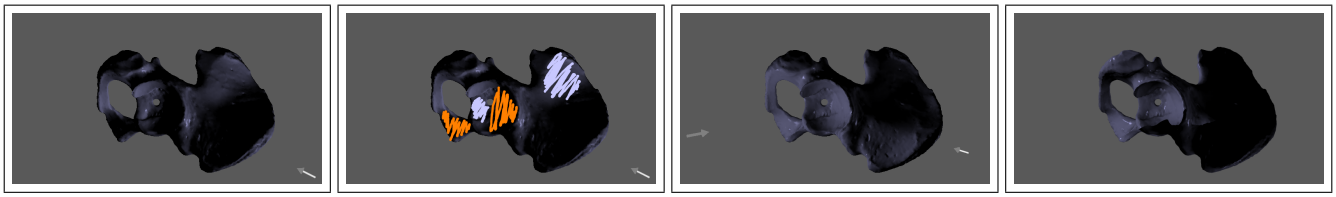
Crayon Lighting: Sketch-guided Illumination of Models

Amit Shesh and Baoquan Chen*
University of Minnesota–Twin Cities



(a) (b) (c) (d)

Figure 1: *An example output. (a) the original hip model with 40,000 triangles. (b) The user uses orange and blue strokes to bring the cavity into focus and recede the rear part by darkening it. (c) a sample output produced by our system by moving the existing light and adding a new light. This image is rendered using conventional* OpenGL *rendering. (d) compliance with the input is reinforced by this raytraced image of the same model under the same lighting conditions, with shadowing effects.*

Abstract

An interactive and intuitive way of designing lighting around a model is desirable in many applications. In this paper, we present a tool for interactive inverse lighting in which a model is rendered based on sketched lighting effects. To specify target lighting, the user freely sketches bright and dark regions on the model as if coloring it with crayons. Using these hints and the geometry of the model, the system efficiently derives light positions, directions, intensities and spot angles, assuming a local point-light based illumination model. As the system also minimizes changes from the previous specifications, lighting can be designed incrementally. We formulate the inverse lighting problem as that of an optimization and solve it using a judicious mix of greedy and minimization methods. We also map expensive calculations of the optimization to graphics hardware to make the process fast and interactive. Our tool can be used to augment larger systems that use point-light based illumination models but lack intuitive interfaces for lighting design, and also in conjunction with applications like ray tracing where interactive lighting design is difficult to achieve.

CR Categories: I.3.6 [Computing Methodologies]: Computer Graphics—Methodologies and Techniques; I.3.2 [Computing Methodologies]: Computer Graphics—Graphics Systems

Keywords: Inverse lighting, sketch-based input, optimization

1 Introduction

Lighting enhances structural details to depict the geometry of a model well in a single image. As lighting plays an integral role in almost any graphics application, an interactive and easy-to-use method to design lighting around a given model is highly desirable. However, a given 3D model is often lit by moving lights around and "playing" with their colors and intensities in a largely trial-and-error fashion, usually in the form of non-intuitive tweaking of actual numbers or an interface to move lights in three dimensions and change their properties (which has its own shortcomings). In this paper, we present a tool, Crayon Lighting, that sets up appropriate lighting for a 3D model/scene using rough sketchy inputs. Our focus is to set up lighting quickly with intuitive user input.

Although the user may have a fair idea of how he/she wants the lit model to look and what features are to be enhanced (effect), developing an intuition for which placement of lights causes these effects is difficult. This cause-effect relationship becomes even more complex with multiple and diverse sources of light. The problem of designing even simple shadows significantly complicates the problem, as shadows are much more complex functions of geometry and light properties than highlighting is. In order to alleviate the user from "thinking technically" about the physics of lighting, our user input is goal-oriented–the user sketches desired lighting effects while the program determines lighting parameters from them.

Inverse lighting is an old problem and has been researched well in the context of many diverse applications like architectural lighting design [Kawai et al. 1993], visualization [Lee et al. 2004], animation and production rendering [Marks et al. 1997; Pellacini et al. 2007], etc. While some lighting methods target automatic lighting design [Shacked and Lischinski 2001], others make the process interactive by making the lighting interactive to facilitate real-time manual movement of lights [Kristensen et al. 2005], offering lighting cues [Schoeneman et al. 1993; Poulin et al. 1997] or even sketching [Pellacini et al. 2007]. However a perusal of previous work brings forth the following problems that are addressed only partially: (1) formulating the lighting problem so that it can be applied widely across applications, (2) optimization techniques that have good convergence, good quality results and also work at interactive rates, (3) tradeoff between choice of illumination model and quality of results and interactivity, and (4) intuitive user interfaces that can be learned easily (5) ability to work in conjunction with existing software (i.e. popular modeling systems) that use lighting but do not offer interfaces geared towards designing it. Our tool ad-

*e-mail: ashesh | baoquan@cs.umn.edu

dresses all these problems and targets naive users who would rather not think about the physics of light transport while achieving the lighting they desire. Barring some optional parameter specification, the user input is restricted to sketching on a model.

In this paper, we design a tool that can be used to design lighting that uses local point-light based illumination models (like *OpenGL* lighting). Such a tool can be used in conjunction with many modeling tools that offer the option of lighting, but do not offer a good interface for designing it. Another application of such a tool is to design lighting quickly and use the results in an expensive rendering algorithm (like ray tracing) that inherently does not support changing lights interactively for design purposes. Our tool solves for various lighting properties like positions, directions, spot angles and intensities. It offers a unique feature of designing lighting while preserving existing lighting conditions, which allows lighting to be designed incrementally. Our optimization framework is general enough to be applied to a wide range of lighting models, although all results in this paper use only one such model. We achieve interactivity by solving this optimization problem using a judicious mix of greedy and conventional minimization methods, and delegating expensive operations in the optimization to graphics hardware.

Crayon Lighting works as follows: the user starts by loading in a model that is lit using a default lighting (Figure 1(a)). The user uses an orange highlighting pen to sketch highlights and a blue darkening pen for darkening parts of geometry by contrast (Figure 1(b)). The system determines affected parts of the model and the target lighting conditions. Various lighting parameters like positions, directions and spot angles of light sources are optimized to minimize the per-vertex differences between the actual and target lighting (Section 4). When the optimized lighting is presented, the user can rotate the model, specify more constraints similarly and continue the design procedure. After satisfactory lighting is achieved, the system outputs all the relevant lighting parameters that can be plugged into any other program using a similar lighting model to reproduce the lighting. We show how our tool can be used in conjunction with existing applications/tools with lighting capabilities by designing lighting for *OpenGL*-like systems and ray tracing.

2 Related Work

Inverse lighting is a sought-after area of research. In this section, we shortly summarize work done in this area. Patow *et al.* [2003] provide a more comprehensive survey.

Inverse lighting has been tackled in various contexts ranging from interior design to cinematography. Barzel [1997] proposes a very general lighting model with a lot of degrees of freedom in the context of cinematographic lighting. Radiooptimization [1993] and Painting-with-Light [1993] use radiosity and target interior design applications. We target the large number of applications which either use lighting when lighting design and setup is not their main purpose, or applications like ray tracing which are computationally expensive and hence make interactively designing lights difficult.

Various user models have been attempted in the context of lighting design. Kristensen *et al.* [2005] concentrate on real-time rendering so that the user may interactively place lights and examine their visual effects. Automatic techniques to infer lighting are based on analysis of the scene geometry [Shacked and Lischinski 2001]. In contrast, many approaches allow the user to specify desired lighting effects in some intuitive manner and design lighting that produces them. Our tool belongs to this category. A popular approach is to directly "splatter" the model with desired colors [Schoeneman et al. 1993; Poulin and Fournier 1995]. Poulin *et al.* [1992; 1997] allow users to hint shadows and highlights by outlining "footprints" of light sources on the model. If all the specified highlights are assumed to come from specular lighting effects, then such footprints or contours make the light placement problem easier. However determining light positions and intensities is more difficult for diffuse lighting effects. Many applications target specific users like animators and professional artists [Marks et al. 1997; Pellacini et al. 2005; Pellacini et al. 2007] by offering domain-specific interface metaphors. We target naive users whose main aim is not to design lighting but to use it in a bigger application.

Depending on the application, choice of lighting and user model, the inverse lighting problem can be formulated in several ways. Some formulations are tightly coupled with the lighting model (inferring patch radiosities [Kawai et al. 1993]) while others are more generalized. Gumhold [2002] formulates it as an entropy minimization. Costa *et al.* [1999] propose a comprehensive general technique based on optimizing complex cost functions constructed from hints about desired rendering using global illumination. Two radically different formulations are presented in DesignGalleries [1997] where the user selects between various configurations through an interface, and Light Collages [2004] which formulate the problem as an efficient greedy problem that infers possibly globally inconsistent lighting. Shacked *et al.* [2001] take an image-based approach in which the object is lit and rendered into a portion of the frame buffer, read back and evaluated. Though the complexity depends only on the size of this buffer, determining its size so that no features are lost is a difficult problem. An object-based approach considers vertex intensities as a representation of a candidate light field. Since this method works on 3D object data, various geometric characteristics like edges, etc. can be precomputed for efficiency during optimization. A disadvantage of this method is that this operation is now of scene complexity and hence is slower for larger models. We formulate this operation so that it can be efficiently executed on modern graphics hardware. Our formulation is similar to that of Lighting-with-Paint [2007], in that we minimize the per-primitive difference between actual and target lighting. Our work is different from theirs in three aspects: (1) we optimize over vertices, allowing the user to easily and frequently change view points (although their method can do this, they target an application where view point changes, if any, are infrequent. (2) our framework attempts to retain existing lighting conditions and hence is more amenable to designing lighting incrementally (3) whereas they rely on the user to choose between adding a new light and retaining the existing ones, we automate this process. The last feature is significant because the freedom of adding a new light automatically within the optimization framework significantly complicates solving the optimization problem.

The rest of the paper is organized as follows: we explain our sketch-based interface and various data structures used to obtain the target lighting specifications in Section 3. Section 4 discusses our formulation and solution of the inverse lighting problem. Section 5 discusses implementation details. We discuss results in Section 6 and conclude in Section 7.

3 Sketching Interface

A wide range of user interfaces can be used in the context of inverse lighting systems; Painting-with-light [1993] and the work done by Poulin *et al.* [1997] are some good examples. Sketching strokes is an intuitive way even for amateurs to specify lighting of a model in an abstract way. Sketching can not only be used to illustrate lighting in an abstract way, but also hint desired lighting. We call this the "crayon coloring interface".

Many inverse-rendering systems are based on a user interface in which the user directly "paints" desired colors onto visible parts

of the model. As the user cannot be expected to exactly paint the correct colors, the painted colors are regarded as "hints". Such inaccurate hints may be interpreted incorrectly as a target lighting field, often misguiding the underlying optimization. To circumvent this potential problem and to relieve the user from selecting the most appropriate colors, our method works towards a more high-level goal of brightening and darkening. Orange and blue strokes can be used to specify bright and dark regions respectively. The user can cross-hatch or even directly paint, as only the vertices the strokes approximately cover are of importance. It is not necessary to stay within the silhouettes. Strokes can be retraced to emphasize greater brightness or darkness.

We refer to the vertices that the user sketches upon as "hit vertices". In order to identify these vertices, we enclose the model in a volumetric grid C_{ray} and use it for efficient ray-casting. We use the fast voxel traversal algorithm proposed by Amanatides *et al.* [1987] for this purpose. In our current implementation, we use a 256^3 volume to achieve a reasonable trade-off between speed and memory requirements. Although using the depth buffer to identify triangles may be faster, our volumetric grid is useful for other purposes as well, as explained in Section 4.3.

4 Lighting Design

Once the user has finished sketching highlighted and darkened regions, a target lighting field is constructed from these hints. A non-linear optimization is formulated that attempts to design lighting to achieve this target lighting.

4.1 Quantifying the target lighting

The user's hints merely indicate which regions should be made brighter or darker. We now quantify this input by determining a target light field,i.e. we assign a "desired final" intensity to every vertex of the model that reflects the user's input. We start from the current vertex intensities, and then increment or decrement them according to the input. Since triangles other than those sketched upon may also be affected by the desired lighting parameters, we need a target light field that gradually changes over the model.

A given light can affect vertices that are geodesically close quite differently, depending on their normals, which can vary significantly due to curvature inspite of the small geodesic distance between then. Thus, a good target light field should mimic this by enhancing every vertex according to its geometric context, i.e. the local gradient around it. We employ a scoring method to approximate the surface gradient around a vertex by pre-computing a score k_v for every vertex v in the mesh. We start with a default score of 0.1 for every vertex. For every edge e in the mesh, we increase the score of its end vertices proportional to the gradient around it from the (at most 2) triangles that share it. Thus, k_v is an indicator of the change in surface geometry around vertex v. We make increments and decrements of vertex intensities linear functions of k_v to obtain their target intensities.

4.2 Optimization Formulation and Solution

We now explain how various lighting parameters are obtained, given the target field generated as explained in the previous section. The following notation is used in this section:

V	:	Set of all vertices of the model
V_h	:	Set of hit vertices
V_{other}	:	Vertices in $V \setminus V_h$ to evaluate light field
L_i	:	i^{th} light
$I(L_i)$	:	Intensity of i^{th} light
X	:	Set of all unknowns (lighting parameters)
c_i	:	Intensity of vertex i in candidate field
t_i	:	Intensity of vertex i in target field

We construct a function that is to be minimized to achieve two main objectives: (1) the function should capture the difference between the target lighting field and a candidate lighting field in a particular iteration for all hit vertices, and (2) for incremental lighting design it is desired that the current setting of lights minimizes changes in those set previously. Our minimizing function is given by

$$f(X) = w_1 * f_{change} + w_2 * f_{retain} + w_3 * f_{barrier} \qquad (1)$$

X is the set of various lighting parameters. We use $[w_1, w_2, w_3] = [0.7, 0.1, 0.2]$ for all results in this paper. We consider seven lighting parameters as degrees of freedom (DOFs): position (in polar coordinates) (θ, ϕ), direction (in polar coordinates)$(\theta_{dir}, \phi_{dir})$, diffuse (k_d) and specular (k_s) intensities and spot angle ψ.

$$f_{change} = \frac{\sum_{v_i \in V_h} |c_i - t_i|}{|V_h|}$$

$$f_{retain} = \frac{\sum_{v_i \in V_{other}} |c_i - t_i|}{|V_{other}|}$$

$$f_{barrier} = \sum_{lights L_i} max(0, -I(L_i)) + max(0, I(L_i) - 1)$$

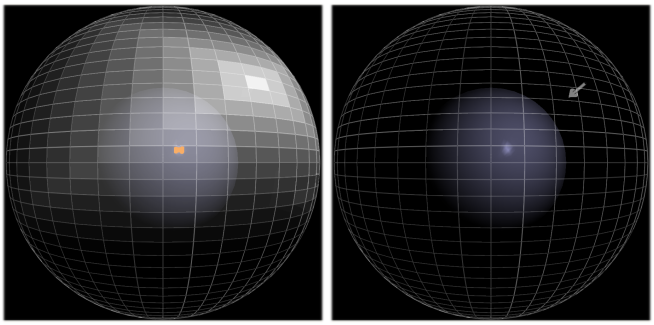
f_{change} minimizes the sum of differences between the candidate and target intensities of vertices in V_h. This term is similar to that in Lighting-with-paint [2007]. f_{retain} minimizes the change of intensities of vertices in V_{other}. $f_{barrier}$ prevents light intensities from falling below 0 and going above 1.

The problem is to minimize $f(X)$ parameterized by the various DOF's mentioned above. The generality of such a function makes its solution difficult in many aspects. Firstly, as $f(X)$ is discontinuous (because of $f_{barrier}$ and spot angle cutoffs), conventional optimization methods, if used directly, may not converge properly. Secondly, as global minimization methods like genetic methods may be infeasible for interactive solutions, locally minimizing methods must be used. A good initial guess is critical for such methods to work correctly. Thirdly, when darkening or shadowing is desired, none of the DOFs have a continuous and direct relationship with shadowing effects, especially those concerning self-shadowing. The fourth and most critical difficulty is that not only are the various DOFs unknown for each light, but also the number of lights (thus the number of variables itself is unknown). Devising an optimizing function that automatically adds/removes variables is very difficult. Lastly, the goals of interactivity and good quality of solution in general may appear contradictory in practice.

We address the above problems in two ways. First, we devise a novel method to initialize lighting parameters of all newly added lights that works well in practice towards converging to the correct minima. We use this method to also add or delete lights from the

system, thereby automating this choice. We take into account self-shadowing effects while initializing lighting configurations. Secondly, we delegate evaluation of $f(X)$ to the GPU to facilitate faster computations.

4.3 Lighting Setup



We surround the model with a *sphere of lights* S_{lights}. We assume that any new lights lie on this sphere; the position of every light source L can thus be described in polar coordinates (θ_L, ϕ_L). This sphere is centered at the center of the model with a radius equal to the body diagonal of its enclosing volume C_{ray} (Section 3). We discretize the sphere into quadrilateral bins. Let $v \in V_h$ be a vertex with normal $\vec{n}$. Let $\vec{q}$ be a vector from v to the center of a quadrilateral Q on S_{lights}. If v is to be highlighted, then the score of every quadrilateral Q is increased by $w * (\vec{n} \cdot \vec{q})$, while if v is to be darkened, it is increased by $(1 - w) * (1 - \vec{n} \cdot \vec{q})$ (in the adjoining illustrating figure, whiter quads have greater scores). The weight $w = (0, 1)$ is a visibility term that encodes whether Q is visible from v along $\vec{n}$, and accounts for shadowing. We tried two different implementations to get w: ray casting using C_{ray} and hardware-based occlusion queries. Surprisingly we found that ray casting performed comparably to the occlusion queries. We believe this is because there are a large number of small occlusion queries (one per (v, Q) pair). Occlusion queries are inefficient in such scenarios because they cause blocking calls from the driver between the CPU and the GPU.

When all the vertices in V_h are processed this way, S_{lights} encodes a per-light probability of it being added to the system. In addition, every bin encodes the complete initial configuration of the light: initial position (on the sphere), direction (towards the center), default intensity (0.5) and spot angle (10°). Our experiments have shown that discretizing C_{ray} into 400 quadrilaterals gives satisfactory results; making it finer increases the potential number of light sources that the program adds in response to a series of inputs.

4.4 Solving the optimization

In order to achieve automatic light addition/deletion and to keep the problem tractable, we perform the optimization in several stages. At any stage, if $f(X)$ falls below a certain threshold $T_{f(X)}$ (0.2 for all the results shown in the paper), the system declares success. In the first stage, only the positions and intensities of all existing lights are used to minimize $f(X)$. If this succeeds (the value of $f(X)$ falls below $T_{f(X)}$), the system declares success.

If this is not the case, the system iteratively checks S_{lights} for "hot spots", i.e bins that have a clearly large probability over others. To do this, it monitors the maximum, mean and standard deviations of probabilities in S_{lights}. If the ratio of standard and maximum deviations rises above a certain threshold K(40%), it concludes that there are no more clear choices of new lights. If it finds hot-spots, it greedily selects the one with maximum probability and adds the corresponding light to the system, with all its DOF's enabled. It then attenuates the probabilities in S_{lights} by a function that increases as one moves away from the selected light on S_{lights}. The intuition is to prevent selection of a neighboring light in the very next iteration. Then it solves the minimization problem using previous lights (position modification disabled) and the newly added light (with all its DOFs enabled). If $f(X)$ falls below $T_{f(X)}$, it declares success. If $f(X)$ increases in value because of addition of this light, it deletes it from the system. If $f(X)$ has decreased but is above the threshold, it disables all the DOFs of the newly added light except its intensity for the next iteration. It then looks for another hot-spot in S_{lights} and continues this until a maximum of 5 new lights are added, $f(X)$ goes below $T_{f(X)}$ or the ratio of standard and maximum deviations in S_{lights} exceeds K.

We use the conjugate gradient method to solve the actual optimization at any stage. This method requires calculation of the gradient of $f(X)$ for which we use partial central differences. $f_{barrier}$ produces a discontinuity for all intensities $I < 0$ or $I > 1$. Since these cases are easily detectable (an unusually large derivative component in the forward/backward difference but a normal component in the other), we set it to the lesser of the two to "guide" the optimization away from the discontinuity. If the gradient using forward difference is positive and backward difference is negative (implying that the function is minimum at this point along the particular variable domain), we set the gradient to 0 to preclude it from further optimization steps. Thus, we use the conventional conjugate gradient minimization method in various stages with greedy initialization and light retention.

5 Implementation and System Features

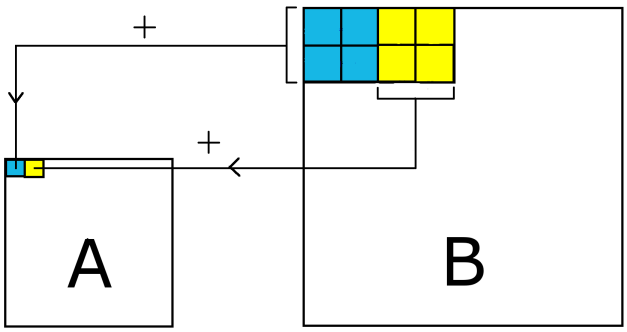
In this section we include implementation details that make our system interactive. This tool has been implemented with OpenGL and GLSL for graphics on a desktop machine with a 3.0GHz Pentium 4 processor with 1GB RAM and an NVIDIA GeForce FX-6600 on the Windows XP platform. Tablet input and output is provided by an external Wacom Cintiq PL-550 tablet device.

5.1 Calculating the minimization function

The most computationally expensive operation in each iteration of the optimization is calculating $f(X)$ at a given X, whose bottlenecks in turn are f_{change} and f_{retain}. We implement their evaluation fully on the GPU. This technique is inspired by the work by Windsheimer *et al.* [2004], who implement a visual difference metric in hardware.

It can be observed that both f_{change} and f_{retain} are linear functions over (intensities of) sets of vertices V_h and V_{other} respectively. We map these vertex sets to texture memory and evaluate f_{change} and f_{retain} as texture operations.

Consider a set V of vertices over which a linear function f has to be calculated. We first arrange all vertices in V in a vertex quad. If $|V| = n$ then this quad is $\sqrt{n} \times \sqrt{n}$ pixels in dimensions. We pass the position of each vertex in this quad as its texture coordinate. In a vertex shader, we perform per-vertex lighting computations and set the target location of the vertex to its texture coordinate. We render this quad as a texture T_1 and use it as input to render the vertices in V again, this time with the target vertex intensities and no lighting. In this pass, we compute in a fragment shader the difference $|c_i - t_i|$ and store it in the target render texture DT, where c_i is the current pixel intensity and t_i is the value of this pixel in T_1. We thus obtain a "difference texture" where every texel represents the difference between the target and candidate light field at a vertex. For calculating f_{change}, $V = V_h$ and for f_{retain}, $V = V_{other}$.



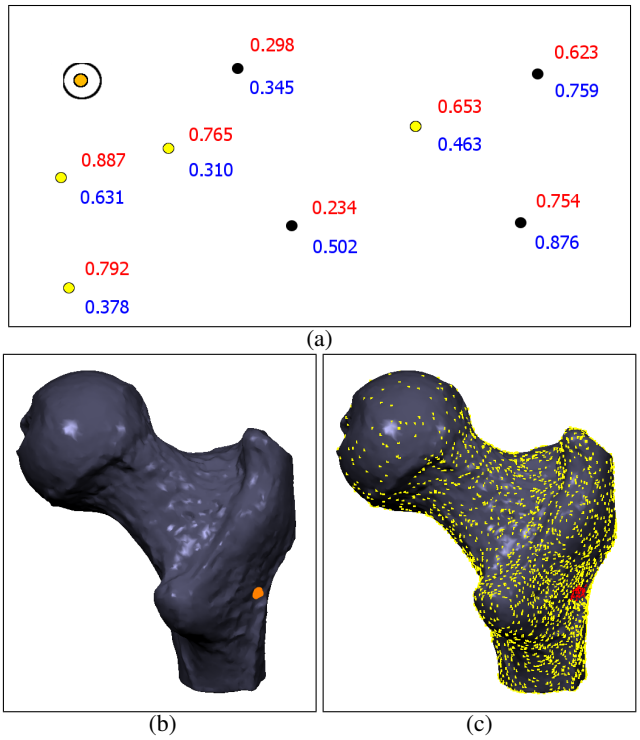
To calculate f_{change} and f_{retain} we have to find the sum of all texels in its difference texture. We use a multi-pass approach to achieve this. We create two float buffers and render DT in one of them. We use them alternately as the rendering context and input texture in various passes as follows: we start from $s_0 = \sqrt{n}$ and at the i^{th} iteration, we render a quad of size $s_i = \frac{s_0}{2^{i+1}}$. The quad in the $(i-1)^{th}$ iteration of size s_{i-1} is used as an input texture T in the i^{th} iteration. In a fragment shader, every pixel $P(x, y)$ reads the texture locations $T(2x, 2y), T(2x+1, 2y), T(2x, 2y+1), T(2x+1, 2y+1)$ and stores their sum as the color of $P(x, y)$. As we use float buffers, fragment colors are not clamped. In this way, the size s_i goes on decreasing to 1, when we simply read out a single pixel value from one of the buffers. This operation takes $\log_4 n$ passes (on the GPU using a pixel shader) and involves reading only one pixel from the GPU into the CPU.

For this algorithm to work, the initial difference texture DT must be a square texture of power-of-two dimensions. If $|V_h|$ or $|V_{other}|$ do not satisfy this condition, DT is padded with 0's.

5.2 Sampling vertices

The set of vertices $V_h \cup V_{other}$ used to calculate $f(X)$ can simply be all the vertices in the mesh. Considering all vertices in the mesh causes two problems. First, calculation of f_{retain} and hence $f(X)$ becomes expensive. Secondly, if f_{retain} is a function of a large number of vertices, then it causes the optimization to always converge to the trivial local minimum (the previous lighting configuration). Thus some sampling scheme must be devised to select a subset of $V \setminus V_h$ as V_{other}. It is desirable that this sampling be fairly representative of the whole mesh, so that minimizing the difference between target and candidate lighting intensities over V_{other} retains the overall look of parts of the model not in V_h.

A sampling of vertices such that those near the hit triangles have a greater probability of being selected than the distant ones is desirable, so that only these samples are included in V_{other} instead of all the vertices. We achieve this by a Sampling-By-Random-Number (SRN) method as follows (please refer to Figure (a) above): we generate a random number r_i for every vertex v_i in the mesh (red numbers). We then compute a score for each vertex (blue numbers) that varies directly with its distance from the hit region (orange dot) (in practice we consider the centroid of a contiguous hit region for this purpose). If this score is less than r_i, the vertex is selected (yellow), else it is rejected (black). Figures (b-c) illustrate this sampling on an actual model. In Figure (b) the input strokes are drawn to focus on a region. The corresponding hit vertices are shown in red and the sampled vertices are shown in yellow in (c). Notice how sampling is sparser in regions distant from the marked region in (b). Similar random sampling methods have been used in the context of non-photorealistic rendering [Nguyen et al. 2003], volume rendering [Yuan and Chen 2004] and point-based rendering [Qu et al. 2006].



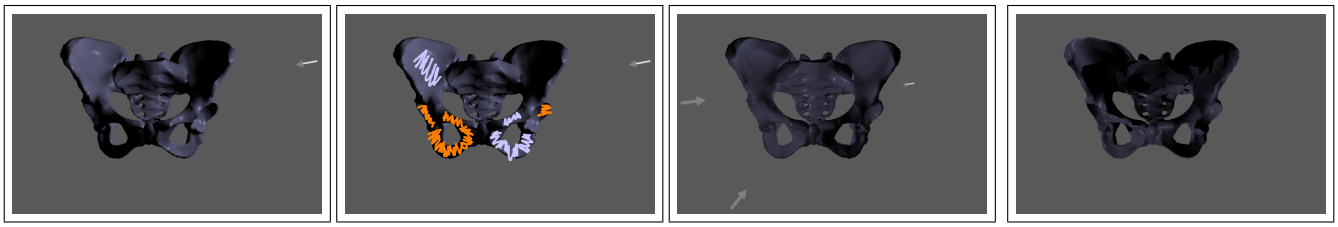
5.3 Added Benefits and Features

Our implementation choices lead to some minor but useful features that make user experience more convenient. The user can enable/disable individual DOFs of lights to facilitate better results or to exclude features that he/she does not need.

If the result looks to conform with the input in quality but not in quantity (e.g. looks brighter but not bright enough), the user can choose to repeat the previous input without sketching anything. It can be observed that S_{lights} not only stores configurations of lights to be added, but also those that are already added. Thus, if the user wishes to minimize the number of lights while relaxing the constraint on their intensity ranges, lights can be trivially clustered together using S_{lights}. This can be useful if the external application supports only a fixed maximum number of lights.

6 Results

Figures 1, 2, 3, 4 and 5 show results on some models. All of these results were produced by a computer science professional (not one of the authors) who does not primarily work in computer graphics. In general several inputs could be required progressively to arrive at the desired result. Each figure shows lights in the form of arrows; the base of the arrow indicates its position and the direction shows its direction. Figure 1(a) shows an example of the medical model of a hip with 40,000 triangles, lit with default lighting. Figure 1(b) shows the user input. The aim of the user input was to shift focus to the cavity in the model while receding the remaining model into shadow. Figure 1(c) shows the result produced by our system in 5 seconds, in which a light is added to create the highlight and the existing one is shifted to darken the part marked in the input with the blue pen. Figure 1(d) shows the final result produced by an anti-aliased ray tracer. It can be seen how the light placement has included self-shadowing aspects.



(a) (b) (c) (d)

Figure 2: *Results: Pelvis. (a) a pelvis model with 50,000 triangles with default lighting. (b) A slightly contrived input is given to switch the contrast between oppositely lit lower cavities. (c) the system realizes this input by adding two lights and moving the existing one. Notice how one lower cavity is highlighted while the other remains dark. (d) the final ray-traced output.*

Figure 2(a) shows a pelvis model[1] with a slightly contrived user input. The aim is to switch the opposing lighting conditions on the two lower cavities of the pelvis. Figure 2(b) shows how the system accordingly complies with this input. Although the rendering of the upper cavities seems unsatisfactory in the *OpenGL* rendering of this figure, the ray-traced result in (c) shows compliance with the input. In particular it can be seen that the lighting has produced a dark region in the upper left region through shadowing.

Figure 3 shows results on the ball joint model having 35,000 triangles. The goal of the input in Figure 3(a) is to highlight the ball region. The result of this input is shown in Figure 3(b-c). This result took 4.16 seconds.

Figure 4[2] shows how lights can be obtained by progressively transforming a model and sketching on it. Figure 4(a) shows the filigree model having 177,000 triangles with default lighting. The goal is to contrast facets facing in opposite directions to arrive at a better lighting that enhances the structure of the model. The object is rotated and sketched upon in Figure 4(b) to obtain the lighting of Figure 4(c-d). Again, the object is rotated and some more hints are sketched (Figure 4(e)) to increase the contrast, as shown in Figure 4(f-h).

Figure 5 exemplifies an interesting application of our system. Figure 5(a) shows a pre-existing image of an NPR rendering of the same model that shows lit and darkened regions[3]. We estimated the view point by trial and error and then attempted to reproduce the lighting effects in this image by progressively sketching highlights and dark spots corresponding to the dark and light regions of the image (Figure 5(b) shows one such input). Figures 5(c-d) show the result in which most of the effects in Figure 5(b) have been reproduced. Thus, our system can be used to *reverse-engineer* lighting of a model, given an example image of its rendering.

7 Discussion and Future Work

Crayon lighting is a tool that performs inverse lighting given a sketchy input in which the user sketches bright and dark regions directly on the model. We envision this tool being used by modelers or researchers in computer graphics and visualization as a simple tool that outputs the necessary lighting parameters for good model visualization. With some modification it can even be used in an educational setting to teach the primary physics of light in introductory graphics courses. Our system is easy to use, works at interactive rates and restricts user input to drawing highlights and shadows.

[1] model obtained courtesy of VCG-ISTI by the AIM@SHAPE Shape repository

[2] model obtained courtesy of SensAble Technologies inc. by the AIM@SHAPE Shape repository

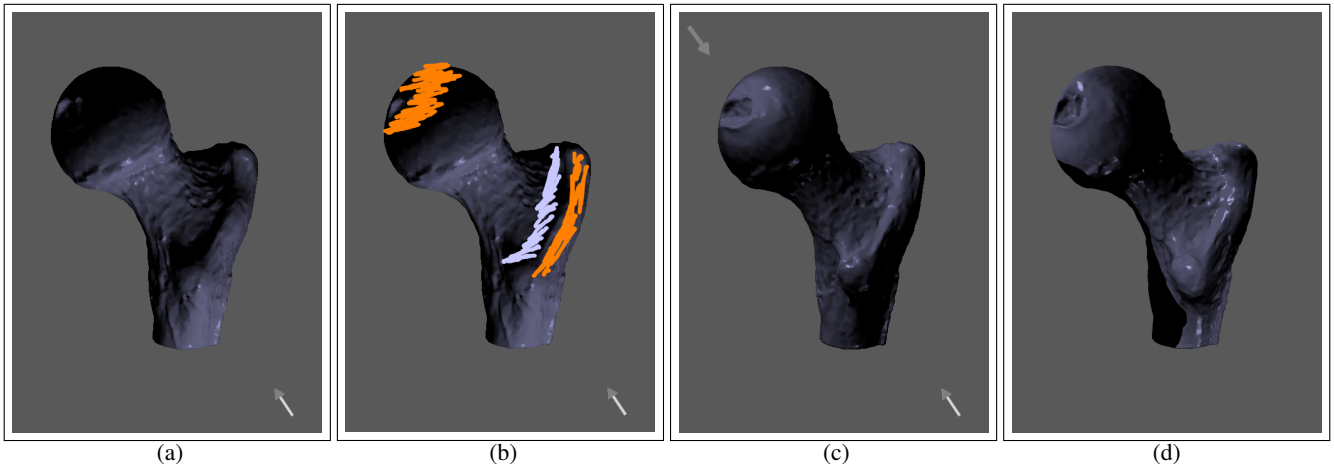
[3] both model and image were obtained from the Suggestive Contour Gallery (http://www.cs.princeton.edu/gfx/proj/sugcon/models/)

While using this tool, we encountered some issues that an implementor or user should be aware of:

Error detection mechanism: There is currently no built-in error detection mechanism–if the user chooses to specify an invalid input (e.g. highlighted and darkened regions very close to each other), the system will still try to solve for the lighting and may come up with an unsatisfactory or degenerate result. Besides the user refraining from specifying such inputs, an undoing mechanism can be easily incorporated into the system to revert to the previous stable lighting.

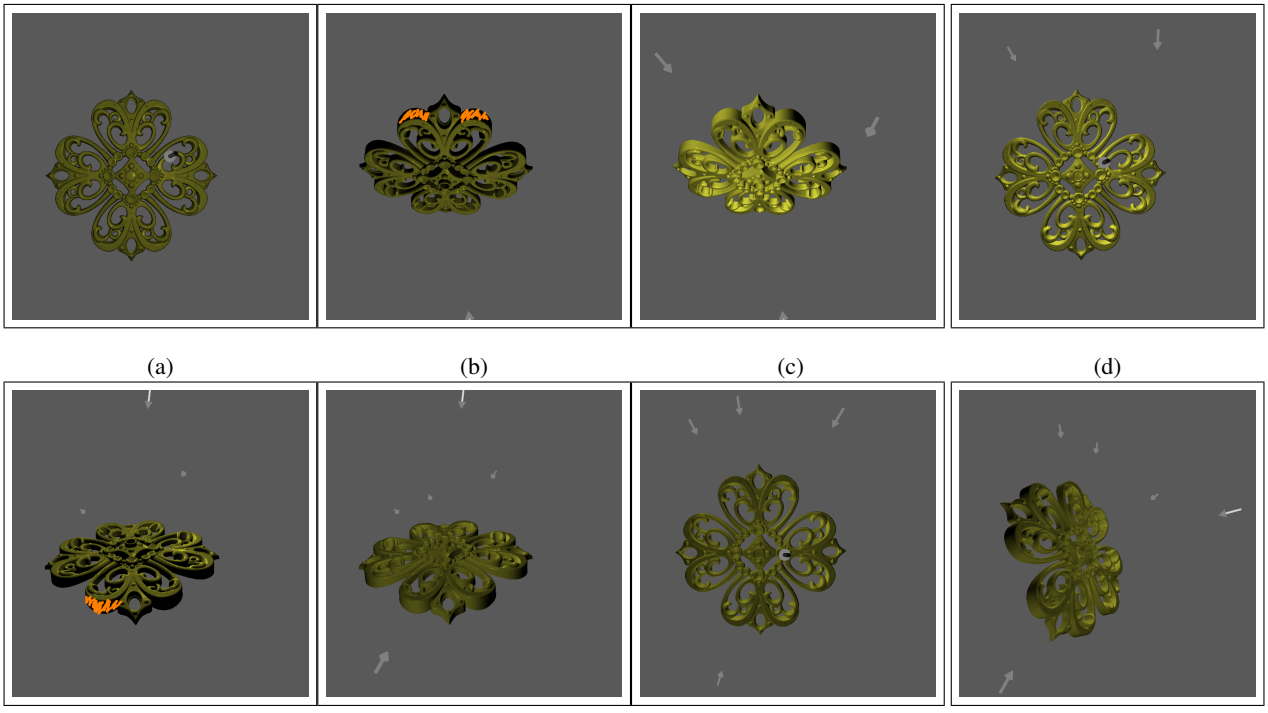
Choice of parameters: Discretization of C_{ray} simply affects the speed vs. memory ratio of the ray casting (Section 4.1) and not the quality of the final result. If S_{lights} (Section 4.3) is finely discretized, the likelihood of a new light source being added for an input is greater. Hence, if the desired number of light sources is limited, a coarser discretization may be more suitable. Optimization threshold $T_{f(X)}$ and K were obtained empirically after testing on various 3D models for various inputs. Changing K results in more or less lights to be added. However we discovered that after some iterations, the program starts to "thrash", i.e. it adds a new light and upon realizing that the value of the objective function actually increased , it deletes the light. Thus increasing the number of added lights does not necessarily result in a better quality output.

The work described in this paper is mainly for the purposes of visualizing a model by setting the lighting in an intuitive manner. However, the larger problem of inverse lighting has a variety of applications. Architectural lighting design and cinematic lighting use complex and realistic illumination models and diverse sources of lights, like colored lights, spot lights, point and linear lights, etc. A second requirement is the ability to have fine control over the lighting conditions (i.e, in cinematic lighting, directors need precise control over how and where shadows fall, to make the set look as appropriate to the mood of the scene as possible). Accordingly, a method to solve the interactive inverse lighting problem that allows usage of complex illumination models, and offers fine control over the produced lighting conditions can be useful in many ways to a large number of applications. We are currently investigating solving the interactive inverse lighting for more common, real-world but complex applications like architectural and cinematic lighting design. There has been recent work in the area of interactive global illumination [Ng et al. 2003; Hašan et al. 2007], some of which is even targeted towards real-time lighting design [Kristensen et al. 2005]. Many of these alleviate the costs of performing global illumination by using approximation methods like wavelets or smart matrix approximations. Our interest lies in investigating whether such methods that make rendering efficient can also be leveraged to design lighting based on intuitive sketch-based input.



(a) (b) (c) (d)

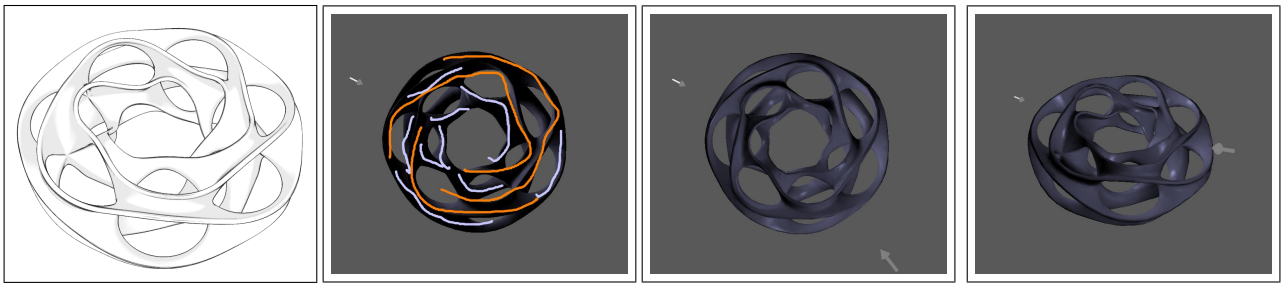
Figure 3: *Results: Ball joint. (a) the ball joint model with 35,000 triangles with default lighting. (b) user input. (c) output produced by our tool showing the resulting lighting focusing on the ball. (d) ray-traced rendering using the same lighting conditions as (b).*



(a) (b) (c) (d)

(e) (f) (g) (h)

Figure 4: *Progressive inverse lighting. Figures show a filigree model having 177,000 triangles. The aim is to contrast opposite faces with light and darkness. (a) the model with default lighting. (b) the model is rotated and some input is given. (c) result of input from (b). (d) the lighting seen from the original view point. Some faces are better lit than in the default lighting in (a). (e) model is rotated again and more strokes are sketched. (f) result of input from (e). (g) the lighting seen from the original view point. (h) the lighting from a new view point to show the light positions better. All images have been rendering using conventional OpenGL rendering.*



(a) (b) (c) (d)

Figure 5: *Results: Heptoroid. (a) pre-existing NPR rendering of the heptoroid showing areas of highlights and shadows. (b) we progressively sketched highlights and dark regions on the model to replicate the darker and lighter regions in the image. (c-d) the result showing most of the light effects reproduced. In this way our system can be used to reverse engineer lighting, given a model and a pre-rendered image.*

Acknowledgements

Support for this work includes University of Minnesota Digital Technological Center Seed Grant, a Microsoft Research Gift, NSF CCF-0238486 (CAREER) and NSF DMS-0528492 (MSPA-MCS). We thank Poonam Shanbhag and Neha Shanbhag for experimenting with our program and generating some of the results of this paper, and Nathan Gossett for proof-reading the paper. We thank the anonymous reviewers for their critique and useful comments.

References

AMANATIDES, J., AND WOO, A. 1987. A fast voxel traversal algorithm for ray tracing. In *Proc. Eurographics*, 3–10.

BARZEL, R. 1997. Lighting controls for computer cinematography. *Journal of Graphics Tools 2*, 1, 1–20.

COSTA, A. C., SOUSA, A. A., AND FERREIRA, F. N. 1999. Lighting design: A goal based approach using optimisation. In *Rendering Techniques*, 317–328.

GUMHOLD, S. 2002. Maximum entropy light source placement. In *Proc. IEEE Visualization*, 275–282.

HAŠAN, M., PELLACINI, F., AND BALA, K. 2007. Matrix row-column sampling for the many-light problem. *ACM Trans. Graph. 26*, 3, 26.

KAWAI, J. K., PAINTER, J. S., AND COHEN, M. F. 1993. Radioptimization: goal based rendering. *Proc. SIGGRAPH*, 147–154.

KRISTENSEN, A., AKENINE-MOLLER, T., AND JENSEN, H. 2005. Precomputed local radiance transfer for real-time lighting design. In *Proc. SIGGRAPH*, 1208–1215.

LEE, C. H., HAO, X., AND VARSHNEY, A. 2004. Light collages: Lighting design for effective visualization. In *Proc. IEEE Visualization*, 281–288.

MARKS, J., ANDALMAN, B., BEARDSLEY, P., FREEMAN, W., GIBSON, S., HODGINS, J., KANG, T., MIRTICH, B., PFISTER, H., RUML, W., RYALL, K., SEIMS, J., AND SHIEBER, S. 1997. Design galleries: A general approach to setting parameters for computer graphics and animation. *Proc. SIGGRAPH*, 389–400.

NG, R., RAMAMOORTHI, R., AND HANRAHAN, P. 2003. All-frequency shadows using non-linear wavelet lighting approximation. In *Proc. SIGGRAPH*, 376–381.

NGUYEN, M. X., XU, H., YUAN, X., AND CHEN, B. 2003. Inspire: An interactive image assisted non-photorealistic rendering system. In *Proc. Pacific Graphics*, 472–477.

PATOW, G., AND PUEYO, X. 2003. A survey of inverse rendering problems. *Computer Graphics Forum 22*, 4, 663–688.

PELLACINI, F., VIDIMČE, K., LEFOHN, A., MOHR, A., LEONE, M., AND WARREN, J. 2005. Lpics: a hybrid hardware-accelerated relighting engine for computer cinematography. In *Proc. SIGGRAPH*, 464–470.

PELLACINI, F., BATTAGLIA, F., MORLEY, R. K., AND FINKELSTEIN, A. 2007. Lighting with paint. *ACM Trans. Graph. 26*, 2, 9.

POULIN, P., AND FOURNIER, A. 1992. Lights from highlights and shadows. In *Proc. SI3D*, 31–38.

POULIN, P., AND FOURNIER, A. 1995. Painting surface characteristics. In *Proc. EGSR*, 160–169.

POULIN, P., RATIB, K., AND JACQUES, M. 1997. Sketching shadows and highlights to position lights. In *Proc. CGI*, 56.

QU, L., YUAN, X., NGUYEN, M. X., MEYER, G., AND CHEN, B. 2006. Perceptually guided texture mapping on points. In *Proc. IEEE/Eurographics Sym. Point-based Graphics*, 95–102.

SCHOENEMAN, C., DORSEY, J., SMITS, B., ARVO, J., AND GREENBURG, D. 1993. Painting with light. In *Proc. SIGGRAPH*, 143–146.

SHACKED, R., AND LISCHINSKI, D. 2001. Automatic lighting design using a perceptual quality metric. *Computer Graphics Forum 20*, 3.

WINDSHEIMER, J. E., AND MEYER, G. W. 2004. Implementation of a visual difference metric using commodity graphics hardware. In *Human Vision and Electronic Imaging IX, Proceedings of the SPIE*, vol. 5292, 150–161.

YUAN, X., AND CHEN, B. 2004. Illustrating surfaces in volume. In *Proc. IEEE/EG VisSym*, 9–16.