OpenDR: An Approximate Differentiable Renderer*

Matthew M. Loper and Michael J. Black

Max Planck Institute for Intelligent Systems, Tübingen, Germany
{mloper,black}@tue.mpg.de

Abstract. Inverse graphics attempts to take sensor data and infer 3D geometry, illumination, materials, and motions such that a graphics renderer could realistically reproduce the observed scene. Renderers, however, are designed to solve the forward process of image synthesis. To go in the other direction, we propose an approximate *differentiable renderer (DR)* that explicitly models the relationship between changes in model parameters and image observations. We describe a publicly available *OpenDR* framework that makes it easy to express a forward graphics model and then automatically obtain derivatives with respect to the model parameters and to optimize over them. Built on a new auto-differentiation package and OpenGL, OpenDR provides a local optimization method that can be incorporated into probabilistic programming frameworks. We demonstrate the power and simplicity of programming with OpenDR by using it to solve the problem of estimating human body shape from Kinect depth and RGB data.

Keywords: Inverse graphics, Rendering, Optimization, Automatic Differentiation, Software, Programming.

1 Introduction

Computer vision as *analysis by synthesis* has a long tradition [9,24] and remains central to a wide class of generative methods. In this top-down approach, vision is formulated as the search for parameters of a model that is *rendered* to produce an image (or features of an image), which is then compared with image pixels (or features). The model can take many forms of varying realism but, when the model and rendering process are designed to produce realistic images, this process is often called *inverse graphics* [3,33]. In a sense, the approach tries to reverse-engineer the physical process that produced an image of the world.

We define an observation function $f(\Theta)$ as the forward rendering process that depends on the parameters Θ. The simplest optimization would solve for the parameters minimizing the difference between the rendered and observed image intensities, $E(\Theta) = \|f(\Theta) - I\|^2$. Of course, we will specify much more sophisticated functions, including robust penalties and priors, but the basic idea

* Electronic supplementary material -Supplementary material is available in the online version of this chapter at http://dx.doi.org/10.1007/978-3-319-10584-0_11. Videos can also be accessed at http://www.springerimages.com/videos/978-3-319-10583-3

D. Fleet et al. (Eds.): ECCV 2014, Part VII, LNCS 8695, pp. 154–169, 2014.

remains – minimize the difference between the synthesized and observed data. While much has been written about this process and many methods fall under this rubric, few methods literally adopt the inverse graphics approach. High dimensionality makes optimizing an objective like the one above a challenge; renderers have a large output space, and realistic renderers require a large input parameter space. Fundamentally, the forward rendering function is complex, and optimization methods that include it are often purpose-built with great effort. Put succinctly, *graphics renderers are not usually built to be inverted.*

Here we fully embrace the view of vision as inverse graphics and propose a framework to make it more practical. Realistic graphics engines are available for rendering the forward process and many discriminative approaches exist to recover scene properties directly from images. Neither explicitly models how the observables (pixels or features) smoothly change with model parameters. These derivatives are essential for optimization of high-dimensional problems and constructing these derivatives by hand for each application is onerous. Here we describe a general framework based on differentiating the render. We define a *differentiable renderer (DR)* as a process that (1) supplies pixels as a function of model parameters to simulate a physical imaging system and (2) supplies derivatives of the pixel values with respect to those parameters. To be practical, the DR also has to be fast; this means it must have hardware support. Consequently we work directly with OpenGL. Because we make it publicly available, we call our framework *OpenDR* (`http://open-dr.org`).

Since many methods formulate generative models and differentiate them, why has there been no general DR framework until now? Maybe it is because rendering seems like it is not differentiable. At some level this is true, but the question is whether it matters in practice. All renderers are approximate and our DR is no exception. We describe our approximations in Sections 3 and 4 and argue that, in practice, "approximately differentiable" is actually very useful.

Our goal is not rendering, but inverse rendering: we wish to specify and minimize an objective, in which the renderer is only one part. To that end, our DR is built upon a new autodifferentiation framework, called Chumpy, in Python that makes programming compact and relatively easy. Our public autodiff framework makes it easy to extend the basic features of OpenDR to address specific problems. For example, instead of specifying input geometry as vertices, one might parameterize the vertices in a shape space; or in the output, one might want a Laplacian pyramid of pixels, or edges, or moments, instead of the raw pixel values. While autodifferentiation does not remove the need to write these functions, it does remove the need to differentiate them by hand.

Using this we define the OpenDR framework that supports a wide range of real problems in computer vision. The OpenDR framework provides a compact and efficient way of expressing computer vision problems without having to worry about how to differentiate them. This is the first publicly-available framework for differentiating the image generation process.

To evaluate the OpenDR, and to illustrate how to use it, we present two examples. The first is a simple "hello world" example, which serves to illustrate

the basic ideas of the OpenDR. The second, more complex, example involves fitting an articulated and deformable model of 3D human body shape to image and range data from a Kinect. Here we optimize 3D body shape, pose, lighting, albedo, and camera parameters. This is a complex and rich generative model and optimizing it would generally be challenging; with OpenDR, it is straightforward to express and optimize.

While differentiating the rendering process does not solve the computer vision problem, it does address the important problem of local refinement of model parameters. We see this as piece of the solution that is synergistic with stochastic approaches for probabilistic programming [22]. We have no claim of novelty around vision as inverse graphics. Our novelty is in making it practical and easy to solve a fairly wide class of such problems. We believe the OpenDR is the first generally available solution for differentiable rendering and it will enable people to push the analysis-by-synthesis approach further.

2 Related Work

The view of vision as *inverse graphics* is nearly as old as the field itself [3]. It appears in the work of Grenander on *analysis by synthesis* [9], in physics-based approaches [13], in regularization theory [5,32], and even as a model for human perception [18,24,27]. This approach plays an important role in Bayesian models and today the two notions are tightly coupled [19]. In the standard Bayesian formulation, the likelihood function specifies the forward rendering process, while the prior constrains (or regularizes) the space of models or parameters [19]. Typically the likelihood does not involve an actual render in the standard graphics sense. In graphics, "inverse rendering" typically refers to recovering the illumination, reflectance, and material properties from an image (e.g. the estimation of BRDFs); see [26] for a review. When we talk about inverting the rendering process we mean something more general, involving the recovery of object shape, camera parameters, motion, and illumination.

The theory of inverse graphics is well established, but what is missing is the direct connection between rendering and optimization from images. Graphics is about synthesis. Inference is about going from observations to models (or parameters). *Differentiable rendering* connects these in a concrete way by explicitly relating changes in the observed image with changes in the model parameters.

Stochastic Search and Probabilistic Programming. Our work is similar philosophy to Mansinghka et al. [22]. They show how to write simple probabilistic graphics programs that describe the generative model of a scene and how this relates to image observations. They then use automatic and approximate stochastic inference methods to infer the parameters of the scene model from observations. While we share the goal of automatically inverting graphics models of scenes, our work is different and complimentary. They address the stochastic search problem while we address the deterministic refinement problem. While stochastic sampling is a good way to get close to a solution, it is typically not a good way to refine a solution. A full solution is likely to incorporate both of these elements

of search and refinement, where the refinement stage can use richer models, deterministic optimization, achieve high accuracy, and be more efficient.

Our work goes beyond [22] in other ways. They exploit a very general but computationally inefficient Metropolis-Hastings sampler for inference that will not scale well to more complex problems. While their work starts from the premise of doing inference with a generic graphics rendering engine, they do not cope with 3D shape, illumination, 3D occlusion, reflectance, and camera calibration; that is, they do no render graphics scenes as we typically think of them. None of this is to diminish the importance of that work, which lays out a framework for probabilistic scene inference. This is part of a more general trend in probabilistic programming where one defines the generative graphical model and lets a generic solver do the inference [8,23,37]. Our goal is similar but for deterministic inference. Like them we offer a simple programming framework in which to express complex models.

Recently Jampani et al. [15] define a generic sampler for solving inverse graphics problems. They use discriminative methods (bottom up) to inform the sampler and improve efficiency. Their motivation is similar to ours in that they want to enable inverse graphics solutions with simple generic optimization methods. Their goal differs however in that they seek a full posterior distribution over model parameters, while we seek a local optimum. In general, their method is complimentary to ours and the methods could be combined.

Differentiating Graphics Models. Of course we are not the first to formulate a generative graphics model for a vision problem, differentiate it, and solve for the model parameters. This is a tried-and-true approach in computer vision. In previous work, however, this is done as a "one off" solution and differentiating the model is typically labor intensive. For a given model of the scene and particular image features, one defines an observation error function and differentiates this with respect to the model parameters. Solutions obtained for one model are not necessarily easily applied to another model. Some prominent examples follow.

Face Modeling: Blanz and Vetter [6] define a detailed generative model of human faces and do analysis by synthesis to invert the model. Their model includes 3D face shape, model texture, camera pose, ambient lighting, and directional lighting. Given model parameters they synthesize a realistic face image and compare it with image pixels using sum-of-squared differences. They explicitly compute derivatives of their objective function and use a stochastic gradient descent method for computational reasons and to help avoid local optima.

3D Shape Estimation: Jalobeanu et al. [14] estimate underlying parameters (lighting, albedo, and geometry) of a 3D planetary surface with the use of a differentiated rendering process. They point out the importance of accurate rendering of the image and the derivatives and work in *object space* to determine visibilities for each pixel using computational geometry. Like us, they define a differentiable rendering process but with a focus on Bayesian inference.

Smelyansky et al. [29] define a "fractional derivative renderer" and use it to compute camera parameters and surface shape together in a stereo reconstruction problem. Like [14], they use geometric modeling to account for the

fractional contributions of different surfaces to a pixel. While accurate, such a purely geometric approach is potentially slow.

Bastian [2] also argues that working in object space avoids problems of working with pixels and, in particular, that occlusions are a problem for differentiable rendering. He suggests super-sampling the image as one solution to approximate a differentiable render. Instead he uses MCMC sampling and suggests that sampling could be used in conjunction with a differentiable renderer to avoid problems due to occlusion. See also [34], which addresses similar issues in image modeling with a continuous image representation.

It is important to remember that any render only produces an approximation of the scene. Consequently any differentiable render will only produce approximations of the derivatives. This is true whether one works in object space or pixel space. The question is how good is the approximation and how practical is it to obtain? We argue below that pixel space provides the better tradeoff.

Human Pose and Shape: Sminchisecu [31] formulates the articulated 3D human tracking problem from monocular video. He defines a generative model of edges, silhouettes and optical flow and derives approximations of these that are differentiable. In [30] Sminchisescu and Telea define a generic programming framework in which ones specifies models and relates these to image observations. This framework does not automatically differentiate the rendering process.

de La Gorce et al. [20] recover pose, shape, texture, and lighting position in a hand tracking application. They formulate the problem as a forward graphics synthesis problem and then differentiate it, paying special attention to obtaining derivatives at object boundaries; we adopt a similar approach. Weiss et al. [36] estimate both human pose and shape using range data from Kinect and an edge term corresponding to the boundary of the human body. They formulate a differentiable silhouette edge term and mention that it is sometimes not differentiable, but that this occurs at only finitely many points, which can be ignored.

The above methods all render a model of the world and differentiate some image error with respect to the model parameters. Despite the fact that they all can be seen as inverse rendering, in each case the authors formulate an objective and then devise a way to approximately differentiate it. Our key insight is that, instead of differentiating each problem, we *differentiate the render.* Then any problem that can be posed as rendering is, by construction, (approximately) differentiable. To formulate a new problem, one writes down the forward process (as expressed by the rendering system), the derivatives are given automatically, and optimization is performed by one of several local optimization methods. This approach of differentiating the rendering process provides a general solution to many problems in computer vision.

3 Defining Our Forward Process

Let $f(\Theta)$ be the rendering function, where Θ is a collection of all parameters used to create the image. Here we factor Θ into vertex locations V, camera parameters C, and per-vertex brightness A: therefore $\Theta = \{V, C, A\}$. Inverse graphics is inherently approximate, and it is important to establish our approximations in

both the forward process and its differentiation. Our forward model makes the following approximations:

Appearance (A): Per-pixel surface appearance is modeled as product of mipmapped texture and per-vertex brightness, such that brightness combines the effects of reflectance and lighting. Spherical harmonics and point light sources are available as part of OpenDR; other direct lighting models are easy to construct. Global illumination, which includes interreflection and all the complex effects of lighting, is not explicitly supported.

Geometry (V): We assume a 3D scene to be approximated by triangles, parameterized by vertices V, with the option of a background image (or depth image for the depth renderer) to be placed behind the geometry. There is no explicit limit on the number of objects, and the DR does not even "know" whether it is rendering one or more objects; its currency is triangles, not objects.

Camera (C): We approximate continuous pixel intensities by their sampled central value. We use the pinhole-plus-distortion camera projection model from OpenCV. Its primary difference compared with other projections is in the details of the image distortion model [7], which are in turn derived from [11].

Our approximations are close to those made by modern graphics pipelines. One important exception is appearance: modern graphics pipelines support per-pixel assignment on surfaces according to user-defined functions, whereas here we support *per-vertex* user-defined functions (with colors interpolated between vertices). While we also support texture mapping, we do not yet support differentiation with respect to intensity values on the texture map. Unlike de La Gorce [20], we do not support derivatives with respect to texture; whereas they use bilinear interpolation, we would require trilinear interpolation because of our use of mipmapping. This is future work.

We emphasize that, if the OpenDR proves useful, users will hopefully expand it, relaxing many of these assumptions. Here we describe the initial release.

4 Differentiating Our Forward Process

To describe the partial derivatives of the forward process, we introduce U as an intermediate variable indicating 2D projected vertex coordinate positions. Differentiation follows the chain rule as illustrated in Fig. 1. Our derivatives may be grouped into the effects of appearance ($\frac{\partial f}{\partial A}$), and changes in projected coordinates ($\frac{\partial U}{\partial C}$ and $\frac{\partial U}{\partial V}$), and the effects of image-space deformation ($\frac{\partial f}{\partial U}$).

4.1 Differentiating Appearance

Pixels projected by geometry are colored by the product of texture T and appearance A; therefore $\frac{\partial f}{\partial A}$ can be quickly found by rendering the texture-mapped geometry with per-vertex colors set to 1.0, and weighting the contribution of surrounding vertices by rendered barycentric coordinates. Partials $\frac{\partial A}{\partial V}$ may be zero (if only ambient color is required), may be assigned to built-in spherical harmonics or point light sources, or may be defined directly by the user.

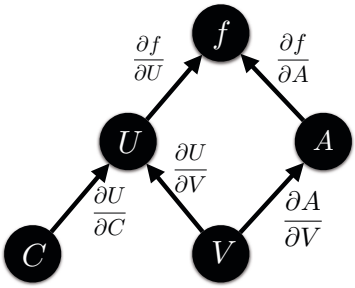

Fig. 1. Partial derivative structure of the renderer

4.2 Differentiating Projection

Image values relate to 3D coordinates and camera calibration parameters via 2D coordinates; that is, where U indicates 2D coordinates of vertices,

$$\frac{\partial f}{\partial V} = \frac{\partial f}{\partial U}\frac{\partial U}{\partial V}, \qquad \frac{\partial f}{\partial C} = \frac{\partial f}{\partial U}\frac{\partial U}{\partial C}.$$

Partials $\frac{\partial U}{\partial V}$ and $\frac{\partial U}{\partial C}$ are straightforward, as projection is well-defined. Conveniently, OpenCV provides $\frac{\partial U}{\partial C}$ and $\frac{\partial U}{\partial V}$ directly.

4.3 Differentiating Intensity with Respect to 2D Image Coordinates

In order to estimate $\frac{\partial f}{\partial U}$, we first segment our pixels into occlusion boundary pixels and interior pixels, as inspired by [20]. The change induced by boundary pixels is primarily due to the replacement of one surface with another, whereas the change induced by interior pixels relates to the image-space projected translation of the surface patch. The assignment of boundary pixels is obtained with a rendering pass by identifying pixels on edges which (a) pass a depth test (performed by the renderer) and (b) join triangles with opposing normals: one triangle facing towards the camera, one facing away. We consider three classifications for a pixel: interior, interior/boundary, and many-boundary.

Interior: a pixel contains no occlusion boundaries. Because appearance is a product of interpolated texture and interpolated color, intensity changes are piecewise smooth with respect to geometry changes. For interior pixels, we use the image-space first-order Taylor expansion approach adopted by [17]. To understand this approach, consider a patch translating right in image space by a pixel: each pixel becomes replaced by its lefthand neighbor, which is similar to the application of a Sobel filter. Importantly, we do not allow this filtering to cross or include boundary pixels (a case not handled by [17] because occlusion was not modeled).

Specifically, on pixels not neighboring an occlusion boundary, we perform horizontal filtering with the kernel $\frac{1}{2}[-1, 0, 1]$. On pixels neighboring an occlusion

boundary on the left, we use $[0, -1, 1]$ for horizontal filtering; with pixels neighboring occlusion boundaries on the right, we use $[-1, 1, 0]$; and with occlusion boundaries on both sides we approximate derivatives as being zero. With vertical filtering, we use the same kernels transposed.

Interior/Boundary: a pixel is intersected by one occlusion boundary. For the interior/boundary case, we use image-space filtering with kernel $\frac{1}{2}[-1, 0, 1]$ and its transpose. This approximates one difference (that between the foreground boundary and the surface behind it) with another (that between the foreground boundary and a pixel neighboring the surface behind it). Instead of "peeking" behind an occluding boundary, we are using a neighboring pixel as a surrogate and assuming that the difference is not too great. In practical terms, the boundary gradient is almost always much larger than the gradient of the occluded background surface patch, and therefore dominates the direction taken during optimization.

Many-Boundary: more than one occlusion boundary is present in a pixel. While object space methods provide exact derivatives for such pixels at the expense of modeling all the geometry, we treat this as an interior/boundary case. This is justified because very few pixels are affected by this scenario and because the exact object-space computation would be prohibitively expensive.

To summarize, the most significant approximation of the differentiation process occurs boundary pixels where we approximate one difference (nearby pixel minus occluded pixel) with another (nearby pixel minus almost-occluded pixel). We find this works in practice, but it is important to recognize that better approximations are possible [20].

As an implementation detail, our approach requires one render pass when a raw rendered image is requested, and an additional three passes (for boundary identification, triangle identification, and barycentric coordinates) when derivatives are requested. Each pass requires read back from the GPU.

4.4 Software Foundation

Flexibility is critical to the generality of a differentiable renderer; custom functions should be easy to design without requiring differentiation by hand. To that end, we use automatic differentiation [10] to compute derivatives given only a specification of the forward process, without resorting to finite differencing methods. As part of the OpenDR release we include a new automatic differentiation framework (Chumpy). This framework is essentially Numpy [25], which is a numerical package in Python, made differentiable. By sharing much of the API of Numpy, this allows the forward specification of problems with a popular API. This in turn allows the forward specification of models not part of the renderer, and allows upper layers of the renderer to be specified minimally. Although alternative auto-differentiation frameworks were considered [4,35,21], we wrap Numpy for its ease-of-use. Our overall system depends on Numpy [25], Scipy [16], and OpenCV [7].

5 Programming in OpenDR: Hello World

First we illustrate construction of a renderer with a texture-mapped 3D mesh of Earth. In Sec. 3, we introduced f as a function of $\{V, A, U\}$; in Fig. 2, V, A, U and f are constructed in turn. While we use spherical harmonics and a static set of vertices, anything expressible in Chumpy can be assigned to these variables, as long the dimensions make sense: given N vertices, then V and A must be $N \times 3$, and U must be $N \times 2$.

```
from opendr.simple import *
w, h = 320, 240

import numpy as np
m = load_mesh('nasa_earth.obj')

# Create V, A, U, f: geometry, brightness, camera, renderer
V = ch.array(m.v)
A = SphericalHarmonics(vn=VertNormals(v=V, f=m.f),
                       components=[3.,1.,0.,0.,0.,0.,0.,0.,0.],
                       light_color=ch.ones(3))
U = ProjectPoints(v=V, f=[300,300.], c=[w/2.,h/2.], k=ch.zeros(5),
                  t=ch.zeros(3), rt=ch.zeros(3))
f = TexturedRenderer(vc=A, camera=U, f=m.f, bgcolor=[0.,0.,0.],
                     texture_image=m.texture_image, vt=m.vt, ft=m.ft,
                     frustum={'width':w, 'height':h, 'near':1,'far':20})
```

Fig. 2. Constructing a renderer in OpenDR

Figure 3 shows the code for optimizing a model of Earth to match image evidence. We reparameterize V with translation and rotation, express the error to be minimized as a difference between Gaussian pyramids, and find a local minimum of the energy function with simultaneous optimization of translation, rotation, and light parameters. Note that a Gaussian pyramid can be written as a linear filtering operation and is therefore simply differentiable. The process is visualized in Fig. 4.

In this example, there is only one object; but as mentioned in Sec. 3, there is no obvious limit to the number of objects, because geometry is just a collection of triangles whose vertices are driven by a user's parameterization. Triangle face connectivity is required but may be disjoint.

Image pixels are only one quantity of interest. Any differentiable operation applied to an image can be applied to the render and hence we can minimize the difference between functions of images. Figure 5 illustrates how to minimize the difference between image edges and rendered edges. For more examples, the `opendr.demo()` function, in the software release, shows rendering of image moments, silhouettes, and boundaries, all with derivatives with respect to inputs.

```
# Parameterize the vertices
translation, rotation = ch.array([0,0,4]), ch.zeros(3)
f.v = translation + V.dot(Rodrigues(rotation))

# Create the energy
difference = f - load_image('earth_observed.jpg')
E = gaussian_pyramid(difference, n_levels=6, normalization='SSE')

# Minimize the energy
light_parms = A.components
ch.minimize(E, x0=[translation])
ch.minimize(E, x0=[translation, rotation, light_parms])
```

Fig. 3. Minimizing an objective function given image evidence. The derivatives from the renderer are used by the minimize method. Including a translation-only stage typically speeds convergence.

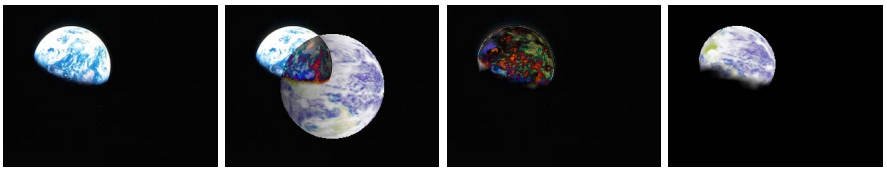

Fig. 4. Illustration of optimization in Figure 3. In order: observed image of earth, initial absolute difference between the rendered and observed image intensities, final difference, final result.

6 Experiments

Run-time depends on many user-specific decisions, including the number of pixels, triangles, underlying parameters and model structure. Figure 6 illustrates the effects of resolution on run-time in a simple scenario on a 3.0 GHz 8-core 2013 Mac Pro. We render a subdivided tetrahedron with 1024 triangles, lit by spherical harmonics. The mesh is parameterized by translation and rotation, and timings are according to those 6 parameters. The figure illustrates the overhead associated with differentiable rendering.

```
rn = TexturedRenderer(...)
edge_image = rn[:,1:,:] - rn[:,:-1,:]
ch.minimize(ch.sum((edge_image - my_edge_image)**2.),
             x0=[rn.v], method='bfgs')
```

Fig. 5. Optimizing a function of the rendered image to match a function of image evidence. Here the function is an edge filter.

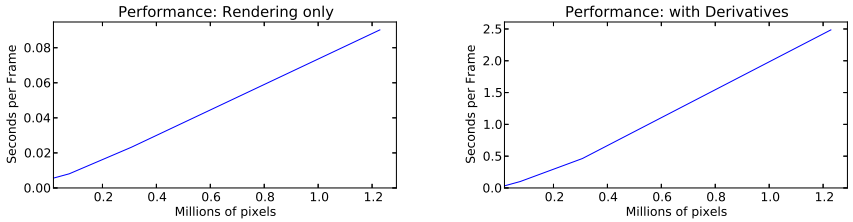

Fig. 6. Rendering performance versus resolution. For reference, 640x480 is 0.3 million pixels. Left: with rendering only. Right: with rendering and derivatives.

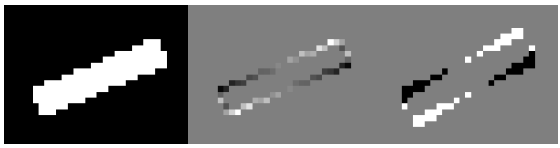

Fig. 7. Differentiable rendering versus finite differencing. Left: a rotating quadrilateral. Middle: OpenDR's predicted change in pixel values with respect to in-plane rotation. Right: finite differences recorded with a change to in-plane rotation.

Finite differences on original parameters are sometimes faster to compute than analytic differences. In the experiment shown in Fig. 6, at 640x480, it is 1.75 times faster to compute forward finite differencing on 6 parameters than to find analytic derivatives according to our approach. However, if derivatives with respect to all 514 vertices are required, then forward finite differencing becomes approximately 80 times slower than computing derivatives with our approach.

More importantly, the correct finite differencing epsilon is pixel-dependent. Figure 7 shows that the correct epsilon for finite-differencing can be spatially varying: the chosen epsilon is too small for some pixels and too large for others.

6.1 Body Shape from Kinect

We now address a body measurement estimation problem using the Kinect as an input device. In an analysis-by-synthesis approach, many parameters must be estimated to effectively explain the image and depth evidence. We effectively estimate thousands of parameters (per-vertex albedo being the biggest contributor) by minimizing the contribution of over a million residuals; this would be impractical with derivative-free methods.

Subjects were asked to form an A-pose or T-pose in two views separated by 45 degrees; then a capture was performed without the subject in view. This generates three depth and three color images, with most of the state, except pose, assumed constant across the two views.

Our variables and observables are as follows:

– **Latent Variables:** lighting parameters A_L, per-vertex albedo A_C, color camera translation T, and body parameters B: therefore $\Theta = \{A_L, A_C, T, B\}$.

– **Observables:** depth images $D_{1...n}$ and color images $I_{1...n}$, $n = 3$.

Appearance, A, is modeled here as a product of per-vertex albedo, A_C, and spherical harmonics parameterized by A_L: $A = A_C H(A_L, V)$, where $H(A_L, V)$ gives one brightness to each vertex according to the surface normal. Vertices are generated by a BlendSCAPE model [12], controlled by pose parameters $P_{1..n}$ (each of n views has a slightly different pose) and shape parameters S (shared across views) which we concatenate to form B.

To use depth and color together, we must know the precise extrinsic relationship between the sensors; due to manufacturing variance, the camera axes are not perfectly aligned. Instead of using a pre-calibration step, we pose the camera translation estimation as part of the optimization, using the human body itself to find the translation, T, between color and depth cameras.

Our data terms includes a color term E_C, a depth term E_D, and feet-to-floor contact term E_F. Our regularization terms include a pose prior E_P, a shape prior E_S (both Gaussian), and smoothness prior E_Q on per-vertex albedo:

$$E = E_C + E_D + E_F + E_P + E_S + E_Q. \tag{1}$$

The color term accumulates per-pixel error over images

$$E_C(I, A_L, A_C, T, B) = \sum_i \sum_u \|I_{iu} - \tilde{I}_{iu}(A_L, A_C, T, B)\|^2 \tag{2}$$

where $\tilde{I}_{ui}$ is the simulated pixel intensity of image-space position u for view i.

The depth term is similar but, due to sensor noise, is formulated robustly

$$E_D(D, T, B) = \sum_i \sum_u \|D_{iu} - \tilde{D}_{iu}(T, B)\|^\rho \tag{3}$$

where the parameter ρ is adjusted from 2 to 1 over the course of an optimization.

The floor term E_F minimizes differences between foot vertices of the model and the ground

$$E_F(D, B) = \sum_k \|r(B, D_b, k)\|^2 \tag{4}$$

where $r(B, D_b, k)$ indicates the distance between model footpad vertex k and a mesh D_b constructed from the background shot,

The albedo smoothness term E_Q penalizes squared differences between the log albedo of neighboring mesh vertices

$$E_Q = \sum_e \|\log(b(e, 0)) - \log(b(e, 1))\|^2 \tag{5}$$

where $b(e, 0)$ denotes the albedo of the first vertex on edge e, and $b(e, 1)$ denotes the albedo of the other vertex on edge e.

Finally, shape and pose parameter priors, $E_S(S)$ and $E_P(P)$, penalize the squared Mahalanobis distance from the mean body shape and pose learned during BlendSCAPE training.

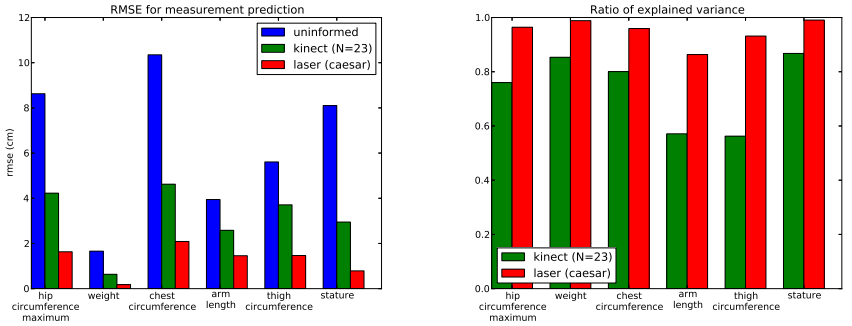

Fig. 8. Accuracy of measurement prediction for Kinect-based fitting compared to measurements from CAESAR scans or guessing the mean (uninformed). Left: root mean squared error (RMSE) in cm. Right: percentage of explained variance.

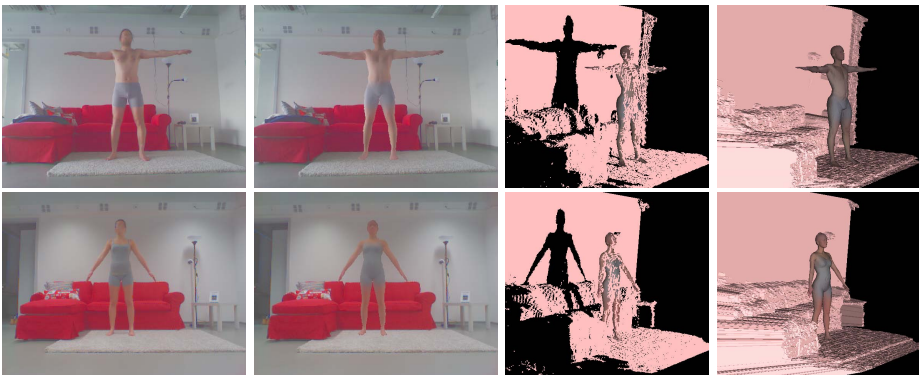

Fig. 9. Reconstruction of two subjects (top and bottom). **First column:** original captured images, with faces blurred for anonymity. **Second column:** simulated images after convergence. **Third column:** captured point cloud together with estimated body model. **Fourth column:** estimated body shown on background point cloud. More examples can be found in the supplemental materials.

Initialization for the position of the simulated body could be up to a meter away from the real body and still achieve convergence. Without the use of Gaussian pyramids or background images, initialization would require more precision (while we did not use it, initialization could be obtained with the pose information available from the Kinect API).

Male and female body models were each trained from approximately 2000 scans from the CAESAR [28] dataset. This dataset comes with anthropometric measurements for each subject; similar to [1], we use regularized linear regression to predict measurements from our underlying body shape parameters. To evaluate accuracy of the recovered body models, we measured RMSE and percentage of explained variance of our predictions as shown in Fig. 8. For comparison, Fig. 8 also shows the accuracy of estimating measurements directly from 3803

meshes accurately registered to the CAESAR laser scans. Although these two settings (23 subjects by Kinect and 3803 subjects by laser scan) differ in both subjects and method, and we do not expect Kinect scans to be as accurate, Fig. 8 provides an indication of how well the Kinect-based method works.

Figure 9 shows some representative results from our Kinect fitter; see the supplemental material for more. While foot posture on the male is slightly wrong, the effects of geometry, lighting and appearance are generally well-estimated. Obtaining this result was made significantly easier with a platform that includes a differentiable renderer and a set of building blocks to compose around it.

Each fit took around 7 minutes on a 3.0 GHz 8-core 2013 Mac Pro.

7 Conclusions

Many problems in computer vision have been solved by effectively differentiating through the rendering process. This is not new. What is new is that we provide an easy to use framework for both renderer differentiation and objective formulation. This makes it easy in Python to define a forward model and optimize it. We have demonstrated this with a challenging problem of body shape estimation from image and range data. By releasing the OpenDR with an open-source license (see `http://open-dr.org`), we hope to create a community that is using and contributing to this effort. The hope is that the this will push forward research on vision as inverse graphics by providing tools to make working on this easier.

Differentiable rendering has its limitations. When using differences between RGB Gaussian pyramids, the fundamental issue is overlap: if a simulated and observed object have no overlap in the pyramid, the simulated object will not record a gradient towards the observed one. One can use functions of the pixels that have no such overlap restriction (e.g. moments) to address this but the fundamental limitation is one of visibility: a real observed feature will not pull on simulated features that are entirely occluded because of the state of the renderer.

Consequently, differentiable rendering is only one piece of the puzzle: we believe that informed sampling [15] and probabilistic graphics programming [22] are also essential to a serious application of inverse rendering. Despite this, we hope many will benefit from the OpenDR platform.

Future exploration may include increasing image realism by incorporating global illumination. It may also include more features of modern rendering pipelines (for example, differentiation through a fragment shader). We are also interested in the construction of an "integratable renderer" for posterior estimation; although standard sampling methods can be used to approximate such an integral, there may be graphics-related techniques to integrate in a more direct fashion within limited domains.

Acknowledgements. We would like to thank Eric Rachlin for discussions about Chumpy and Gerard Pons-Moll for proofreading.

References

1. Allen, B., Curless, B., Popović, Z.: The space of human body shapes: Reconstruction and parameterization from range scans. ACM Trans. Graph. 22(3), 587–594 (2003)
2. Bastian, J.W.: Reconstructing 3D geometry from multiple images via inverse rendering. Ph.D. thesis, University of Adelaide (2008)
3. Baumgart, B.G.: Geometric modeling for computer vision. Tech. Rep. AI Lab Memo AIM-249, Stanford University (Oct 1974)
4. Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., Bengio, Y.: Theano: a CPU and GPU math expression compiler. In: Proceedings of the Python for Scientific Computing Conference (SciPy) (June 2010)
5. Bertero, M., Poggio, T., Torre, V.: Ill-posed problems in early vision. Proc. IEEE 76(8), 869–889 (1988)
6. Blanz, V., Vetter, T.: A morphable model for the synthesis of 3D faces. In: Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1999, pp. 187–194. ACM Press/Addison-Wesley Publishing Co., New York (1999)
7. Bradski, G., Kaehler, A.: Learning OpenCV. O'Reilly Media Inc. (2008), `http://oreilly.com/catalog/9780596516130`
8. Goodman, N., Mansinghka, V., Roy, D., Bonawitz, K., Tenenbaum, J.: Church: A language for generative models. In: McAllester, D.A., Myllymäki, P. (eds.) Proc. Uncertainty in Artificial Intelligence (UAI), pp. 220–229 (July 2008)
9. Grenander, U.: Lectures in Pattern Theory I, II and III: Pattern Analysis, Pattern Synthesis and Regular Structures. Springer, Heidelberg (1976–1981)
10. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. Frontiers in Appl. Math., vol. 19. SIAM, Philadelphia (2000)
11. Heikkila, J., Silven, O.: A four-step camera calibration procedure with implicit image correction. In: Proceedings of 1997 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pp. 1106–1112 (June 1997)
12. Hirshberg, D.A., Loper, M., Rachlin, E., Black, M.J.: Coregistration: Simultaneous alignment and modeling of articulated 3D shape. In: Fitzgibbon, A., Lazebnik, S., Perona, P., Sato, Y., Schmid, C. (eds.) ECCV 2012, Part VI. LNCS, vol. 7577, pp. 242–255. Springer, Heidelberg (2012)
13. Horn, B.: Understanding image intensities. Artificial Intelligence 8, 201–231 (1977)
14. Jalobeanu, A., Kuehnel, F., Stutz, J.: Modeling images of natural 3D surfaces: Overview and potential applications. In: Conference on Computer Vision and Pattern Recognition Workshop, CVPRW 2004, pp. 188–188 (June 2004)
15. Jampani, V., Nowozin, S., Loper, M., Gehler, P.V.: The informed sampler: A discriminative approach to Bayesian inference in generative computer vision models. CoRR abs/1402.0859 (February 2014), `http://arxiv.org/abs/1402.0859`
16. Jones, E., Oliphant, T., Peterson, P., et al.: SciPy: Open source scientific tools for Python (2001), `http://www.scipy.org/`
17. Jones, M.J., Poggio, T.: Model-based matching by linear combinations of prototypes. In: A.I. Memo 1583, pp. 1357–1365. MIT (1996)
18. Kersten, D.: Inverse 3-D graphics: A metaphor for visual perception. Behavior Research Methods, Instruments & Computers 29(1), 37–46 (1997)
19. Knill, D.C., Richards, W.: Perception as Bayesian Inference. The Press Syndicate of the University of Cambridge, Cambridge (1996)

20. Gorce, M.d.L., Paragios, N., Fleet, D.J.: Model-based hand tracking with texture, shading and self-occlusions. In: 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2008), Anchorage, Alaska, USA, June 24-26. IEEE Computer Society (2008)
21. Lee, A.D.: ad: a python package for first- and second-order automatic differentation (2012), http://pythonhosted.org/ad/
22. Mansinghka, V., Kulkarni, T.D., Perov, Y.N., Tenenbaum, J.: Approximate Bayesian image interpretation using generative probabilistic graphics programs. In: Burges, C., Bottou, L., Welling, M., Ghahramani, Z., Weinberger, K. (eds.) Advances in Neural Information Processing Systems 26, pp. 1520–1528 (2013)
23. Minka, T., Winn, J., Guiver, J., Knowles, D.: Infer.NET 2.4. Microsoft Research Cambridge (2010), http://research.microsoft.com/infernet.
24. Mumford, D.: Neuronal architectures for pattern-theoretic problems. In: Koch, C., Davis, J.L. (eds.) Large-scale Neuronal theories of the Brain, pp. 125–152. Bradford (1994)
25. Oliphant, T.E.: Python for scientific computing. Computing in Science and Engineering 9(3), 10–20 (2007)
26. Patow, G., Pueyo, X.: A survey of inverse rendering problems. Computer Graphics Forum 22(4), 663–687 (2003)
27. Richards, W.: Natural Computation. The MIT Press (A Bradford Book), Cambridge (1988)
28. Robinette, K., Blackwell, S., Daanen, H., Boehmer, M., Fleming, S., Brill, T., Hoeferlin, D., Burnsides, D.: Civilian American and European Surface Anthropometry Resource (CAESAR) final report. Tech. Rep. AFRL-HE-WP-TR-2002-0169, US Air Force Research Laboratory (2002)
29. Smelyansky, V.N., Morris, R.D., Kuehnel, F.O., Maluf, D.A., Cheeseman, P.: Dramatic improvements to feature based stereo. In: Heyden, A., Sparr, G., Nielsen, M., Johansen, P. (eds.) ECCV 2002, Part II. LNCS, vol. 2351, pp. 247–261. Springer, Heidelberg (2002)
30. Sminchisescu, C., Telea, A.: A framework for generic state estimation in computer vision applications. In: International Conference on Computer Vision Systems (ICVS), pp. 21–34 (2001)
31. Sminchisescu, C.: Estimation algorithms for ambiguous visual models: Three dimensional human modeling and motion reconstruction in monocular video sequences. Ph.D. thesis, Inst. National Polytechnique de Grenoble (July 2002)
32. Terzopoulos, D.: Regularization of inverse visual problems involving discontinuities. IEEE PAMI 8(4), 413–424 (1986)
33. Terzopoulos, D.: Physically-based modeling: Past, present, and future. In: ACM SIGGRAPH 89 Panel Proceedings, pp. 191–209 (1989)
34. Viola, F., Fitzgibbon, A., Cipolla, R.: A unifying resolution-independent formulation for early vision. In: 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 494–501 (June 2012)
35. Walter, S.F.: Pyadolc 2.4.1 (2012), https://github.com/b45ch1/pyadolc
36. Weiss, A., Hirshberg, D., Black, M.: Home 3D body scans from noisy image and range data. In: Int. Conf. on Computer Vision (ICCV), pp. 1951–1958. IEEE, Barcelona (2011)
37. Wood, F., van de Meent, J.W., Mansinghka, V.: A new approach to probabilistic programming inference. In: Artificial Intelligence and Statistics (2014)