Subject: Inserted Text Date: 7/9/2019 4:37:24 PM
currently support one

Author: snoeyink Subject: Pencil Date: 7/9/2019 4:37:44 PM

Author: snoeyink Subject: Cross-Out Date: 7/9/2019 4:37:36 PM

Fig. 3: Utah Teapot Render from three different views. The camera definition shown in Listing [1] can be easily modified to generate all these views.

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"))*.

We show the benefits of using Acceleration Structures in Figures [1] and [2]. As can be seen from the plots we not only get a good performance gain (Figure [1]) in terms of speed, we also end up allocating much less memory (Figure [2] on using Bounding Volume Hierarchy.

4.2 Rendering the Utah Teapot

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

5. Inverse Rendering

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]~~.

6. Experiments

In this section we showcase our differentiable renderer. Using the following two experiments we validate the functionality of the gradients of our renderer. In both the experiments we make use of the Adam optimizer as described in [6]. 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.

Page: 3

Author: snoeyink Subject: Highlight Date: 7/9/2019 3:12:05 PM
Lose the brackets for Fig, Listing, Alg references, because it leads to ambiguity between them and references.

It is confusing to have Listing 1 and Algorithm 1; make them Algorithm 1 and Algorithm 2.

Author: snoeyink Subject: Highlight Date: 7/9/2019 4:38:50 PM
This is an experiment that you have not described with enough detail to be replicated. What is the scene? Why is pixels the measure of complexity? (And why is so much memory being allocated?)

Also, lose the [] around figure numbers.

Author: snoeyink Subject: Cross-Out Date: 7/9/2019 3:58:28 PM
stay in present tense

Author: snoeyink Subject: Cross-Out Date: 7/9/2019 3:58:16 PM

Author: snoeyink Subject: Cross-Out Date: 7/9/2019 3:58:49 PM

Author: snoeyink Subject: Cross-Out Date: 7/9/2019 3:59:56 PM

Author: snoeyink Subject: Cross-Out Date: 7/9/2019 3:58:56 PM

Author: snoeyink Subject: Inserted Text Date: 7/9/2019 3:59:12 PM
readers

Author: snoeyink Subject: Highlight Date: 7/9/2019 2:54:37 PM

Author: snoeyink Subject: Inserted Text Date: 7/9/2019 4:00:48 PM
then

Author: snoeyink Subject: Highlight Date: 7/9/2019 3:51:44 PM
Writing tip: Find and strengthen parallels. If you use the exact same sentence structure and word choice when you compare or contrast, then the essential differences will pop out:
The rendering problem is to project 3D scene parameters to form an image on a 2D plane; the inverse rendering problem is the opposite: mapping from the 2D image back to the parameters of the 3D scene.

Using the same words for the same concepts conserves the reader's energy for sorting out your ideas.

Author: snoeyink Subject: Inserted Text Date: 7/9/2019 3:13:24 PM
Listing (be consistent.) Actually, make this Algorithm.

Author: snoeyink Subject: Pencil Date: 7/9/2019 3:53:12 PM

Author: snoeyink Subject: Cross-Out Date: 7/9/2019 3:51:44 PM

Author: snoeyink Subject: Cross-Out Date: 7/9/2019 3:51:59 PM

Author: snoeyink Subject: Inserted Text Date: 7/9/2019 3:52:55 PM
problems (there is not one problem, but many, depending on the parameters you choose to solve for.)

Author: snoeyink Subject: Inserted Text Date: 7/9/2019 2:54:37 PM
with respect to

Author: snoeyink Subject: Inserted Text Date: 7/9/2019 3:54:41 PM
Algorithm 1 (will be Algorithm 2 if you take my previous advice.

Author: snoeyink Subject: Highlight Date: 7/9/2019 3:58:14 PM
It is essential that enough detail is given in experiments that they can be replicated.

Author: snoeyink Subject: Highlight Date: 7/9/2019 3:56:34 PM

Comments from page 3 continued on next page

Fig. 3: Utah Teapot Render from three different views. The camera definition shown in Listing [1] can be easily modified to generate all these views.

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"))*.

We show the benefits of using Acceleration Structures in Figures [1] and [2]. As can be seen from the plots we not only get a good performance gain (Figure [1]) in terms of speed, we also end up allocating much less memory (Figure [2] on using Bounding Volume Hierarchy.

4.2 Rendering the Utah Teapot

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

5. Inverse Rendering

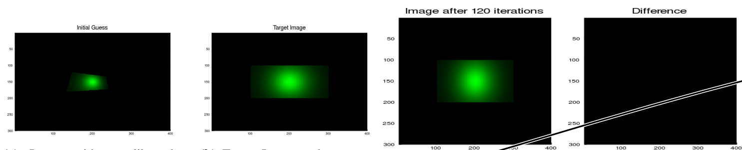
Just like rendering can be thought of as the projection of 3D objects into a 2D plane, the inverse rendering problem can be described as just the opposite. It is the mapping of the 2D image back to the parameters of the scene.

~~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 parameters 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]~~.

6. Experiments

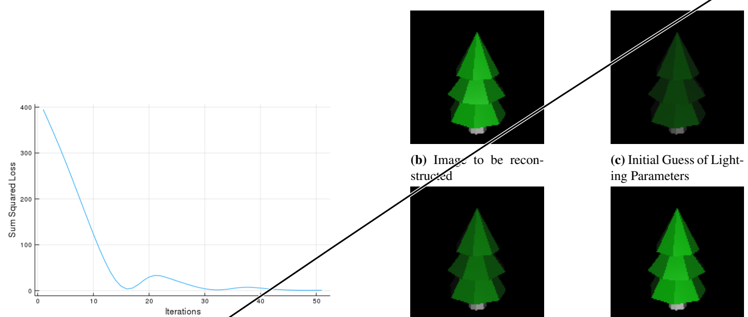
In this section we showcase our differentiable renderer. Using the following two experiments we validate the functionality of the gradients of our renderer. In both the experiments we make use of the Adam optimizer as described in [6]. We interface the raytracer with Flux to use these optimizers. As an alternative, we have tested the functioning of our package with the optimizers present in Optim.

demonstrate that we can use AD gradients to recover camera or lighting parameters for a scene.



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

Fig. 4: Calibration of Camera Parameters to reconstruct Image [4b] from Image [4a]



(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]

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

6.1 Calibration of Camera Parameters

In this experiment we start with the image of a rectangle under some configuration of the Camera model. Our aim is to reconstruct the image of this rectangle by modifying the focus and the location of the camera. We assume that all the other parameters of the model, like the light configuration, position of objects, etc. are known apriori.

We follow the standard algorithm as defined in ~~[1]~~ for optimizing the parameters. We make an initial guess of the parameters and initialize the camera as:

```
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...
    )
```


Page: 4

Author: snoeyink Subject: Highlight Date: 7/9/2019 3:45:55 PM
It is unfortunate that the aspect ratio changes in the image from b to c. Can you fix that?

Author: snoeyink Subject: Inserted Text Date: 7/9/2019 3:56:54 PM
2

Author: snoeyink Subject: Inserted Text Date: 7/9/2019 3:56:48 PM
Algorithm 2

~~We use the mean squared difference between the rendered image and the target image as the loss and try to minimize that.~~ The Adam optimizer is initialized with a learning rate of 0.05 and the model is declared to have as converged if the loss falls below 10. Figure [4] shows the optimization steps.

6.2 Optimizing the Light Source

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 [5].

7. Conclusion

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] Michael Innes. Don't unroll adjoint: Differentiating ~~ssa~~ form programs. *CoRR*, abs/1810.07951, 2018.

[4] 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.

[5] 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.

[6] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[7] 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.

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

Page: 5

Author: snoeyink Subject: Inserted Text Date: 7/9/2019 3:43:49 PM
As our loss function, we use the mean squared difference between rendered and target images, each with 512x512 pixels having fractional RGB values.
We minimize loss with the Adam optimizer, with learning rate 0.05, and declare the model to have converged if loss falls below 10.

[There is no need to switch to passive voice.]

Author: snoeyink Subject: Highlight Date: 7/9/2019 3:32:46 PM
To be able to replicate, or to understand what 10 means, I need to know more about your loss function. How many pixels? What color space are you using? Are colors integers or fractions?

Author: snoeyink Subject: Highlight Date: 7/9/2019 3:21:44 PM
To replicate this experiment one needs the details of how many lights and lighting parameters were used.

Author: snoeyink Subject: Highlight Date: 7/9/2019 3:09:42 PM

Author: snoeyink Subject: Inserted Text Date: 7/9/2019 3:05:51 PM
Julia

Author: snoeyink Subject: Highlight Date: 7/9/2019 3:09:31 PM
"the problem" is not well defined -- if the problem is "inverse graphics" then there are many non-differentiable decisions in creating a model...

Author: snoeyink Subject: Highlight Date: 7/9/2019 3:08:11 PM
Is the pipeline the system? The system is not defined, and the previous use was about Julia making fully differentiable systems.

Author: snoeyink Subject: Inserted Text Date: 7/9/2019 3:19:30 PM
{SSA}

Author: snoeyink Subject: Highlight Date: 7/9/2019 3:18:50 PM
complete this entry.

Author: snoeyink Subject: Inserted Text Date: 7/9/2019 3:17:54 PM
{M}onte {C}arlo will keep the capitalization in bibtex.