Laboratório VISGRAF
Instituto de Matemática Pura e Aplicada

Physically Based Differentiable Rendering

Thales Magalhaes,
Luiz Henrique de Figueiredo and Luiz Velho (supervisors)

Technical Report TR-21-05 Relatório Técnico

April - 2021 - Abril

Physically Based Differentiable Rendering

Thales Magalhães

1 Introduction

This technical report describes a simple differentiable path tracer based on the radiative backpropagation algorithm as proposed by Nimier-David et al. [1] and the adjoint method for light transport by Jos Stam [2]. Written for educational purposes, this aims to be an understandable, yet reasonably efficient implementation of the algorithm.

2 Background

Rendering algorithms, whether based on rasterization or path tracing, are tipically not a differentiable process. This can be a problem for applications which rely on taking gradients, such as when solving optimization problems. In particular, a differentiable renderer is requried when computing the loss term involves a rendering step. This is often the case in inverse rendering problems, where the goal is to estimate scene parameters based on ground-truth images. These types of renderers are also used during the training phase in some machine learning applications. Differentiable rendering algorithms solve this problem by formulating the rendering process in such a way that gradients may be obtained from pixel colors with respect to the scene parameters.

2.1 Light Transport

Physically based rendering algorithms model the propagation of light in a scene via the the so-called "light transport" equations. These equations (defined below) describe how light radiates in straight lines between surface points and then scatters in an energy-conserving manner. The rendering process, in effect, consists of finding approximate solutions to these equations in order to estimate the ammount of light arriving at a particular sensor (such as the pixels in a camera). This is done by using Monte Carlo integration via the path tracing algorithm, in which light paths are randomly sampled throughout the scene and their radiance determined according these same equations.

$$L_i(\mathbf{p}, \boldsymbol{\omega}) = L_o(t(\mathbf{p}, \boldsymbol{\omega}), -\boldsymbol{\omega})$$
$$L_o(\mathbf{p}, \boldsymbol{\omega}) = L_e(\mathbf{p}, \boldsymbol{\omega}) + \int_{S^2} L_i(\mathbf{p}, \boldsymbol{\omega}')\ f(\mathbf{p}, \boldsymbol{\omega}, \boldsymbol{\omega}')\ \mathrm{d}\boldsymbol{\omega}'$$

Here $(\mathbf{p}, \boldsymbol{\omega})$ defines a ray with origin $\mathbf{p} \in \mathbb{R}^3$ and direction $\boldsymbol{\omega} \in S^2$. L_i, L_o, and L_e represent the incoming, outgoing, and emitted radiances respectively. f is the bidirectional reflectance function (BRDF) and t computes the first intersection of a ray.

2.2 Radiative Backpropagation

By taking the derivative of these equations with respect to the scene parameters (details in the aforementioned papers), we arrive at a formulation for the gradient. Here we discover that, perhaps surprisingly, these new "adjoint transport" equations are in fact very similar to the original light transport equations which we started with. Therefore, it stands to reason that we may solve these equations by using a process analogous to the path tracing algorithm. This key insight is what enables us to model the propagation of gradients throughout the scene in similar fashion to how we would with light in a process called "radiative backpropagation."

$$\partial_{\mathbf{x}} L_i(\mathbf{p}, \boldsymbol{\omega}) = \partial_{\mathbf{x}} L_o(t(\mathbf{p}, \boldsymbol{\omega}), -\boldsymbol{\omega})$$
$$\partial_{\mathbf{x}} L_o(\mathbf{p}, \boldsymbol{\omega}) = \partial_{\mathbf{x}} L_e(\mathbf{p}, \boldsymbol{\omega}) + \int_{S^2} \left[\ \partial_{\mathbf{x}} L_i(\mathbf{p}, \boldsymbol{\omega}')\ f(\mathbf{p}, \boldsymbol{\omega}, \boldsymbol{\omega}') + L_i(\mathbf{p}, \boldsymbol{\omega}')\ \partial_{\mathbf{x}} f(\mathbf{p}, \boldsymbol{\omega}, \boldsymbol{\omega}')\ \right] \mathrm{d}\boldsymbol{\omega}'$$

The above variables hold the same meaning as before while $\mathbf{x}$ is a vector representing the scene parameters.

3 Implementation

Efficient computation of the gradient is a two step process. In the forward step, we determine the radiance values along a light path as we would in a traditional path tracer. Then, in the backward step, we propagate the gradient in the opposite direction according to the differential transport equations. Perhaps the most straightforward way to implement this radiative backpropagation algorithm would be to explicitly perform both of these steps. We will be taking a more general approach by only performing the forward step explicitly, while leaving the backward step to be performed as an automatic differentiation task. Because of this, our differential path tracer implementation won't look much different from a traditional path tracer, which is a big plus. On the other hand, this approach relies on having an existing framework with some basic automatic differentiation functionality.

3.1 Automatic Differentiation

For our purposes, we only need to support a small set of vector operations. To this end we'll be employing a strategy inspired by PyTorch [3], in which we record every relevant operation as part of a computation graph. We may then perform automatic differentiation by backpropagating the gradients throught this graph. In practice, every instance of our vector class will contain a `backward` member function which may be called to trigger this backpropagation process.

As an example, the following code declares a variable `x`. This type of vector stores both a value and a gradient and will therefore always create a new node in the computation graph when used as part of an expression (template parameters have been ommited for brevity):

```
Vector x({1, 2, 3}, true);
```

This, on the other hand, declares two constant vectors `c` and `d`. Since they do not store gradient values, these vectors won't crate new nodes in the computation graph unless they appear as part of an expression along with other, non-constant vectors:

```
Vector c = {4, 5, 6};
Vector d = {7, 8, 9};
```

Since it contains a variable, the execution of the following expression will be implicitly recorded as part of the computation graph by creating a new node representing the addition operation between `x` and `c`:

```
auto y = x + c;
```

On the other hand, since the following expression only depends on the values of two constants, it will not be recorded as part of the graph but instead will generate a new vector constant:

```
auto k = c + d;
```

Below we explicitly define a backpropagation procedure by passing in a functor (in this case a closure defined via a lambda expression) as and additional constructor parameter. While in this example the procedure simply outputs a string to `cout`, this alternative method of constructing vectors may be useful in cases when explicitly defining the backpropagation procedure would be more efficient or accurate than doing so implicitly:

```
Vector z(y, [](auto _) { cout << "Hello, world!" << endl; });
```

Finally, invoking `backward` on the resulting vector triggers the backpropagation process and accumulates the computed gradients for every relevant variable. After that, the gradients can be acessed by invoking the `grad` member function on the respective variables:

```
z.backward();
x.grad() // Returns the accumulated gradient value
```

3.2 Path Tracing

By using this automatic differentiation framework as a backend to our differentiable path tracer we may implement it almost exactly as we would a traditional path tracer. This is because the path tracing algorithm is equivalent to the forward step in the differentiation process, and by implementing it we get the backward step "for free." Computing the gradients becomes just a matter of running the algorithm forward to obtain the radiance values then invoking `backward` on the resulting vector, as seen below (details ommited for brevity):

```
Vector param = /* define parameter */
Scene scene = /* build scene using parameters */
Pathtracer tracer;

for (auto pixel : image) {
  auto [orig, dir] = pixel_to_ray(pixel);
  auto radiance = tracer.trace(scene, orig, dir);
  auto loss = loss_func(radiance);
  loss.backward();
}

// Access the accumulated gradient per parameter
param.grad();
```

3.3 Avoiding Bias

While the above approach approximates gradients well enough for most applications, technically it does not yield truly unbiased results. Because our estimates rely heavily on Monte Carlo integration, we should be careful to take samples independently at each step of the differentiation process. Unfortunately, since our automatic differentiation strategy relies on recording intermediate operations, it just so happens that our backward step reuses the exact same samples as the forward step. To solve this problem, we introduce an explicit integration operator that ensures independent sampling for both steps by defining a custom backpropagation strategy:

```
template <typename Forward, typename Sampler>
struct IntegrateBackward {
    void operator()(Vector grad) const;

    Forward forward;
    Sampler sampler;
    size_t n_samples;
};
```

```
template <typename Forward, typename Sampler>
Vector integrate(Forward forward,
                 Sampler sampler,
                 size_t n_samples)
{
    Vector result(0);
    for (size_t i = 0; i < n_samples; ++i) {
        auto [sample, pdf] = sampler();
        r += forward(sample) / pdf;
    }
    return Vector(r,
        IntegrateBackward{forward, sampler, n_samples});
}

template <typename Forward, typename Sampler>
void IntegrateBackward<Forward, Sampler>
    ::operator()(Vector grad) const
{
    for (size_t i = 0; i < n_samples; ++i) {
        auto [sample, pdf] = sampler();
        forward(sample).backward(grad / pdf);
    }
}
```

Although it takes care of the bias problem, this solution comes at the cost of an increased computational complexity when computing gradients. For this reason, it's usage is not enforced since having truly unbiased estimates is often not a requirement for optimization problems.

4 Results

The resulting gradients obtained from this implementation have been experimentally validated against those generated by using forward mode automatic differentiation. To do this, we simply ran the path tracer using dual numbers [4] as the underlying data type instead of using floating-point numbers. This simple form of automatic differentiation is analogous to using finite differences, except it is less prone to precision-related errors.

5 Limitations

It is important to note that this implementation, as well as the methods which it is based on, is not concerned with handling the visibility-related discontinuities which may arise during the rendering process. As such, it will not compute the correct gradients w.r.t. parameters related to scene geometry such as object positions, rotations, scale, etc. It will, however, compute accurate gradients w.r.t.

parameters related to shading such as diffuse color, specularity, and emission strength[1]. With that said, the techniques required for handling such discontinuities are largely orthogonal to the methods implemented here. Therefore, extending this differentiable renderer to handle such discontinuities should not pose a major challenge.

References

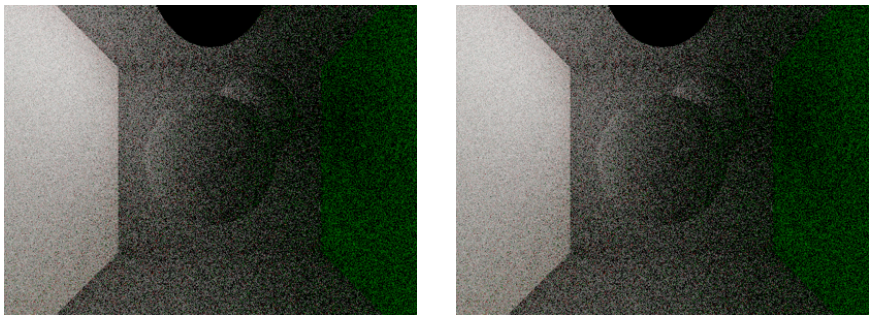
[1] Merlin Nimier-David et al. "Radiative Backpropagation: An Adjoint Method for Lightning-Fast Differentiable Rendering". In: *Transactions on Graphics (Proceedings of SIGGRAPH)* 39.4 (July 2020). DOI: 10.1145/3386569.3392406.

[2] Jos Stam. *Computing Light Transport Gradients using the Adjoint Method*. 2020. arXiv: 2006.15059 [cs.GR].

[3] Adam Paszke et al. "Automatic differentiation in PyTorch". In: 2017.

[4] Wikipedia contributors. *Dual number — Wikipedia, The Free Encyclopedia*. [Online; accessed 10-April-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Dual_number&oldid=1015073301.

[1] Assuming no discontinuities are present in the BRDFs themselves.



(a) Our method (b) Ground-truth

Figure 1: Gradients of the pixel colors in a sample scene w.r.t. the parameter controlling the color of the left wall. Notice how the gradient values seem to be emitted from the wall and scatter around the scene much like light would. Results obtained using the radiative backpropagation algorithm (a) and forward mode automatic differentiation (b).