J Vis (2017) 20:195–204
DOI 10.1007/s12650-016-0369-3

REGULAR PAPER

Koki Nagano · Thomas Collins · Chi-An Chen · Aiichiro Nakano

Massively parallel inverse rendering using Multi-objective Particle Swarm Optimization

Received: 30 November 2015 / Revised: 12 March 2016 / Accepted: 28 April 2016 / Published online: 10 June 2016

Abstract We present a novel GPU-accelerated per-pixel inverse rendering optimization algorithm based on Particle Swarm Optimization (PSO). Our algorithm estimates the per-pixel scene attributes—including reflectance properties—of a 3D model, and is fast enough to do in situ visualization of the optimization in real-time. The algorithm's high parallel efficiency is demonstrated through our GPU/GLSL shader implementation of the method. IRPSO is validated experimentally on simulated ground truth images, while a suite of tests performed on the University of Southern California's High Performance Computing Center cluster provides strong evidence that our method can scale to larger, more difficult inverse rendering problems.

Keywords Inverse rendering · Particle Swarm Optimization · GPU acceleration · In situ visualization

1 Introduction

Photorealistic rendering of real-world objects has been a long-standing goal in computer graphics. It is a fundamentally difficult problem that requires a detailed representation of 3D scene geometry, a physically accurate model of the reflectance properties of the materials in the scene, and a sophisticated light transport model. Together, we refer to these models as a Given a rendering model and an instantiation of its parameters, forward rendering (FR) is the process of specifying the pixel color at each pixel coordinate in an image. When these parameters are known a priori, a host of deterministic and stochastic techniques exist to generate convincing renderings of 3D scenes.

Inverse rendering (IR), on the other hand, is the process of recovering or estimating the unknown attributes of a scene (e.g., illumination, reflectance parameters) given a reference image such that the image generated by the rendering model (instantiated with the recovered parameters) matches the reference image as closely as possible. Once these parameters are recovered, they can be used to realistically render the 3D model with the material under novel lighting conditions, in novel camera views, and so on (Fig. 1).

Current state-of-the-art rendering techniques leverage 3D scanning techniques to record accurate 3D geometry as well as reflectance properties such as per-pixel surface orientation, surface and subsurface

K. Nagano (✉) · T. Collins · C.-A. Chen · A. Nakano
University of Southern California, Los Angeles, CA, USA
E-mail: knagano@usc.edu

T. Collins
E-mail: collinst@usc.edu

C.-A. Chen
E-mail: chianc@usc.edu

A. Nakano
E-mail: anakano@usc.edu

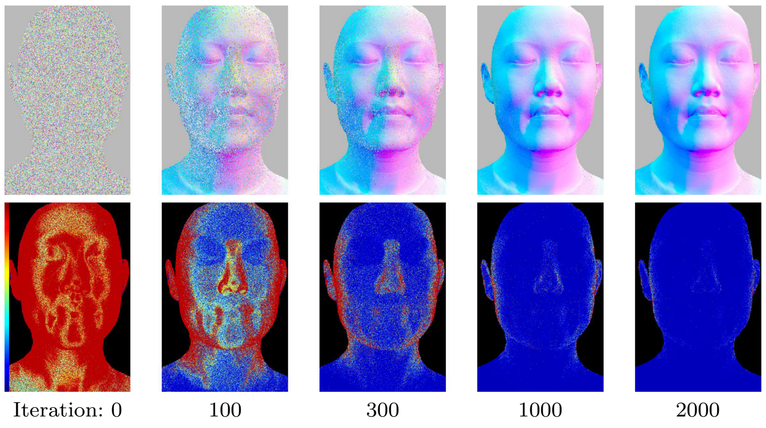

Fig. 1 Per-pixel estimated normal (*top*), and corresponding error visualization (*bottom*) with iterations 0, 100, 300, 1000, and 2000

reflectance, etc. (Ghosh et al. 2011; Weyrich et al. 2006) up to sub-millimeter resolution. However, significant manual effort is still required to produce photorealistic renderings from the data. This is primarily due to the fact that the measured reflectance data comprise only a subset of the per-pixel reflectance properties necessary to create physically plausible renderings. Hence more research is necessary to investigate a semi-automatic method to estimate complete per-pixel reflectance properties in a way that scales up to such high-resolution data.

The primary difficulty inherent in this problem is the sheer number of parameters to be estimated, which is easily on the order of millions, necessitating a novel framework that can handle this computational burden in an efficient manner. In this work, we exploit current graphics hardware to accelerate the computation in a massively parallel way. We propose a GPU-accelerated, inverse rendering (IR) algorithm, called IRPSO (Inverse Rendering using Particle Swarm Optimization), which estimates the per-pixel reflectance parameters of a realistic rendering model using an optimization approach based on Particle Swarm Optimization (PSO) (Eberhart and Kennedy 1995). Based on Nagano et al. (2015), we provide additional new results, and performed extensive new analyses, validation and scalability tests.

2 Related work

2.1 Inverse rendering

A significant body of work has been done in the category of image-based techniques. Given photographs, and rendering models that define how the light interacts with the scene, unknown scene geometry (Debevec et al. 1996), reflectance parameters, and/or lighting (Marschner 1998) can be solved, even under global illumination (Yu et al. 1999). The quality and detail of the acquired unknown parameters is largely limited by the quality of the measured data, the fidelity of the presumed rendering model, and known attributes of the scene. Since it is very costly to simulate the entire light transport process (including global illumination), entire global illumination may not be considered and/or people have been using techniques to separate the response of the light on the material to bring down the complexity of the light transport model necessary to solve for the unknown parameters such as with spherical harmonics (Ramamoorthi and Hanrahan 2001). People also employed a gradient-based optimization including (Donner et al. 2008) that estimated per-pixel subsurface reflectance. However they are limited to a subset of surface and subsurface reflectance parameters, unable recover spatially varying parameters, and/or the result is shown on a rather small patch. It is still non-trivial to scale up the existing method to acquire per-pixel reflectance with the entire material—which requires estimating or optimizing on the order of millions of points with potentially high dimensional parameters—present in real-world materials, and some research needs to be done in efficiently optimizing such large numbers of parameters.

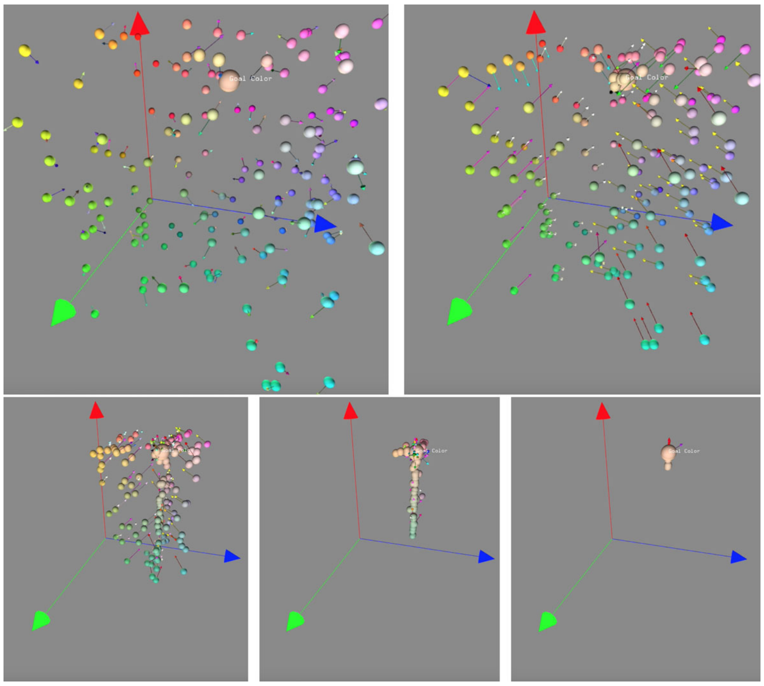

Fig. 2 A visualization of the PSO algorithm optimizing in three-dimensional RGB space converging to the color (labeled "Goal Color") that minimizes the fitness function

2.2 Particle Swarm Optimization

Particle Swarm Optimization (PSO) (Eberhart and Kennedy 1995) is a swarm-based optimization algorithm that has been shown to be effective in solving difficult optimization problems in many diverse domains (Poli 2008; Sun et al. 2011). The basic idea of PSO is that a swarm of m particles, each n-dimensional, perform an independent search in the space of possible n-dimensional solutions. Each particle i is a point x_i in this search space with a certain velocity v_i and has an associated fitness given by the objective function (F, to be minimized) value at that point $F(x_i)$. Particles move around in this search space randomly with sampling biased toward a random weighted average of the best (lowest $F(x_i)$) position achieved by any particle in the swarm g and the best position achieved by each particle individually, p_i. This focuses random searches on areas of the search space where a global optimum is expected to be. g and p_i are updated at each iteration.

This randomized searching process continues until a certain fitness threshold h is reached by some particle in the swarm (i.e., a low enough value of F is found) or a maximum number of iterations N is performed. At termination, the global best position found (g) is returned. Figure 2 visualizes this procedure in RGB color space.

Many real-world problems are most readily described as multi-objective optimization problems, in which a set of objective functions must be simultaneously minimized in the same search space. Such problems can be solved using a Multi-objective version of Particle Swarm Optimization (MOPSO).

Many different MOPSO frameworks have been proposed in order to manage the increased computational and mathematical difficulty that comes from trying to minimize many (possibly conflicting) objective functions simultaneously. For our purposes, the most relevant MOPSO approaches are those that devote one swarm to optimizing each objective function [Vector-evaluated MOPSO; (Chow and Tsui 2004; Parsopoulos and Vrahatis 2004)] as they naturally lend themselves to parallelism. An overview of the literature on MOPSO is given in Parsopoulos and Vrahatis (2008).

To the best of our knowledge, there is no previous work regarding the application of PSO or MOPSO to the inverse rendering problem. However, there is considerable experimental evidence that PSO and MOPSO are powerful optimization frameworks that provide high-quality solutions to very difficult optimization problems. We detail a massively parallel MOPSO solution to the IR problem in the next section.

3 Multi-objective PSO for inverse rendering

3.1 Algorithm overview

Let $\mathcal{I}$ be the reference image input into the algorithm. We can view $\mathcal{I}$ as a width $\times$ height matrix or texture of pixels. Let R be our rendering model, which requires n unknown values at each pixel to be fully instantiated and generate a best approximation image, which we call $\mathcal{T}$. The reference pixel color value at each pixel of $\mathcal{I}$ is $c_{[i,j]}$. The set of reference pixels $c_{[i,j]}$ (reference image $\mathcal{I}$) is given as known together with known camera intrinsic and extrinsic parameters which map the coordinates between the model and the reference camera if necessary.

R has q parameters, which, when instantiated, determine the rendered image. $r \leq q$ of these parameters are known at each pixel, while the remaining $n = q - r$ parameters are unknown at each pixel. Denote the set of known and unknown rendering model parameters $\hat{\mathbf{p}}_{[i,j]}$. Then we have that $R(\hat{\mathbf{p}}_{[i,j]}) = \hat{c}_{[i,j]}$. In other words, given an instantiation of all known and unknown parameters, rendering model R generates an output image $\mathcal{T}$, which we can compare with the ground truth image $\mathcal{I}$.

The inverse rendering problem is that of recovering the unknown $n = q - r$ parameters in $\hat{\mathbf{p}}_{[i,j]}$ at each pixel that minimize the total amount of error between the generated image $\mathcal{T}$ and the ground truth image $\mathcal{I}$. This reduces to solving the following minimization problem at each pixel:

$$\operatorname{argmin}_{\hat{\mathbf{p}}_{[i,j]}} \|\hat{c}_{[i,j]} - c_{[i,j]}\| \tag{1}$$

The multi-objective optimization problem at hand is to find a set of $n \times$ height $\times$ width parameters that simultaneously minimizes the errors at each pixel. We assume that each pixel's parameters can be optimized separately using PSO, but allow for interaction between spatially close swarms using a special term in the PSO fitness function (described below).

The IRPSO algorithm (illustrated in Fig. 3) begins by initializing a swarm of m particles associated with each pixel(i, j). Once the particle swarms have been initialized, the following procedure is repeated until some stopping criterion is met (number of iterations, fitness threshold, etc.):

1. In parallel, a random particle from each swarm at each pixel $P^k_{in}(i,j)$—with $k(\leq m)$ being the particle id—is selected and its position and velocity are updated to the associated data structure in $P^k_{out}(i,j)$ according to equations (2) and (3) below.
2. In parallel, the personal best particles and global best particles for each swarm are updated to $P^k_{out}(i,j)$ as in the original single-objective PSO algorithm.
3. Once all threads are synchronized, return to step 1.

Though it may seem that this is simply a parallel evaluation of single-objective PSO procedures, it is truly an MOPSO approach as we use the following velocity and position update equations, which modify those in Chow and Tsui (2004) to operate over a regular grid of particle swarms:

$$\begin{aligned} v^l_{[i,j]_{k,t+1}} &= v^l_{[i,j]_{k,t}} + c_1 R^l_{1[i,j]_{k,t}} \left(p^l_{[i,j]_{k,t}} - x^l_{[i,j]_{k,t}}\right) \\ &\quad + c_2 R^l_{2[i,j]_{k,t}} \left(g^l_{[i,j]_t} - x^l_{[i,j]_{k,t}}\right) + c_3 A^l_{[i,j]_k} \end{aligned} \tag{2}$$

$$x^l_{[i,j]_{k,t+1}} = x^l_{[i,j]_{k,t}} + v^l_{[i,j]_{k,t+1}} \tag{3}$$

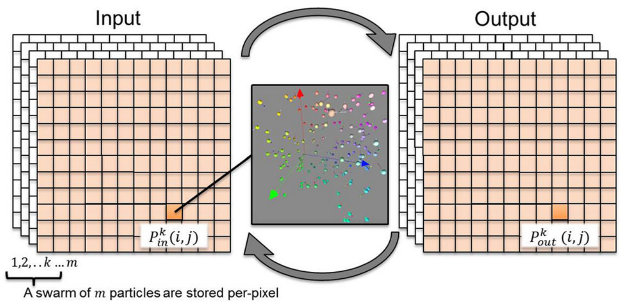

Fig. 3 A visual representation of the IRPSO framework

In Eq. 2, we update each dimension l of each n-dimensional particle k associated with pixel (i, j) at time t in order to generate a new velocity at time $t + 1$. c_1, c_2 and c_3 are constants (algorithm parameters, where $c_1 \approx c_2$ and $c_3 << c_1$), $R^l_{1[i,j]_{k,t}} \sim U(0, 1)$, and $R^l_{2[i,j]_{k,t}} \sim U(0, 1)$. The term $A^l_{[i,j]_k}$ at the end of Eq. 2 is defined as follows:

$$A^l_{[i,j]_k} = \left(\frac{1}{(H)^2} \sum_{m=0}^{H-1} \sum_{n=0}^{H-1} \left(g^l_{[i+(m-H/2),j+(n-H/2)]_t} \right) - x^l_{[i,j]_{k,t}} \right) \tag{4}$$

This term effectively biases the solution at each pixel (i, j) toward the average of the current global best positions of neighboring particle swarms in an $H \times H$ window centered at swarm (i, j). By increasing or decreasing H, users can determine the range of influence each swarm has on neighboring swarms during optimization. Thread synchronization must occur at each step of the algorithm so that we can ensure that swarms exchange up-to-date global best information with one another.

3.2 Implementation

In this work, we chose the GLSL shader to be the experimental platform to demonstrate the generality of our computational framework. The same algorithm can be implemented in an advanced compute shader, CUDA, OpenCL, and so on depending on applications. Taking advantage of the fact that we wish to associate a particle swarm with each pixel, we can use OpenGL textures to store PSO data vectors and values on a per-pixel (and thus per-swarm) basis. Assuming that each swarm has the same number of particles, we can store the position of one particle of each swarm in a width × height texture of 3D vectors. By allocating m position textures, each consisting of width × height 3D vectors, we can create a swarm of m particles per pixel. We can use this same data layout to store the velocities, best positions, and best fitnesses associated with each particle in m textures each. Similarly, the global best fitness of each swarm and the global best particle of each swarm can be stored in one texture each, as they are shared amongst all the particles in a swarm. We can then perform steps 1 and 2 of the algorithm described above in GLSL shaders during each rendering loop. Step 3 is performed by OpenGL automatically as the GPU threads are synchronized before the next rendering loop is completed.

4 Synthetic target image preparation and results

In order to verify the IRPSO algorithm, we tested it on synthetic ground truth images. In this way, we knew the values of the parameters to be recovered at each pixel and could accurately gauge the quality of the solutions produced by IRPSO.

The synthetic ground truth images were generated using a high-quality 3D mesh of a human head as well as per-pixel surface orientation maps and surface and subsurface albedo maps captured by a spherical LED scanning device (Ghosh et al. 2011). Then the head model was rendered with known camera parameters using GLSL shaders, evaluating diffuse and/or specular reflections under a point light source(s) using captured per-pixel reflectance maps, and per-pixel color is recorded in the ground truth images. For the ground truth images we rendered the 3D face mesh with a point light source sampled in 200 directions spread evenly over the sphere with 10 points in the latitude and 20 points in the longitude direction. Some of the representative frames are shown in Fig. 5. Then we treat the shader as a black box and solve for reflectance parameters, minimizing the difference between the ground truth rendering and the rendering made with estimated parameters.

Figure 1 visualizes how the optimization progresses with the estimation of the per-pixel normal (top row) with the corresponding error visualization shown in false color with high error values being red, and low blue (bottom row). Starting with a completely random guess, the optimization converges within 1000 iterations at most pixels. Figure 6a compares the ground truth rendering (top) and the rendering made with estimated parameters (bottom), and Fig. 6b shows the comparison between ground truth (top row) and our per-pixel estimation (middle row) with false color error visualization (bottom row) with diffuse color (first column), surface normal (second column), and surface roughness with a two lobe Cook–Torrance model (Cook and Torrance 1982) (third column). The ground truth diffuse color, and the normal are acquired from Ghosh et al. (2011), and the ground truth roughness parameters are manually painted based on measurement done per face region (Graham et al. 2013) with RGB channels storing two roughness values, and a convex weight between the lobes. It is interesting to note that some error is still visible in the error visualization around the edges due to the grazing angle, it may be fixed employing multi-view data.

5 Analysis and extensions

Figure 4 provides sample numerical errors encountered during an IRPSO run. These results were part of a set of tests we performed to numerically validate the proposed method. The leftmost image is the current IRPSO per-pixel estimated image. On this image, four pixels are marked and color-coded. The next three images show the numerical errors of each of these pixels—i.e., how far they are away from the known ground truth value—as a function of the number of IRPSO iterations performed when optimizing for (from left to right) diffuse color, specular normal, and two-lobe roughness parameters at each pixel. The error at each pixel during the optimization of each set of parameters converges very quickly to zero (Fig. 5).

To investigate how the algorithm scales as the number of pixel increases, we measured the amount of time required to complete 1000 iterations of algorithm using the surface roughness example as in Fig. 6. Figure 7 shows time (in seconds) to complete the task including real-time visualization with five different image sizes with width and height 150 by 225 (pixels), 300 by 450, 600 by 900, 1200 by 1800, 2400 by 3600. Note that as the image size doubles, the pixel counts quadruples, and the running time increases as a number of pixels to be optimized. In all five different images, we observed that the global average error was consistently about 0.1 % after 1000 iterations.

As was mentioned above, the decision to implement the proposed method in GLSL was one of convenience and simplicity. GLSL allowed us to implement and test our method without having to consider low-level details which might have made our results more difficult to decipher. This first prototype in this

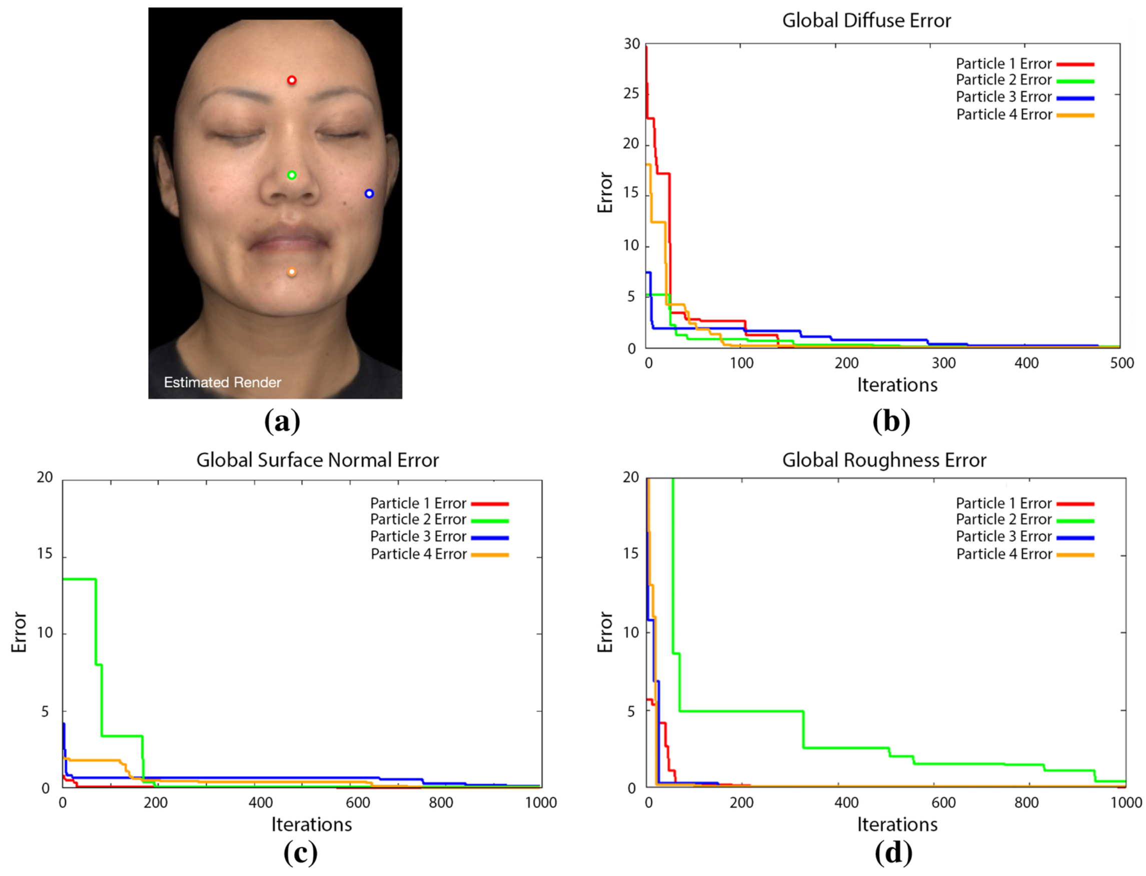

Fig. 4 Sample visualization of IRPSO numerical errors of four pixels. **a** IRPSO estimated image with four color-coded pixels. The next three images **b–d** shows the numerical errors of the associated color-coded pixels vs. the number of iterations of IRPSO when optimizing global diffuse color, surface normals, and two-lobe roughness

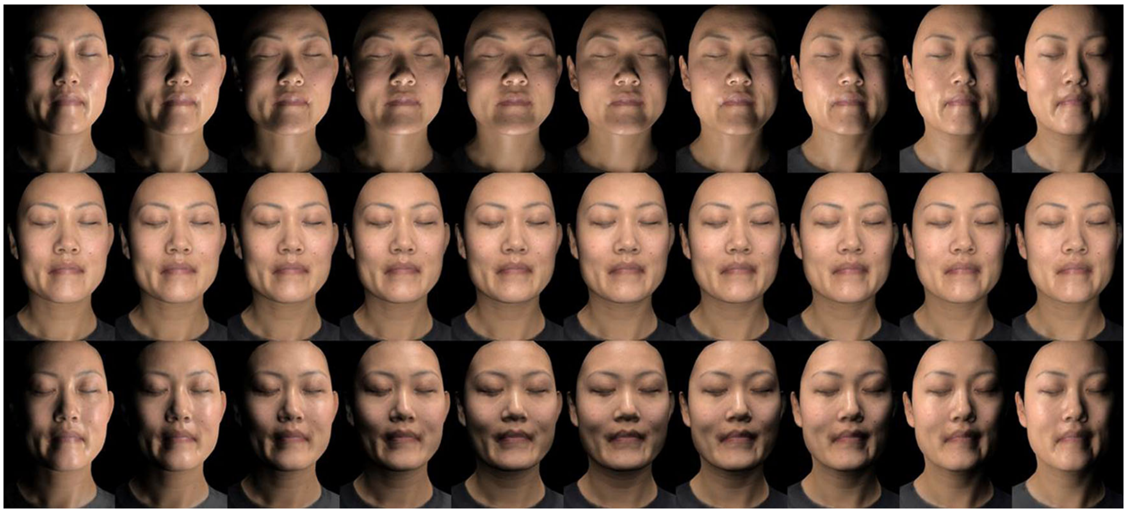

Fig. 5 Representative ground truth renderings as input to the optimization are shown

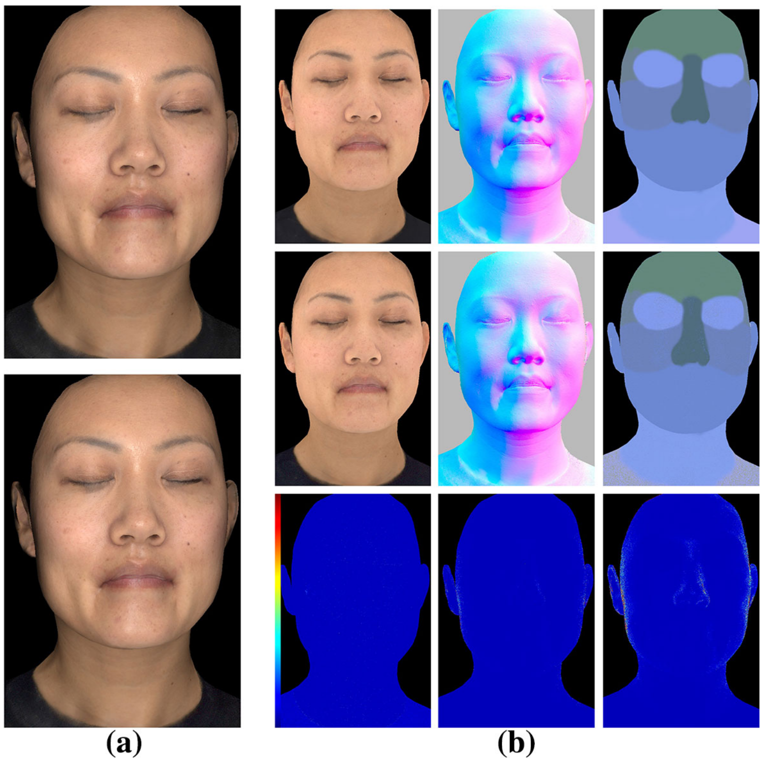

Fig. 6 **a** From *top* to *bottom* starting from completely random initialization, the rendering with estimated parameters converges. **b–d** Comparisons between ground truth (*top row*) and IRPSO optimized best-guess results (*middle row*) showing a maximum of less than 1 % error (*bottom row*)

paper was implemented, and tested on Windows 7 64 bit with a 16 core Intel Xeon E5620 processor, 24 GB RAM, and an Nvidia Quadro K6000 GPU. The fixed-structure parallelism and 2D grid layout offered by GLSL was particularly effective in this instance, since we were operating on a per-pixel basis with swarms communicating only with relatively nearby neighbors.

Implementing our framework in CUDA or OpenCL directly rather than OpenGL/GLSL would make it more general (as well as applicable to more high performance computing (HPC) environments), and this is something the authors are actively working on. However, a more fundamental problem is ensuring that our

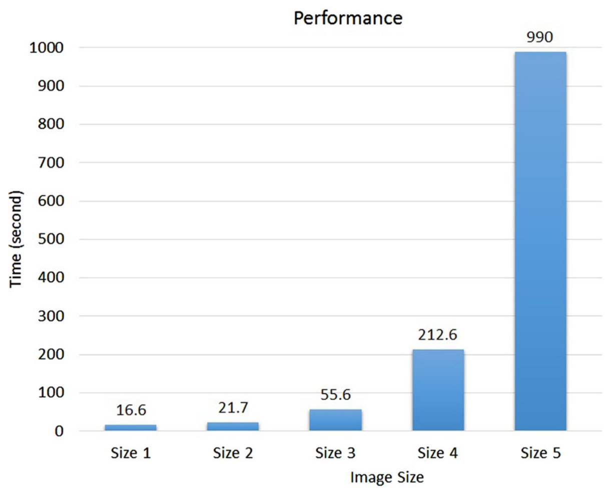

Fig. 7 The *column chart* shows how the algorithm running time scales with images in different pixel width and height with 150 × 225 (size 1), 300 × 450 (size 2), 600 × 900 (size 3), 1200 × 1800 (size 4), 2400 × 3600 (size 5)

framework is useful even in situations where the dimensionality of the parameters being solved at each pixel is high or the fitness function is extremely difficult to solve. This requires more computation at each pixel (i.e., more particles and more iterations). It is important that our framework still offer similar performance in such situations.

If we wish to maintain the same level of near-real-time performance as we have shown in the results above, we must exploit additional axes of parallelism. The most obvious way to do this is to use a parallel version of the PSO algorithm at each pixel, creating an additional layer of parallelism in our framework. If we consider the PSO algorithm, it is easy to see that each particle's position, best position, and velocity can be updated at each step in parallel. Once all the particles have been updated in parallel, a global reduction must be performed at the end of each iteration to find the best particle in the swarm. By continuing this procedure iteratively, we can easily create a parallel PSO implementation.

It is also important to discuss the implementation of this framework in more traditional HPC environments, rather than just on the GPU(s) of a single computer. The IRPSO framework is easily implementable on current HPC environments (assuming the availability of a regular 2D grid structure of *kl* computing nodes) using, e.g., MPI. IRPSO operates on an *mn* grid of pixels, each with an associated particle swarm (mn total swarms). Communication of swarm (*i*, *j*) with its neighbors is limited to a rectangular window (*hh*) around pixel (*i*, *j*). Thus, in an HPC implementation, the *mn* particle swarms should be distributed as evenly as possible amongst the *kl* available nodes in the computing grid, with node *t* responsible for performing each iteration of IRPSO for each swarm *s* in the rectangular grid of pixels assigned to it. Computing node *t* may need to communicate with nearby computing nodes to ensure that swarm *s* has access to the best known positions of swarms in its *hh* neighborhood (if some of those swarms are located on different computing nodes) at the beginning of each iteration. Finally, the massively parallel lock-step nature of the GLSL implementation can be easily replicated by synchronizing all the computing nodes in the 2D grid at the end of each iteration (e.g., with an MPI barrier). At the end of the algorithm or the end of each iteration that needs visualized, each computing node would send its current best-guess optimized pixel values for the rectangular subset of the image assigned to it to some master node. This image could then be stitched together at the master node and rendered. Further parallelization could potentially be performed by parallelizing each PSO execution on each swarm through multithreading or CUDA/OpenCL as described in the previous paragraph. This would allow for a hierarchical parallelization of the algorithm in an HPC environment, in which different computing nodes execute PSO in parallel on a subset of particle swarms, and each particle swarm is itself parallelized at each node.

To illustrate the feasibility of scaling up PSO tasks at each pixel in an HPC environment implementation, we have implemented and tested MPI, OpenMP, and CUDA PSO parallelization schemes. These parallelization techniques were tested using standard benchmark functions on the University of Southern Californias High Performance Computing Center cluster (HPCC) on nodes with Dual Quadcore AMD Opteron

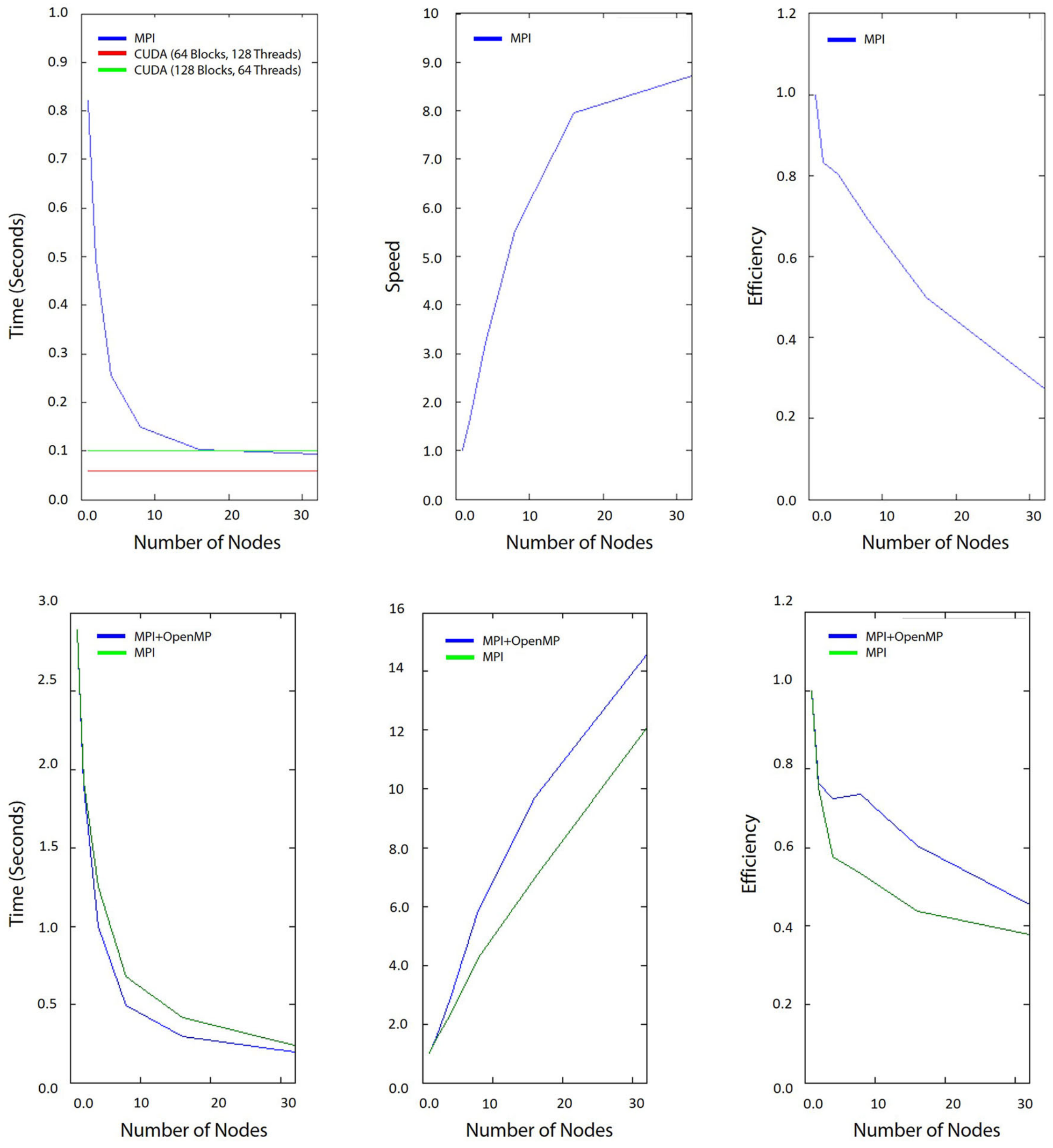

Fig. 8 PSO scalability results generated using the USC High-Performance Computing Center cluster. The top row shows the runtime, speedup and efficiency of an MPI version of PSO with 8192 particles as a function of the number of computing nodes allocated to it. The timing results of two CUDA variants of PSO with 8192 particles are included for reference. The *bottom row* shows the runtime, speedup and efficiency of the MPI and hybrid MPI+OpenMP versions of PSO with 50,000 particles as a function of the number of processors allocated to it

2.3 GHz processors with 16 GB RAM and an NVIDIA K20 Kepler GPU. The results are shown in Fig. 8. They illustrate the high parallel efficiency of PSO. This provides strong evidence that such an HPCC implementation with hierarchical parallelization would be effective for our framework and allow it to scale up to more realistic and difficult inverse rendering problems. We are currently working on the experiments necessary to validate these theoretical results. An MPI with CUDA implementation of the proposed algorithm is currently being developed and will be tested on the University of Southern Californias High Performance Computing Center.

6 Conclusions and future work

In this paper, we presented a novel, GPU-accelerated inverse rendering algorithm based on Multi-objective Particle Swarm Optimization. The use of a parallelized Multi-objective PSO algorithm was motivated by PSO and MOPSO's proven track record in the literature of efficiently solving (to a high-quality approximation) difficult real-world optimization problems. We validated the IRPSO algorithm using ground truth images of a realistic, high-quality 3D model of a head and a complex skin texture. IRPSO was successfully able to per-pixel diffuse color reflectance parameters, per-pixel surface roughness, and per-pixel surface normals, while interactively visualizing the optimization. The composite optimized best-guess images were visually indistinguishable from the ground truth images after only a few seconds of optimization time. For the next step, we wish to apply our method to real data sets in order to observe its real-world performance. It would also be of interest to compare the PSO algorithm with other optimization methods, such as a gradient-based methods both in terms of the number of parameters and the smoothness of the target function. GLSL implementation allowed for a straightforward texture-based implementation of grids of particle swarms. It is also an interesting example of large scale, general purpose computation in GLSL shaders. However, as a next step, we wish to make this method more general by developing a CUDA or OpenCL implementation. This would eliminate some of the restrictions present in the GLSL implementation and would likely lead to better overall performance. Such an implementation will also allow us to explore the multi-layer, hierarchical parallelization discussed in Sect. 5, which will hopefully lead to drastic improvements in the speed and scalability of the method.

Acknowledgments This work was developed from a final project for CSCI 596 (scientific computing and visualization) course at the University of Southern California. Integration of research and education in CSCI 596 was supported by the National Science Foundation, Grant No. 1508131. We also thank the USC Institute for Creative Technologies for the support of the work.

References

Chow CK, Tsui HT (2004) Autonomous agent response learning by a multi-species particle swarm optimization. In: IEEE CEC2004 congress on evolutionary computation, 2004, vol 1, pp 778–785

Cook RL, Torrance KE (1982) A reflectance model for computer graphics. ACM TOG 1(1):7–24

Debevec PE, Taylor CJ, Malik J (1996) Modeling and rendering architecture from photographs: a hybrid geometry- and image-based approach. In: Proceedings of SIGGRAPH 96, computer graphics proceedings, annual conference series, pp 11–20

Donner C, Weyrich T, d'Eon E, Ramamoorthi R, Rusinkiewicz S (2008) A layered, heterogeneous reflectance model for acquiring and rendering human skin. In: ACM TOG (Proceedings of SIGGRAPH Asia)

Eberhart R, Kennedy J (1995) A new optimizer using particle swarm theory. In: Proceedings of the sixth international symposium on micro machine and human science, 1995, MHS '95, pp 39–43. doi:10.1109/MHS.1995.494215

Ghosh A, Fyffe G, Tunwattanapong B, Busch J, Yu X, Debevec P (2011) Multiview face capture using polarized spherical gradient illumination. In: Proceedings of the 2011 SIGGRAPH Asia conference, SA '11. ACM, New York, pp 129:1–129:10. doi:10.1145/2024156.2024163

Graham P, Tunwattanapong B, Busch J, Yu X, Jones A, Debevec P, Ghosh A (2013) Measurement-based synthesis of facial microgeometry. Comput Graphics Forum 32(2pt3):335–344. doi:10.1111/cgf.12053

Marschner S (1998) Inverse rendering for computer graphics. Ph.D. thesis, Cornell University

Nagano K, Collins T, Chen CA, Nakano A (2015) GPU-based inverse rendering with multi-objective particle swarm optimization. In: SIGGRAPH Asia 2015 visualization in high performance computing, SA '15. ACM, New York, pp 8:1–8:4. doi:10.1145/2818517.2818523

Parsopoulos KE, Vrahatis MN (2004) On the computation of all global minimizers through particle swarm optimization. IEEE Trans Evol Comput 8(3):211–224

Parsopoulos KE, Vrahatis MN (2008) Multi-objective particles swarm optimization approaches. In: Thu Bui L, Alam S (eds) Multi-objective optimization in computational intelligence: theory and practice. IGI Global, Hershey, pp 20–42

Poli R (2008) Analysis of the publications on the applications of particle swarm optimisation. J Artif Evol App 4:1–4:10. doi:10.1155/2008/685175

Ramamoorthi R, Hanrahan P (2001) A signal-processing framework for inverse rendering. In: Proceedings of the 28th annual conference on computer graphics and interactive techniques, SIGGRAPH '01. ACM, New York, pp 117–128. doi:10.1145/383259.383271

Sun J, Lai C, Wu X (2011) Particle swarm optimisation: classical and quantum perspectives. Chapman & Hall, London

Weyrich T, Matusik W, Pfister H, Bickel B, Donner C, Tu C, McAndless J, Lee J, Ngan A, Jensen HW, Gross M (2006) Analysis of human faces using a measurement-based skin reflectance model. ACM Trans Graph 25(3):1013–1024

Yu Y, Debevec P, Malik J, Hawkins T (1999) Inverse global illumination: recovering reflectance models of real scenes from photographs. In: SIGGRAPH '99, pp 215–224