UNIVERSITY OF MINNESOTA AT TWIN CITIES
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

This is to certify that I have examined this copy of a doctoral dissertation by

Amit Prakash Shesh

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Name of Faculty Advisor

Signature of Faculty Advisor

Date

GRADUATE SCHOOL

USING HAND-DRAWN SKETCHES FOR MODELING AND RENDERING IN COMPUTER-BASED DESIGN AND ART

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

AMIT PRAKASH SHESH

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

ADVISOR: Baoquan Chen

AUGUST 2008

© Copyright by Amit Prakash Shesh, 2008

Acknowledgements

I would like to thank my advisor Dr. Baoquan Chen for his faith and confidence in me, and for his continuing guidance in my research and career. It is because of his enthusiasm and support that I have been able to convert my raw fascination of computer graphics into the primary focus of my career that it is today. I am also thankful to Dr. Gary Meyer and Dr. Ravi Janardan for their guidance inside and outside the classroom and for their advice in professional matters and my research. I also thank Dr. Andrzej Piotrowski for his insights in architectural design that contributed to the *SMARTPAPER* project.

My doctoral study would not have been as fruitful and enjoyable without the other students in the graphics group at the University of Minnesota. I thank Minh Nguyen, Hui Xu, Xiaoru Yuan, Nathan Gossett, Jie Chen, Clement Shimizu, Lijun Qu, Timothy Urness, Sunghee Kim, Jared Windsheimer, Brian Ries and Jon Konieczny for the many insightful discussions that have made me a better student, teacher and researcher.

Finally I thank the many people who have nothing to do with my particular research, but nevertheless have inspired me through their casual conversations and actions in the classroom, at conferences, etc. I regard such people and their encounters with me as continuous sources of inspiration for my research. On a personal note, I thank my wife Poonam, my parents Prakash and Alka Shesh and my brother Nitin for their critical support.

Support for various projects of my research includes a University of Minnesota Digital Technology Center Seed Grant 2003, a Microsoft Gift, and NSF CAREER ACI-0238486 and NSF DMS-0528492 (MSPA-MCS). The content of my dissertation does not necessarily reflect the position or the policy of these organizations, and no official endorsement should be inferred. Research space and administrative support has been provided by the University of Minnesota Digital Technology Center.

Abstract

The overall theme of my research work is sketch-based computer graphics–using sketches to create and enhance 2D/3D geometric models for a variety of applications in modeling, design and art. This dissertation summarizes my research in four areas, with sketches as the common theme. In sketch-based modeling, I have worked on converting freely drawn 2D sketches of objects into 3D models automatically, or with the minimal use of sketched gestures. An extension of this work constructs navigable 3D models from a single 2D image by treating manual traces of objects in the image as perspective sketches.

Another novel application that this dissertation discusses is sketch-based inverse lighting, which infers lighting information that produces desired effects roughly sketched by a user on a scene.

My thesis culminates in rendering 2D and 3D wire-frame sketches that identifies the intricate interplay between sketch-based and non-photorealistic rendering. The goal of this research is to construct an efficient rendering pipeline for abstract, hand-drawn sketches that preserves tone, shape and also addresses the visibility problem for sketches without any surface information. In addition to sketch-based applications, this research also contributes to the area of efficient non-photorealistic rendering under dynamic conditions.

This dissertation identifies the challenges in each problem, then discusses the contributions of my work in detail and identifies the possibilities in each problem beyond the scope of this dissertation.

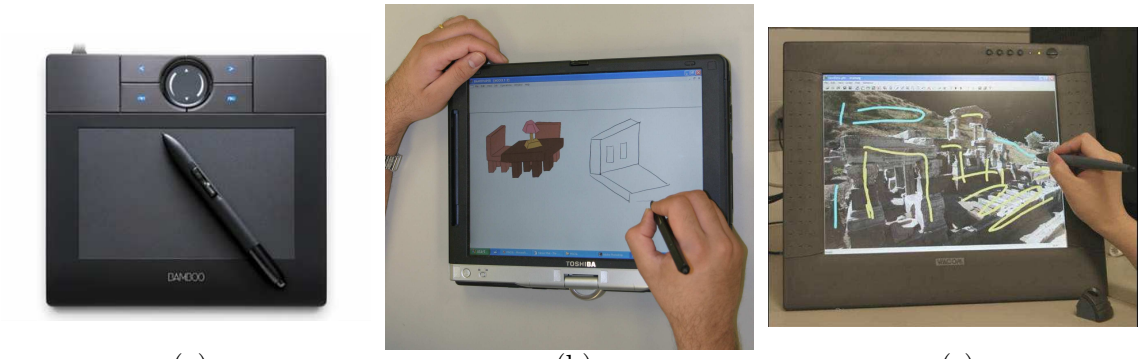
Table of Contents

List of Tables

List of Figures

Chapter 1

Main Theme of my Research



(a) (b) (c)

Figure 1.1: ***Commodity tablet devices.*** *(a) An external tablet (The Bamboo by Wacom[3]) that needs an external display device. (b) A tablet PC that has an integrated tablet as display. (c) An external tablet+display device (Wacom Cintiq[4]) that can be used as a display device.*

My broad research interests lie in computer graphics and its interdisciplinary applications. The greatest personal motivation behind my research work is to not only develop algorithms and techniques that solve important problems in computer graphics, engineering and artistic design, but also make these solutions accessible and useful to the scores of users who are not expert computer scientists.

Various I/O devices have gained popularity in computing technology in the past decades.

However, the counter-intuitiveness of these input technologies is obvious in many engineering, gaming and design applications (drawing with a mouse in a CAD application, driving a car in a computer game with a joystick, etc.). With the advent of commodity sketching devices like external tablet devices and tablet PC's, sketching with a pen on a planar surface has become an input method that is very intuitive and natural for human users.

Hand-drawn sketches have been very effective means of communication for people of all ages and professions. Sketches have great illustrative, expressive and artistic appeal that inspires many problems and solutions in computer science, especially in computer graphics. As tools of communication, sketches can be effectively used to create hitherto inconceivable applications, or even greatly improve the usability of existing applications. Their artistic forms themselves spur one of the most popular areas of study within computer graphics–non-photorealistic rendering. My research in general and this dissertation in particular explore applications of human-drawn sketches in design and art, and also the implicit relationship between sketch-based and non-photorealistic computer graphics.

Human-drawn digital sketches as input metaphors also have an important part to play in making applications and research in computer graphics and other fields accessible to the vast majority of computer users that are not expert computer scientists. Through my current and future research, I wish to make existing but difficult-to-use techniques accessible to the users that find the greatest use for them in practice. Designing new algorithms or modifying existing ones to accommodate the fuzzy nature of sketched input offers a unique challenge, and is an essential component of my research.

Chapter 2

Summary Of Research Contributions

This chapter summarizes various research projects related to the theme of my dissertation that I have worked on:

2.1 SMARTPAPER: 3D Modeling From Freehand 2D Sketches

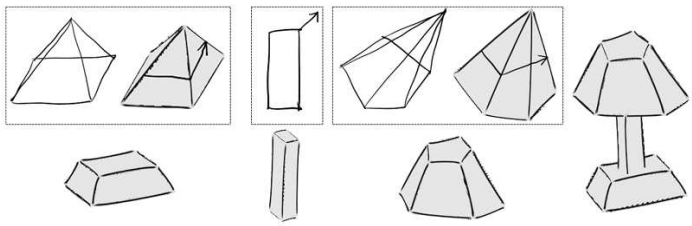


Figure 2.1: ***SMARTPAPER.*** *Figure shows how individual parts are sketched roughly, either directly or using gestures like arrows. The reconstructed parts are then assembled together to form a lamp.*

Engineers and architects typically use pencils and paper during early conceptual design, because no computer-based system offers enough flexibility for rough sketching. When the design becomes more concrete, a CAD model is constructed from scratch. The objective of this research is to take these rough 2D sketches as input and directly construct plausible

3D models from them. Results of this research are prototyped in a system called *SMARTPAPER*. *SMARTPAPER* creates plausible 3D models from orthographic sketches. It has been designed to work with actual sketches of objects so that the user can draw exactly as on paper using a pencil. Several gesture-based operations are also supported.

My research contributions in this work are as follows:

1. My main contribution is to make the entire process of reconstructing 3D models from a single sketch interactive, so that it can be used as an application on a Tablet PC. Several specific contributions listed below contribute to this.

2. Algorithms to return the set of cycles in a graph that correspond to faces of the object whose 2D sketch the graph represents. These algorithms leverage probable sketching order and use shortest-path algorithms from graph theory. They work in real-time for most practical sketches.

3. An initial guess for the optimization problem that greatly speeds up its rate of convergence, and in many cases, the quality of the solution.

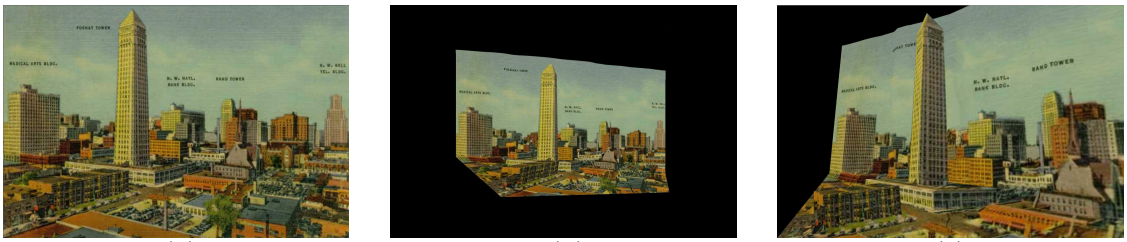
4. A simple gesture recognition system used to signal various operations and place constraints on the model using sketches.

5. A feedback loop using which a user may go back to the sketch from an incorrectly reconstructed 3D model to correct it, leading to a better 3D model. Such feedback is important to support reconstruction of shabby sketches and also aids the user in learning to use *SMARTPAPER*.

6. Several augmenting operations for completion, like cutting an object, combining several objects and annotating them with graffiti.

Chapter 3 discusses this research work in detail. This work is also summarized in [106] and [107].

2.2 Peek-in-the-Pic: Using Sketches To Navigate An Image



(a) (b) (c)

Figure 2.2: ***Peek-in-the-Pic.*** *(a) an input painting of Downtown Minneapolis. (b) a distant view of the constructed geometry. (c) another view of the constructed geometry.*

Restricting the sketches in *SMARTPAPER* to be orthographic is a strategic decision because orthographic projections preserve many geometric properties like parallelism and perpendicularity, which aid in the reconstruction process.

Peek-in-the-Pic represents research in reconstructing perspective sketches. The application of choice is a system that reconstructs a rough 3D model out of a single photograph by treating object trace-outs as perspective sketches.

My research contributions in this area are as follows:

1. My main contribution is to amalgamate many known and some new techniques to solve the overall problem of creating a navigable 3D model from a single image, with the aim of an easy user interface that produces a satisfactory result for images of a certain type.

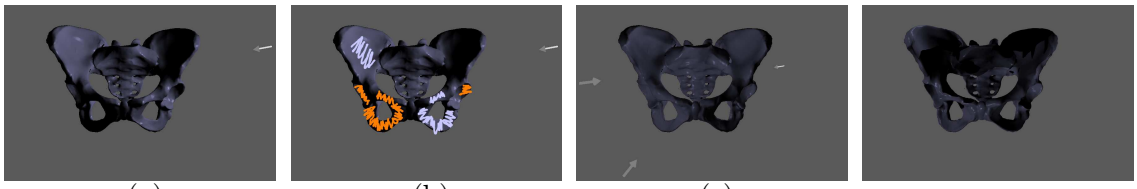
2. Support for reconstructing perspective sketches by performing automatic camera calibration to distort the sketches into their orthographic counterparts before reconstructing them.

3. An ability to progressively complete hidden parts of an object extracted from a photograph. This extends the face-recognition algorithm of *SMARTPAPER*.

4. An automatic method of filling holes in an image that uses conventional texture synthesis techniques, with an easy user-interface for unsatisfactory results.

Chapter 4 discusses this research work in detail. This work is also summarized in [108] and [111].

2.3 Crayon Lighting: Sketch-based Inverse Lighting



(a) (b) (c)

Figure 2.3: ***Crayon Lighting.*** *(a) a pelvis model with default lighting. (b) the sketched input. (c) the resulting lighting using conventional OpenGL lighting. (d) a raytraced output showing the effects of shadowing more clearly.*

Given a 3D model, adjusting lighting for some desired appearance usually requires tweaking light properties like positions, directions and intensity. *Crayon Lighting* is a goal-oriented approach to set up lighting for a given scene. The user sketches "bright" and "dark" regions on the scene to describe desired appearance. The system then automatically and interactively infers various lighting parameters that produces these effects, assuming a simple lighting model. The user can then transform the model and sketch more desired effects, setting up the overall lighting progressively.

My research contributions in this area are as follows:

1. My overall contribution is to make the inverse-lighting problem interactive. Even for a simple lighting model like ours, this is very challenging. Specific contributions below contribute to the overall interactivity.

2. A hashing technique to create good initial guesses for the complex optimization problem. This greatly speeds up the rate of convergence. This technique incorporates

discontinuous self-shadowing effects thereby allowing limited shadow design.

3. An optimization technique that incorporates creating newer lighting configuration without undoing the product of previous design sessions. This makes progressive lighting design possible.

4. Effective use of the GPU to compute the objective function. Specifically this uses a multi-pass algorithm to compute the sum of values in a texture.

5. A unique iterative optimization framework that increases speed of convergence by dividing the solution space of the original problem into more tractable optimization problems.

Chapter 5 discusses this research work in detail. This work is also summarized in [118] and [119].

2.4 Sketches as Non-Photorealistic Renditions

Several sketch-based applications in design and art attach value to the original form of the sketches instead of its refined or recognized form. This research creates a pipeline for sketches that produces static and animated renditions at interactive rates as strokes are being added, moved and deleted by the user. This pipeline requires only two parameters that be refined by the user interactively. This work also makes several contributions in the area of non-photorealistic rendering and thus, reveals the intimate relationship between sketch-based and non-photorealistic computer graphics.

My contributions in this area are as follows:

1. A pipeline that simplifies strokes on the screen efficiently as they are added, deleted or moved. This pipeline operates virtually transparent to the user, as it uses only two user-defined parameters. Both these parameters can be changed interactively.

2. Temporally coherent renditions while working in the dynamic and interactive setting described above. This is an important consideration for animated renditions.

(a) (b) (c)

Figure 2.4: ***Rendering a sketch with fidelity.*** *(a) a 2D artistic sketch of a lamp. Thumbnail shows its appearance when zoomed out. (b) the same lamp simplified while being zoomed out. Strokes are taken in their entirety and are not discriminated by color. (c) When strokes are segmented and color is considered during simplification, the rendition more closely resembles the original sketch. The bigger figures in (b) and (c) show the simplified thumbnails magnified for better illustration.*

3. A method to produce local visibility cues that cause a global occlusion effect. A previously known method is implemented at no extra cost because of our design.

Chapter 6 discusses this research work in detail. This work is also summarized in [109] and [110].

Chapter 3

Sketch-based 3D Modeling

3.1 The Problem and the Applications

Designers in almost all professions use sketches during the early stages of conceptual design. This is especially true in case of architectural design, where initial ideas concentrate on subjective and artistic aspects. At such stages, the design ideas are nascent and perhaps too vague to be expressed precisely in terms of CAD models. Besides, the ambiguous and shabby form of the sketches play an important role in the evolution of the design itself. When the design becomes precise enough, wooden models or CAD models are constructed to convey it to customers and peers. Construction of these models involves looking at the sketches and drawing a CAD model on a computer from scratch. This divide between the initial design and finished prototype stages can be bridged if the sketches could be directly converted into 3D models. The area that deals with this problem is called sketch-based 3D modeling.

In the absence of sophisticated hardware, a sketch drawn on paper would usually be digitally scanned and processed to obtain an line drawing. Earlier tablet devices provided an opaque surface on which one could write on a stylus. Such devices had to be plugged into a computer and the sketch viewed on the screen, as it was being drawn on the tablet surface. Recent commodity tablet devices have integrated the tablet surface with a display, thereby providing a true pencil-and-paper experience of being able to see what is being drawn on the tablet surface itself. Popularity of such devices in the form of external tablets or tablet PCs

has revolutionized the area of sketch-based computing and brought the problem of sketch-based modeling from being mostly an academic interest into the realm of applications of daily, affordable and commercial use.

3.2 The Challenges

Mathematically, three orthogonal views of an object are enough in many cases to construct the 3D model for that object[1]. A sketch provides a single projection of a 3D model, and thus by itself theoretically corresponds to an infinite number of 3D models. Thus the problem of inferring the geometry of the 3D model uniquely from a single sketch is mathematically insoluble. However, when presented with the sketch of a model, most humans have no problem in instantaneously inferring uniquely the implied 3D object. Such compelling empirical evidence of preference of one geometric configuration over others suggests that a computational method to infer the correct 3D model may be possible for many kinds of sketches.

The advent of commodity tablet devices has also raised the expectation of sketch-based modeling applications to be not only successful, but also interactive. Since sketches can be drawn and viewed in real-time, converting them into refined 3D models is also expected to happen within a few seconds for the process to be satisfactorily interactive. The holy grail in the area of sketch-based modeling is thus to provide an "enhanced pencil-and-paper" experience. The contributions of this thesis in this area concentrate mainly on making various steps in the model construction interactive.

3.3 Related Work

The problem of converting line drawings into 3D models has been popular since the 1970s, and some of the earliest references to it can be found in the literature of artificial intelligence. One of the earliest attempts at sketching systems was *Sketchpad*[116]. Earliest approaches classified the edges of a line drawings into convex and concave contours of the object by

[1]It is possible that some features hide behind others in all three views, thereby constructing a "hull" of the 3D object instead of its precise geometry

edge-labeling[48, 56]. Using these labels, it was possible to define the topology of many types of objects from their single-view sketches. Other methods follow similar strategies: inferring 3D geometry based on identifying and classifying local parts of the sketch as corners, oriented faces, etc.[10].

The space of 3D models is too vast, and the sketches drawn for such models differ greatly in their nature and complexity. Therefore most practical sketch-based systems concentrate on building 3D models of specific classes from their respective sketches. Solid geometric models used in engineering or architectural design have been the most popular class[39, 73, 74, 133, 106, 49]. Vegetative models of plants and flowers have been created from sketches[51]. Design of rotund, freeform objects with curved surfaces has been proposed[50, 58, 61]. Our work concentrates on creating 3D rigid body objects from sketches.

Most sketch-based modeling systems can be divided into roughly three categories– recognition, interpretation/reconstruction and gesture-based, with many systems choosing a hybrid approach. The method of recognition aims at building a database of known topologies of objects and then matching a candidate sketch to the closest known object. Matching criteria are usually based on statistical learning of observed facets of the objects in their known projections[74]. While such approaches are promising and often more successful at recognizing known topologies than others, they are not scalable in general to include a large class of objects. Instead of recognizing entire objects, such approaches are used to recognize localized facets and another method is employed to construct the 3D model. The Chateau system[49] suggests candidate 3D objects as the user adds edges to the sketch, by matching the existing sketch with a known database of edge and plane configurations.

Interpretation/reconstruction methods attempt to infer the geometry of the 3D model from a sketch without referring to a database of known objects and their projections. These methods are very popular because they are not confined to succeed only for a known small subset of objects. The simplest solutions in this category use edge labeling as a starting point to infer the topology, which is then refined to obtain precise 3D coordinates[104]. One of the most popular formulations arises from the notion of *inflation*: considering the line drawing as a planar graph and assigning an approximate depth to every vertex, thereby inflating

it into a 3D model. Marill[76] proposed one of the first solutions to sketch reconstruction using such an inflation approach, minimizing the difference between edge angles of the sketch and the 3D model. Subsequently, many extensions to this optimization framework have been suggested: Leclerc *et al.*[66], Grimstead *et al.*[38], Oh *et al.*[83], Lipson *et al.*[73], etc. The Teddy system[50] inflates sketches to create rotund, toy-like objects by using distance-transform-based inflation methods. Another typical approach to inflation is to identify 2D regularities in the line drawings–hints in the sketch that correspond to some geometric property of the implied 3D model. Examples of 2D regularities include planar faces, parallelism and orthogonality of lines and faces, three-edge corners, etc. Our system, *SMARTPAPER*, belongs to this category and delivers an interactive system by providing several extensions to previous work. Identifying additional constraints like mirror symmetry in objects[92], constraints in 3D geometry as a post-process[84], etc. contribute to making the optimization more tractable, thereby reducing the probability of arriving at (visibly) incorrect local minima. The challenges in such a framework tend to be defining an objective function that accurately captures the correspondence between sketches and 3D models, non-linearity and multi-dimensional complexity of the resulting optimization and related problems of the speed and accuracy of convergence. An alternative approach taken by Varley *et al.*[124] in the RIBALD system is to first label a given line drawing using the Huffman-Clowes scheme[48], then categorizing the line drawing into a specific type of topology, and then constructing a 3D model in parts using a series of vertex, edge and face "moves".

The category of gesture-based systems arises from the observation that many complex topologies can be easily expressed procedurally. For example, many complex objects can be identified as surfaces of revolution, linear or non-linear extrusion, etc. Thus, this category of proposed solutions converts 2D inputs into 3D models by constraining the space of possible solutions. This is done by either using gestures instead of actual objects, or offering a constrained user interface. Thus, in contrast with the earlier categories that worked with actual sketches of objects, gestured-based systems often allow the user to draw strokes that do not correspond to the actual geometry of the object, but hints towards their topology. Quick Sketch[27] allows the user to draw rough 2D curves and supports extruding them along

a curve to construct 3D models. The SKETCH system[133] uses intuitive gestures to define a 3D model progressively. The GIDES system[91] allows the user to create 3D models by sketching multiple 2D views or using standard gestures for some 3D configurations. Cherlin *et al.*[17] use scribbling and curved-based gestures to bend surfaces and create complex 3D models. Naya *et al.*[80] allow the user to draw tentative construction lines that acts as "scaffolds" to the actual sketch and aid in the reconstruction of its 3D geometry. In the system designed by Tolba *et al.*[120, 121] the user iteratively sketches objects on paper, scans them into the system and aligns them.

There are some sketching systems whose main contribution is to design an interface metaphor that is more intuitive and geared towards sketch-based modeling. The Chateau system[49] suggests candidate 3D models even before the user has finished drawing the entire sketch. Karpenko *et al.*[60] use epipolar geometry to create a 3D model by multi-view sketching. Tolba *et al.*[120, 121] use a perspective grid to align strokes and interpret them in 3D.

3.4 SMARTPAPER: A primer

SMARTPAPER is a design-by-sketches system that allows construction of objects made with planar faces, and objects of extrusion that have curved surfaces. It presents a unified sketching environment that supports both direct and gestured sketching, with emphasis on the former. It gives more freedom to the user by supporting casual sketching styles, where several overlapping discontinuous strokes could be sketched (Figure 3.4). In addition to sketching objects from scratch, a user can sketch directly on a 3D model to add to its geometry (Figure 3.5(b)). *SMARTPAPER* provides a feedback system that allows a user to examine the interpretation made and provide hints accordingly to improve its performance, leading to greater user satisfaction (Section 3.10). In addition to converting sketches into 3D models, *SMARTPAPER* offers a compendium of Computational Solid Geometry (CSG) operations, synergistically resulting in a practical proof-of-concept system. Also, it employs non-photorealistic rendering techniques to give the reconstructed objects a sketchy look. As

will become evident in Section 3.5, from a user interface perspective, the system combines seamlessly various 2D and 3D operations such 2D sketching, sketching on 3D, cutting and joining.

3.5 User Interface

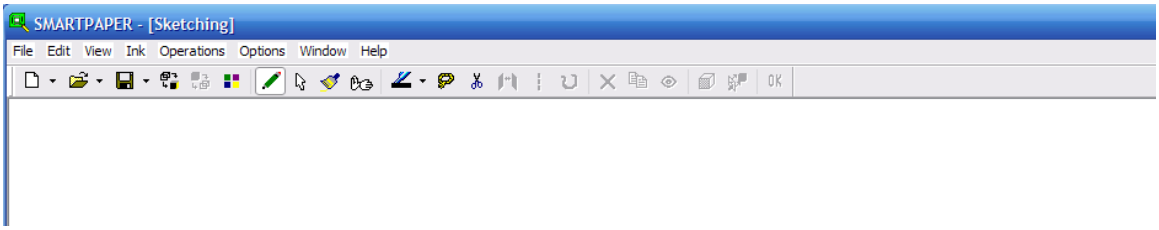


Figure 3.1: ***User interface of SMARTPAPER.*** *The user sketches on the blank window and then presses the appropriate button to specify the operation.*

SMARTPAPER is implemented on the Tablet PC. It presents a directly sketchable blank window. Alternatively, it can provide standard templates like a ground or a room as a starting point for the user. The user can directly begin to sketch in the window with a stylus. The stylus emulates well a standard pencil, by enabling the user to erase sketched strokes with its other end.

The general theme of working with *SMARTPAPER* is to sketch strokes and then specify what they mean. For example, to sketch an object the user places strokes as s/he likes and then specifies the "reconstruct" operation. This allows imperfections in all operations requiring sketchy input.

To construct simple objects like pyramids, prisms, frustums, etc. the user can directly sketch them. All edges of the object must be drawn in this case. The user may draw hidden edges using dotted lines, and this gesture is used accordingly when the 3D object is constructed. Sometimes it is cumbersome for the user to draw all edges of an object. Objects that are extrusions of planar profiles can be constructed by sketching the closed profile and an extruding arrow (Figure 3.5(a)). To construct more complicated objects,

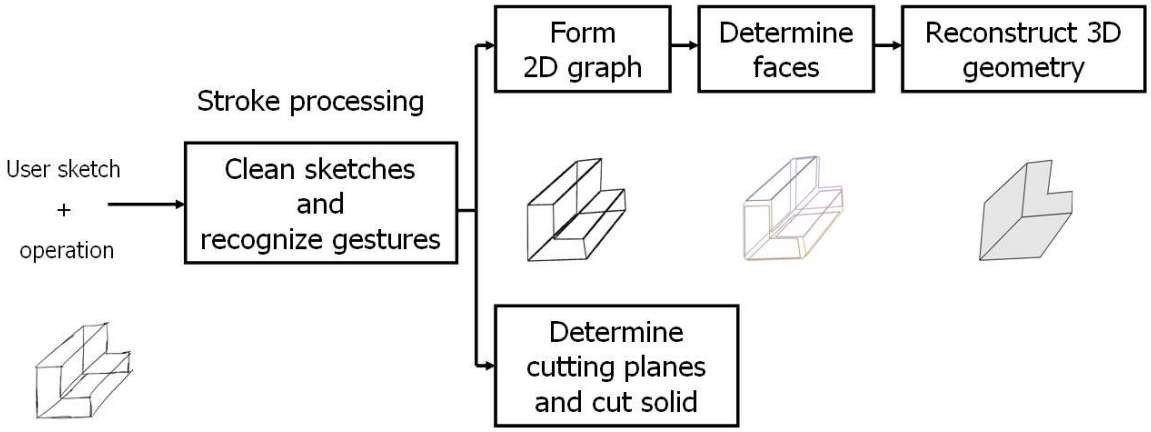


Figure 3.2: ***The processing pipeline of SMARTPAPER.***

the user can employ an incremental process by creating a simple object, transforming it, sketching directly on it to augment it and so on (Figure 3.5(b)), or by combining several simpler objects (Figure 3.5(c-h)). Whole objects may be constructed in multiple sessions, making the process truly incremental.

In order to cut an object, the user can directly draw the cut edges on the object or draw a closed curve and an extruding arrow to cut a hole through it. Pressing the "cut" button completes the cutting operation (Figure 3.8.3). In order to select an object out of many, the user may either encircle it and click on the "lasso" button, or simply double-click on it. In order to join two objects by a face, the user first selects the two faces and then clicks on the "join" button. The user can annotate an object freely by selecting the "annotate" button and then freely scribbling on the current object. All such strokes are projected onto the object. The user can also define a custom plane with respect to the 3D geometry of the object, and sketch on it.

The user can create several layers and create objects in each layer. This is analogous to a designer using transparency sheets or tracing paper to construct different aspects of his/her design. This feature removes the limitation of space created by the screen size.

3.6 System Overview

The block diagram in Figure 3.2 shows the processing pipeline of *SMARTPAPER*. This pipeline shows operations for which a sketched input is required, i.e. drawing a new object, augment existing ones, and cutting. Point-and-click operations such as joining two objects are not illustrated in this pipeline.

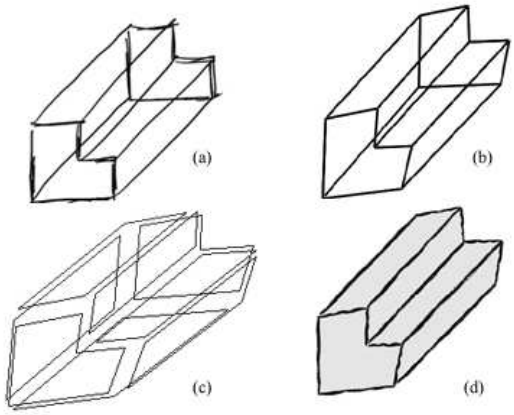


Figure 3.3: ***3D reconstruction pipeline.*** *(a) the input rough sketch (b) the cleaned 2D graph (c) the recognized faces and (d) the reconstructed 3D object.*

When a set of strokes and the user command is given, over tracing and other imperfections are removed, as explained in Section 3.7.1. This preprocessing is done on all sketched strokes irrespective of the operation to be performed. A direct consequence of this is that such sketching imperfections are allowed while drawing as well as cutting. Any gestures that are part of the set are recognized when queried.

The top part of the pipeline enumerates the 3D reconstruction operation, while the lower part represents the cutting operation. If the specified operation is 3D reconstruction, then 2D graphs are formed from the set of strokes (Section 3.7.2). A set of valid cycles forming faces of the sketched object is then determined (Section 3.8.1). 3D-model reconstruction is then performed by optimization to reconstruct the final 3D object (Section 3.8.2). This process can be intervened by user feedback (Section 3.10). Figure 3.3 illustrates various steps in reconstruction.

If the specified operation is cutting, then ray casting is used to determine a set of cutting planes (Section 3.8.3). The object is then cut resulting in two or more objects.

Our system makes a **Closed Object assumption** about a drawn object: only solid objects that are homeomorphic to a sphere can be drawn (this assumption is removed later in Chapter 4). Objects that are not strictly closed (e.g. objects with holes) can be constructed by a series of operations. A series of planes resulting in an open object cannot be drawn.

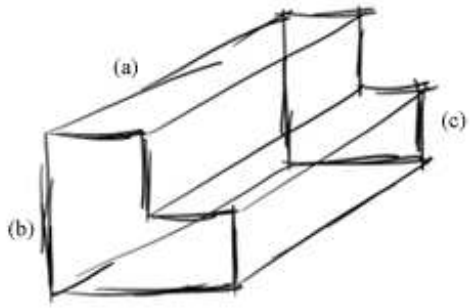


Figure 3.4: ***Examples of over tracing.*** *(a) over tracing done to complete an edge, (b) unintended over tracing, and (c) over tracing to highlight an edge.*

This class of objects encompasses all objects that can be physically constructed, and hence does not impede applications like architectural design.

3.7 2D Processing

The set of strokes is subject to initial pre-processing. This process cleans up the sketch and represents the strokes in a consistent format for 3D reconstruction or other operations.

3.7.1 Sketch Cleaning

Whenever strokes are drawn either for 3D reconstruction or cutting, some common preprocessing is done. Two functions are achieved in this block: over tracing consolidation and gesture recognition.

Over Tracing

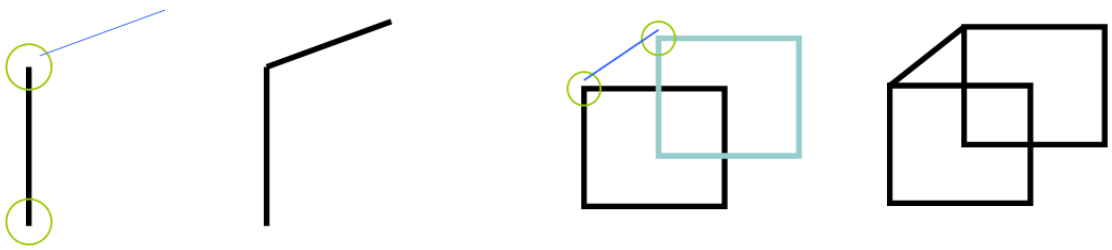
A sketch is often drawn as a series of discontinuous strokes to illustrate edges (Figure 3.4). Such over tracing could either be done unintentionally, to highlight an edge or to simply complete an edge. These strokes are grouped together to form the continuous edge(s) that they collectively represent. This grouping is achieved in two passes over the set of strokes. In the first pass, we find pairs of strokes that qualify as over traced strokes. A pair (A, B) qualifies when they have nearly equal slopes, and at least one end point of B lies in the x and

y ranges of the end points of A. Let $A(e_1, e_2)$ and $B(e_3, e_4)$ be the two strokes, and let e_3 be the end point of B lying in the range of e_1 and e_2. Now, let $length(e_2, e_4) < length(e_1, e_4)$. Then B is changed to $B'(e_2, e_4)$. At the end of this pass, overlapping segments become segments having one common end point. The second pass then culls all vertices, all edges incident to which have nearly equal slope. Thus, for example, if no other edges are incident on e_2, $A(e_1, e_2)$ and $B'(e_2, e_4)$ form a single stroke $A'(e_1, e_4)$. These vertices cannot be removed in the first pass itself because the incidence of *all* edges has to be known before a vertex can be culled. The over traced sketch in Figure 3.3(a) is cleaned and interpreted as Figure 3.3(b).

3.7.2 Graph Generation

Every sketched object is represented as a 2D graph of vertices and edges. A connectivity matrix is maintained for each object. Each vertex stores (x,y) coordinates.

The set of input strokes is distributed among the existing set of objects depending on their proximity with the projection of existing objects in the current viewing plane, as the user does not explicitly specify whether the strokes specify new objects or augmentation to existing ones. Strokes that do not augment existing objects create new graphs. Two graphs are merged if a stroke with one end point in each of the graphs occurs. If an object is drawn by extrusion, then two copies of the profile are made and are connected by edges parallel to the direction of extrusion.



Clustering as proposed by Shpitalni *et al.*[113] is used to group vertices close to each other in the graph. As edges are added to the graph, all end points within a distance of δ

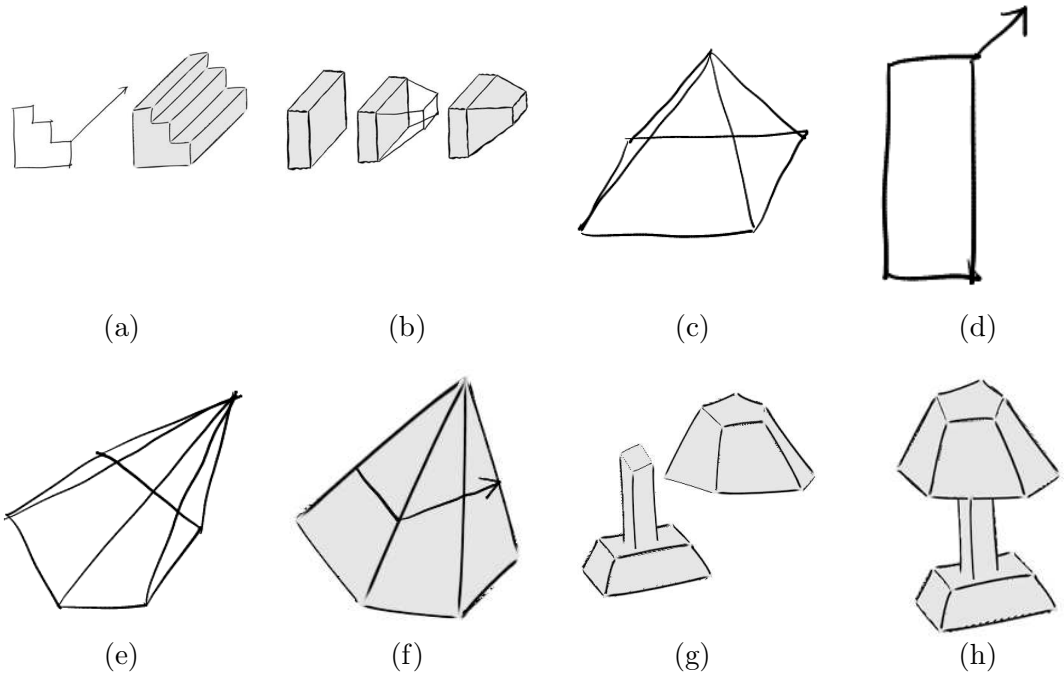


Figure 3.5: ***Construction of 3D models in SMARTPAPER.*** *(a) Constructing a 3D model by extrusion (b) Adding to an existing 3D model (c-h) Constructing an object by parts.*

from an existing vertex are grouped with it. Incorrect thresholding can be corrected in the feedback system to allow sketches with lesser precision (Section 3.10).

To uphold the Closed Object assumption, each vertex must have degree at least 3. This check is used to clean unnecessary vertices, which may be created when a single stroke is incorrectly interpreted as two or more edges due to its slope. The final representation is a 2D graph with vertex degree at least 3 and order at least 4. Augmented and newly sketched objects undergo 3D geometry reconstruction.

3.8 3D Geometry Reconstruction

This section encapsulates the functionality for determining the 3D aspects of each unconstructed object, namely face determination and iterative 3D reconstruction. For the following discussion, for a given graph G, $V(G)$ and $E(G)$ represent the vertex set and edge

set of G, respectively, while $G - e$ is the sub-graph obtained by deleting the edge e from G.

3.8.1 Face Determination

We determine the faces of the object from its representative graph G as illustrated in Figure 3.3(c). All faces are cycles of G; however the converse is not true. Shpitalni *et al.*[112] discuss a face determination algorithm based on an A* or branch-and-bound search. Their algorithm is too slow because it performs an exhaustive search on the set of all possible cycles, and the Closed Object assumption allows us to formulate a definition which simplifies face determination. In the definition, graph G represents the 2D graph of an unconstructed object:

Definition A: All edges of graph G are part of exactly two faces. Every valid face F of G is such that for all pairs $v_1, v_2 \in V(G)$ that are in F, the shortest v_1, v_2-path in G is of the same length as the v_1, v_2-path in F.

Justification: The first statement is a property of closed, non-laminar, rigid objects. If the second statement is not true, let P be the shortest v_1, v_2-path in G and let P' be the shorter v_1, v_2-path in F. There is at least one edge in P not in P'. PP' thus creates a smaller closed walk and hence a smaller cycle C containing v_1 and v_2. The edge set $E(C) - (E(C) \cap E(F))$ divides face F into two or more different planes, which is a contradiction as F is a valid face and hence is planar.

We propose two algorithms for face determination that directly determine all valid faces of G instead of examining all possible cycles of G for validity. While the first algorithm takes advantage of interactive drawing cues and is fast, the second algorithm is theoretically more robust albeit slower.

Algorithm 1: Edge Coherence algorithm

Humans draw objects according to how they perceive them. Often our drawing styles construct the object part by part. This algorithm examines the sequence S in which strokes are drawn to search for cycles that form valid faces. Note that consolidating over traced strokes does not disturb this sequence, as the earliest stroke in a series of over tracing strokes

is used to determine the order of the consolidated stroke.

If the object is drawn face by face, then edges of such faces are adjacent in S and thus, a linear traversal of edges in their order directly recognizes some faces. If two adjacent edges in this order do not have a common end vertex, the algorithm "looks ahead" in S to find an edge connected to the current edge. Our preliminary tests showed that a look ahead of 1 was sufficient for simple primitives like prisms, pyramids, etc. This algorithm works in two passes. Pseudo code for it is provided in Algorithm 1 and is illustrated in Figure 3.6. The main advantage of this algorithm is its speed. The amortized cost of the first pass is $O(e)$, while the second pass takes $O(e1 \cdot n^2)$ time, where e1 is the number of edges that are not part of two faces and n is the number of vertices of G. The first pass amortizes the second pass depending upon how the object is drawn.

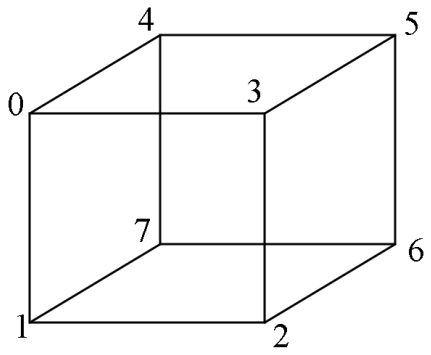


Figure 3.6: ***Edge coherence algorithm.*** *Edges are drawn in order 01-12-23-30-04-45-35-56-62-47-76-71 (a typical way of drawing a cube). The first pass determines faces {01-12-23-30} and {04-45-35-03} and partial faces {56-62}, {47-71} and {76}. The second pass completes these faces.*

The speed of this algorithm is due to the fact that humans intuitively draw objects according to how they perceive them and not in a completely random fashion. However, this assumption is not theoretically strong. Also, if all the strokes collectively representing an edge of the object are erased and redrawn, this order may be disturbed. To improve the performance of *SMARTPAPER*, we have devised a second algorithm for face determination. This is a theoretically more robust albeit considerably slower algorithm, but it still directly determines all valid faces and hence is an improvement over [112].

Algorithm 2: Modified Dijkstra's algorithm

Algorithm 1 Edge Coherence Algorithm

Input: Graph G with vertices V(G). E(G) is in order of sketching
k: Look-ahead factor
Output: Complete set of faces L and incomplete set of faces X
$L \leftarrow \{\}$
$F \leftarrow \{\}$
for all $e \in E(G)$ **do**
  if e is adjacent to any edge in F **then**
    $F \leftarrow F \bigcup \{e\}$
  else
    if an edge e' ahead of e within k in $E(G)$ is adjacent to F **then**
      $F \leftarrow F \bigcup \{e\}$
    else
      Add F to X
      $F \leftarrow \{\}$
    end if
  end if
  if F is a cycle **then**
    $L \leftarrow L \bigcup \{F\}$
    $F \leftarrow \{\}$
  end if
end for
$G' \leftarrow (V(G), \{e \in E(G) : \text{e is not part of two faces}\})$
for all $e = (v_1, v_2) \in E(G')$ **do**
  $C \leftarrow$ shortest v_1, v_2 path in$G' - e$
  $L \leftarrow L \bigcup \{C \bigcup \{e\}\}$
  $G' \leftarrow G' - e$
end for

This algorithm finds all faces for each edge in the graph such that definition A is satisfied. For every edge e, we remove the edge from the graph and find a shortest path between its end points. We employ Dijkstra's shortest path algorithm for this purpose.

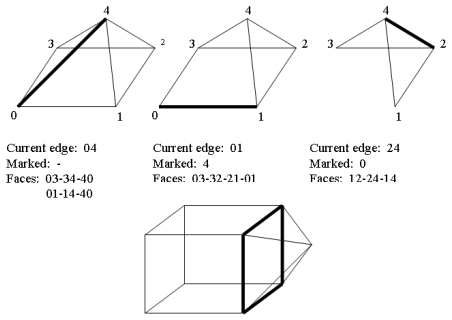


Figure 3.7: ***Edge coherence algorithm.*** *Edges are drawn in order 01-12-23-30-04-45-35-56-62-47-76-71 (a typical way of drawing a cube). The first pass determines faces {01-12-23-30} and {04-45-35-03} and partial faces {56-62}, {47-71} and {76}. The second pass completes these faces.*

We maintain a set R of edges for which all faces containing those edges have been found. If any edge e' in this set shares an end vertex with the current edge e, we mark the other end vertex of e' as traversed. This prevents the Dijkstra's algorithm from finding a shortest path which contains any edge in R in it. Intuitively, once all the faces of which an edge is part of have been determined, the edge cannot be part of any face determined in future, even if it is on some shortest path. Thus, this satisfies definition A and hence the algorithm is correct. Pseudo code for the algorithm is shown in Algorithm 2. The last step in the pseudo code removes any fictitious "internal" faces as illustrated in Figure 3.7. Thus, the algorithm finally produces two faces per edge.

Both algorithms update face lists of objects incrementally. It is important to note that if a sketch augments an existing object, only faces containing vertices to which new edges are incident are determined. If an "incomplete" object is drawn, then the results of both algorithms may be verified from the 3D model: if the face is extremely non-planar, it is flagged as an erroneous face and is not triangulated.

3.8.2 Iterative 3D Reconstruction

SMARTPAPER uses a modification of the optimization process proposed in [73], which we summarize here for completeness. This step "inflates" the planar sketch by assigning suitable Z-coordinates to each vertex of the graph of the object, which are determined using a set of geometric properties. The properties used are planarity of faces, parallelism

Algorithm 2 Modified Dijkstra algorithm

Input: Graph G with vertices $V(G)$ and edges $E(G)$
Output: Set L of all faces of the object
$S \subseteq E(G)$: Set of edges that are not part of at least 2 faces
$R \subseteq E(G)$: Set of edges that are part of at least 2 faces
M: Matrix having faces as rows and edges as columns
$L \leftarrow$ Known faces of the object or null
/*Initialize S */
for all $e \in E(G)$ **do**
  if e is not part of 2 faces according to M **then**
    $S \leftarrow S \bigcup \{e\}$
  end if
end for
/*If a current object is being augmented, S is a proper subset of $E(G)$*/
while $S \neq \{\}$ **do**
  $e = (v_1, v_2) \in S$
  for all $e' \in R$ **do**
    if e' shares an end vertex with e **then**
      Mark end vertices of e' as traversed
    end if
  end for
  while Edge-disjoint v_1, v_2 paths exist in $G - e$ **do**
    $p \leftarrow v_1, v_2$ shortest path in $G - e$
    $L \leftarrow L \bigcup \{p \bigcup \{e\}\}$
    Update M with $\{p \bigcup \{e\}\}$
    Temporarily remove p from $E(G)$
  end while
  $R \leftarrow R \bigcup \{e\}$
  $G \leftarrow G - e$
end while
/*For all faces that have edges part of > 2 faces, remove them */
for all Columns $M(., i)$ of M **do**
  if $M(., i)$ contains more than two faces **then**
    Mark $M(., i)$
  end if
end for
for all Rows $M(i, .)$ of M **do**
  if $M(i, .)$ is made up of only marked columns **then**
    Remove corresponding face from L
  end if
end for

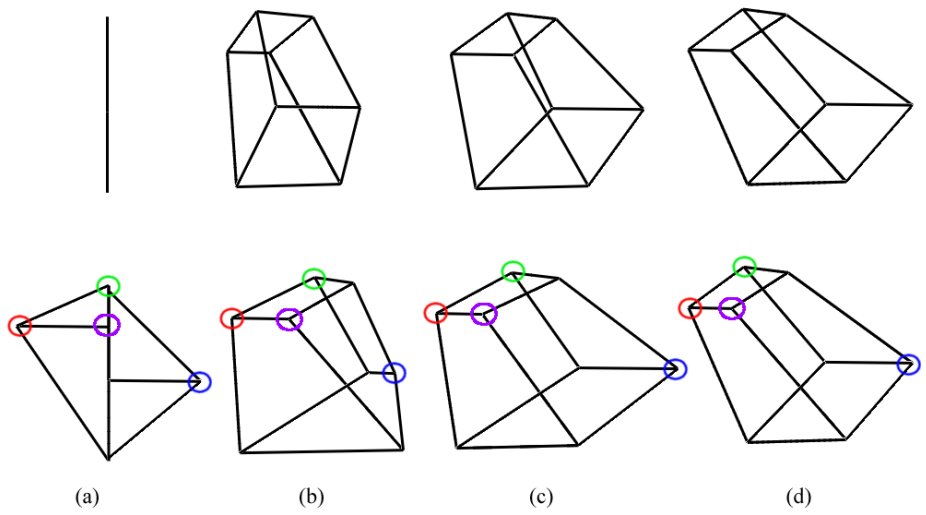


Figure 3.8: ***Inflation of sketch by optimization.*** *Upper and lower rows show how inflation without and with layering respectively: (a) initial condition, (b) and (c) intermediate states, and (d) the final object. The colored circles show how vertices move during inflation.*

and perpendicularity between edges, comparing edge lengths and mutual angles at corners. The general idea of this method is to duplicate the properties available in the 2D sketch in the 3D object. Each constraint is expressed as a factor relating a 2D property to a 3D property. A compliance function $F(Z)$ is computed for an 3D configuration by summing the contribution of the factors. The final compliance function to be optimized takes the form

$$F(Z) = W \sum A$$

where A is the vector of all factors and W is a weighting function. This is an n-dimensional optimization problem. We use Brent's minimization technique[98] to solve it as a set of 1-dimensional optimization problems by cycling through all vertices. For a detailed discussion on factors and formulation of the problem, please refer to [73].

A critical issue is the dependence of the result of optimization on the initial guess. If all Z values are initialized to 0, incorrect local minima are often reached, which is visually indicated by a deformed or collapsed reconstructed object (Figure 3.8 (a) top). The user

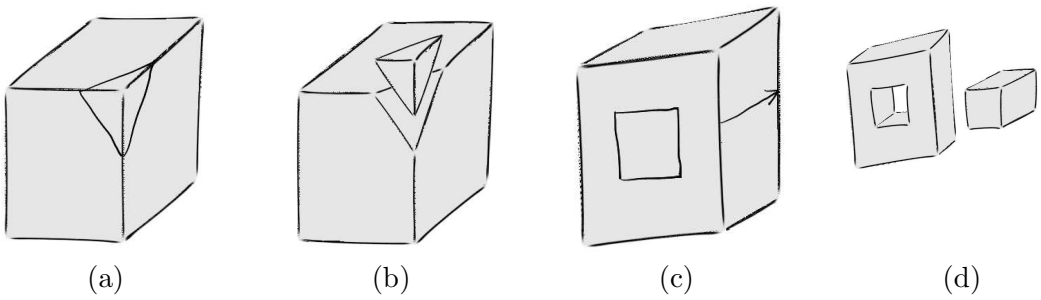


Figure 3.9: ***Cutting a 3D model in SMARTPAPER.*** *(a) A planar cut specified directly by the cut edges (b) the result of the planar cut (c) cutting by extrusion (d) result of the extruded cut.*

can provide hints (dotted lines) in the sketch by specifying hidden edges. If such a hint is available, then the object is divided into 3 Z-layers. One layer consists of vertices to which only hidden edges are incident ($Z = -10$, say), the second layer consists of vertices to which only visible edges are incident ($Z = 10$, say) and a third layer consists of all remaining vertices on silhouettes ($Z = 0$) , as illustrated in Figure 3.8 (a) bottom. This partially inflates the object and our tests show that this produces a better initial guess leading to fewer cases of convergence to incorrect local minima. This method is a simplified version of that proposed by Company *et al.*[88]. It is important to note that 3D information of vertices of existing objects to which new edges are not incident is retained, and hence 3D reconstruction is incrementally performed.

The final representation of the object is a graph with augmented 3D information. This representation is similar to a boundary representation. When the user wishes to sketch from a new view point, all objects are re-projected onto the current viewing plane. These projections are used to determine if strokes drawn augment existing objects or create new ones, and for object selection.

3.8.3 Cutting

The user can specify cutting in two ways. A cutting plane can be directly drawn on the object. Alternatively, an open or closed profile and an extruding arrow can also be drawn. Due to the unified preprocessing in the pipeline, over tracing is allowed in these specifications

as well.

Because our representation scheme is similar to B-reps, the actual cutting algorithm is similar to that proposed in [75]. The user strokes are converted into a 2D graph after preprocessing. A ray is cast from the eye position into the scene through both end vertices of each edge. For the extruding arrow, the shaft edge is used to determine the direction of extrusion by ray casting. A set of cutting planes is obtained and a set F of faces created by them is determined. An algorithm similar in idea to that discussed in [75] then completes this operation.

3.8.4 Joining

Two objects can be joined face-to-face by selecting each object and then selecting the face that is to be joined to the selected face of the other. A simple coordinate transformation "sticks" the two faces together. Now one object can be translated in the plane of the selected face or rotated about its normal for positioning. The join operation joins the two objects after the user commits this positioning. Since objects are drawn roughly, two faces are seldom congruent to each other. Therefore the user can choose to deform one of the selected faces so that it coincides seamlessly with the other selected face. Alternatively, the user can choose to simply stick the two faces without deformation. This is desirable when the two faces are meant to be different in size, like those of a table top and a rectangular leg while constructing a table.

3.9 Gestures

SMARTPAPER recognizes standard gestures for gestured drawing and cutting. Gestures can be used to create a 3D model by extrusion, cut a 3D object by extrusion or modify the geometry of a 3D model by placing additional constraints on it. Gestures are also sketched and are part of the input set of strokes. The reconstruction and cutting modules query a gesture recognizer for gestures. The set of strokes is then passed to the recognizing or cutting module, depending upon the operation specified.

Figures 3.10 and 3.11 summarize all gestures supported by *SMARTPAPER*. The drawing

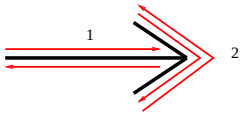
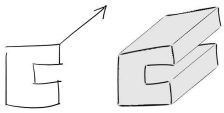
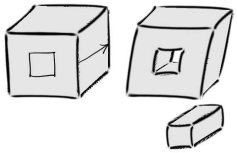
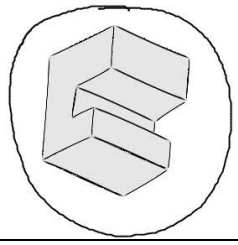
	Arrow gesture: numbers indicate sequence of drawing and red arrows indicate possible pen paths.
	Use of the arrow gesture in drawing by extrusion.
	Use of the arrow gesture in cutting by extrusion.
	Selection of an object.

Figure 3.10: ***Gestures in drawing and editing modes of SMARTPAPER.***

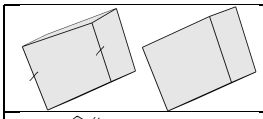
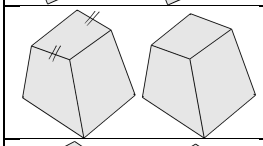
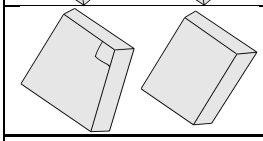
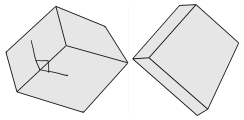
	Making two edges congruent to each other.
	Making two edges of a face parallel to each other.
	Making two connected edges perpendicular to each other.
	Making two faces perpendicular to each other. The right figure is rotated to show the result.

Figure 3.11: ***Gestures in feedback system.*** *In the first column, figures on the right show the result of the respective operation.*

convention for an arrow is to draw two strokes, the first being the shaft and the second being the head, drawn from end to end.

Additional constraints can be placed on the structure of 3D objects after they have been created. All algorithms use and modify the properties of the underlying graph of the object. To visualize these algorithms it is helpful to imagine the solid as a region enclosed by several bounded or unbounded planes and move them as per the operation. Our current implementation supports the following structural changes:

3.9.1 Making two edges of an object parallel to each other

This operation is useful in making two edges parallel. The two edges are specified by drawing two parallel lines across them as shown in Figure 3.11. By default, the second edge is made parallel to the first. The affected vertices are moved as explained in Algorithm 4.

3.9.2 Making two edges of a face perpendicular to each other

This operation is useful to make a face rectangular. It is specified by drawing a bracket between the two edges as shown in Figure 3.11. The two edges specified have to belong to the same face. After determining the edges from the bracket gesture, they are made perpendicular by moving vertices of the graph using Algorithm 5.

3.9.3 Making two faces of the object perpendicular to each other

This is specified by drawing two lines along the two faces meeting near their common edges, and a bracket between them as shown in Figure 3.11. The two faces must share an edge. These faces are determined from the end points of the first two strokes by traditional ray-casting. Algorithm 6 is then used to accordingly move the affected vertices.

3.9.4 Making two faces of an object parallel to each other

The two faces are specified by drawing two parallel lines on them (one face is marked, the object is rotated and then the second face is marked). By default, the second face is changed to make it parallel to the first. Affected vertices are moved according to Algorithm 7 to complete the operation.

Our experience shows that an object can be quickly refined to regular shapes using these gestures. Regularity cannot be implicitly assumed in the sketch because it will artificially classify all sketches into a small class of regular objects. However, it can be achieved easily through operations like these.

3.10 Feedback System

There are many uncertainties in sketch reconstruction and sketching systems in general. The reconstruction process is based on optimization and heuristics and hence results may not always satisfy the user. Also, the sketch can be unpredictably shabby leading to misinterpretation.

We feel that the best way to improve the performance of a system like *SMARTPAPER* is to facilitate user feedback and dialogue. If the user sees what the problem is, s/he can

Algorithm 3 MakeEdgesCongruent

Input: Edges e and $e'(v, v')$ of face F
$F' \leftarrow$ the face sharing e' with F
$e''(v, v'') \leftarrow$ the edge incident on v' in face F
$F'' \leftarrow$ the face sharing edge e'' with F
$e_1(v', v_1) \leftarrow$ the edge incident on v' not in F.
$e_2(v_1, v_2) \leftarrow$ the edge in F' incident on v_1 and is not e_1.
Move v' along e' till the length is equal to e.
Move v_1 an equal distance along e_2 in F' by an equal distance to keep the slope of e_1 unchanged.
/*Face F' has a new plane formed by new v', new v_1 and unchanged v''.*/
for all $V \in F''$, $V \bigcap \{v', v'', v_1\} = \{\}$ **do**
  $E \leftarrow$ edge incident on V that is not in F''.
  $V \leftarrow$ the intersection of E with new F''
end for

Algorithm 4 MakeEdgesParallel

Input: Edges e and $e'(v, v')$ of face F
$F' \leftarrow$ the face sharing edge e' with F
$e''(v, v'') \neq e' \leftarrow$ the edge incident to v' and in F
$e_1(v, v_1) \leftarrow$ the edge incident at v in F'.
Move v' along e'' till e and e' are parallel
$F' \leftarrow$ plane formed by v, v_1 and the new v'
for all $V \in F'$, $V \bigcap \{v, v'\} = \{\}$ **do**
  $E \leftarrow$ edge incident on V that is not in F'
  $V \leftarrow$ new intersection of E and the new F'
end for

Algorithm 5 MakeEdgesPerpendicular

Input: Edges e and $e'(v, v')$ of face F
$v \leftarrow$ the common vertex between e and e'. If there is no such vertex, then the operation cannot be performed.
$F' \leftarrow$ the face sharing e' with F
$e'' \leftarrow$ the edge in F other than e' incident on v'
$e_1(v, v_1) \leftarrow$ the edge incident at v which is not in F
Move v' along e'' till $e \perp e'$.
$F' \leftarrow$ new plane formed by v, new v' and v_1
for all $V \in F', V \bigcap \{v, v', v_1\} = \{\}$ **do**
  $E \leftarrow$ edge incident on V that is not in F'
  $V \leftarrow$ the intersection of E and the new F'
end for

Algorithm 6 MakeFacesPerpendicular

Input: Faces A and B with common edge $e(v_1, v_2)$
$n_A \leftarrow$ normal of A
$n_B \leftarrow$ normal of B
Move B' by computing a new normal $n'_B \perp n_A$
for all $V \in B, V \bigcap \{v_1, v_2\} = \{\}$ **do**
  $e' \leftarrow$ edge at v which is not in B
  $V \leftarrow$ intersection of e' and new B
end for

Algorithm 7 MakeFacesParallel

Input: Faces A and B with face F which shares an edge each with A and B
$n_A \leftarrow$ normal of A
$n_B \leftarrow$ normal of B
$e(v, v') \leftarrow$ edge in F which is shared with B
$e' \leftarrow$ the other edge in F that is incident on v'
$e''(v, v'') \leftarrow$ the edge in B incident at v and not in F
Move v' along e' till n_B equals n_A
Face B has a new plane formed by v, new v' and v''
for all $V \in B, V \bigcap \{v, v', v''\} = \{\}$ **do**
  $E \leftarrow$ edge incident at V that is not in face B
  $V \leftarrow$ intersection of E and B'
end for

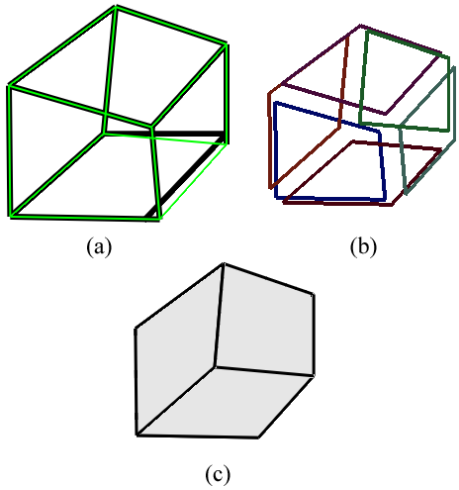


Figure 3.12: ***A single object shown in three views in the feedback system*** *(a) Sketch view (b) Face view (c) Object view.*

greatly assist in quickly correcting it. This is the motivation behind the feedback system in *SMARTPAPER*.

The goals of this system are effective visualization to enable the user to pinpoint the problem, allowance to changes and suggestions and quick response to them, and the provision of ways to refine a correctly reconstructed object. Visualization is important because an erroneous result often offers little insight into its cause. Therefore, three views of an object are offered (Figure 3.12).

3.10.1 Sketch View

This view shows how the system interpreted the user's strokes. If the sketch is dirty, then clustering may be incorrect leading to an erroneous reconstruction. This view shows the cleaned graph superimposed on the user's input (after preprocessing as described in Section 3.7.1). This makes any incorrect clustering obvious.

The user can manually suggest clustering by enclosing all the vertices to be clustered as shown in Figure 3.13 (b). When s/he presses OK, the points are combined, the whole graph is re-clustered and the results are immediately shown as in Figure 3.13 (c). It can be seen from these figures that correcting one clustering can produce wholly correct results. The user can then press the refresh button to make the other views consistent with this view.

3.10.2 Face View

The face view shows an exploded view of the object(Figure 3.12). This view is useful in verifying if the faces have been determined correctly. If any faces have been incompletely or incorrectly determined, the user can sketch a face directly in this view (Figure 3.14). Whenever a new face is sketched by the user, all edges that are part of more than two faces are marked, and all faces made of only marked edges are deleted.

3.10.3 Object View

This view shows the reconstructed object, which is topologically correct if the clustering and face determination was done correctly. This view is used to further refine the object

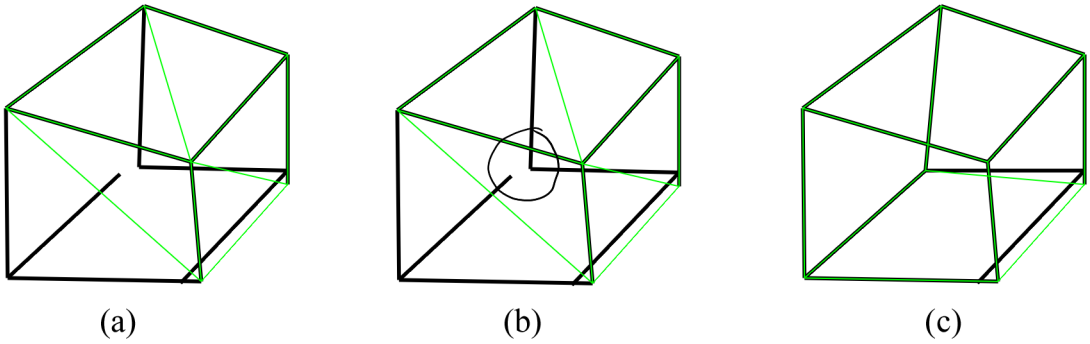


Figure 3.13: ***Example of clustering by giving a hint.*** *The thick black lines show the original sketch and the green lines show result of automatic clustering: (a) Initial sketch and clustering output, (b) the vertices to be clustered are marked by encircling, (c) the result of the hint, showing overall correct clustering by just one hint.*

by specifying additional constraints on its structure (Figure 3.11).

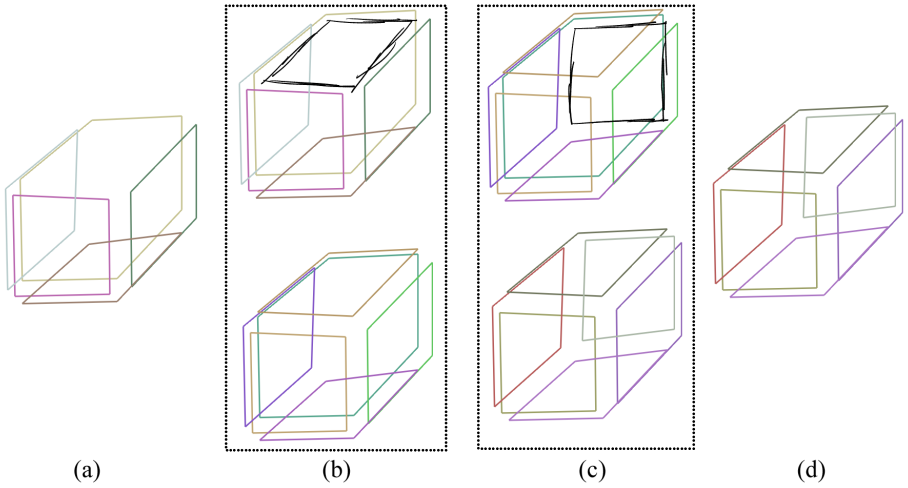


Figure 3.14: ***Gestures in the face view.*** *(a) a sample object with incorrectly determined faces, (b) and (c)upper figures show a face sketched and lower figures show their respective results, (d) the resultant object (with all faces correctly determined.)*

3.11 Construction of curved objects

SMARTPAPER supports drawing objects with curved edges by extrusion. Extrusion requires a closed profile. However, the user may draw a curve whose end points either do not

meet or cross each other. To generate a smooth closed curve from such input, we employ a modified version of the Snake algorithm [126, 5].

The original Snake algorithm is interactive in semi-automatic conditions where the user explicitly inputs an approximated closed curve that is used as an initial guess, resulting in fast convergence. This is not feasible in our case as this problem is transparent to the user. Therefore we fit a circle using least-square approximation to the stroke data and use it as the initial guess. This potentially decreases the rate of convergence and hence we modify the implementation of the Snake algorithm in the following ways:

1. It is required to compute the nearest point in the input data (edge data) to a point on the closed curve in a particular direction during each iteration. This is an expensive operation if there are a lot of points in the input data. Therefore, we allow 3 initial expensive iterations to "guess" a subset of the input for every point on the closed curve, and cache it for further iterations. This greatly speeds up the program.

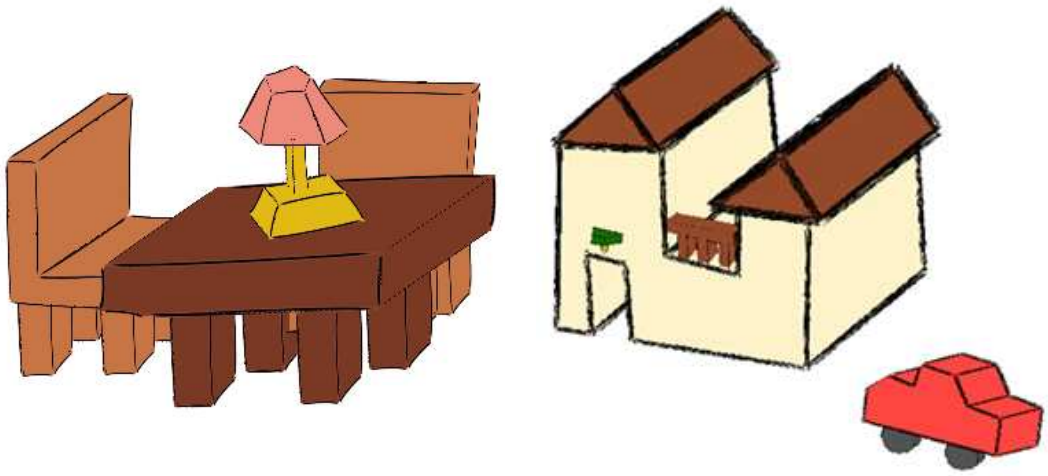
2. The original condition for convergence in the Snake algorithm is the number of points moved in one iteration. This causes points to oscillate if the input data is not smooth, as in the case of strokes. Hence, we eliminate points on the closed curve from further consideration if they are within a certain distance to the input data. Thus more and more points are culled as the iterations proceed, leading to an increase in speed.

3.12 User Evaluation

SMARTPAPER was demonstrated to 10 students of the Department of Architecture at the University of Minnesota. Two graduate students from the Department of Architecture and four computer science graduate students participated in a detailed user evaluation session. The users were briefed about its functionality, gestures and usage and then were asked to perform some pre-defined tasks.

Support for over traced sketching was appreciated by the students, as a lot of them indulged in similar sketching practices. They were asked to construct primitive objects like cubes and prisms and curved objects like cylinders which they did with reasonable ease.

In an effort to confirm the intuitiveness of the gestures in the feedback system, they were not explained to the architecture students beforehand and were asked to do whatever they deemed intuitive. Some gestures were guessed correctly by them, which showed that they were fairly intuitive to users of non-computer science background. Other gestures were learnt easily when they were explained. The users received the seemingly "new" theme of explicit gestures and direct sketching very well, compared to their experience with mouse-based CAD software. The overall theme of the feedback system and the feature of instant shape modification found immediate support among our users. In an attempt to evaluate how all the gestures worked in tandem towards a common goal, the users were asked to make the object shown in Figure 3.12 into a uniform cube. The total time inclusive of the time taken to learn the gestures by both users was approximately 2.5 minutes.



(a) (b)

Figure 3.15: ***Results.*** *(a) A table with lamp (b) A house.*

3.13 Remarks and Future Work

Sketch-based reconstruction is a tough problem with only heuristic solutions. *SMART-PAPER* is an attempt to reconstruct 3D models from sketches by solely relying on the constituent strokes. While we have achieved some success, it is obvious that the success is limited. There exist a large number of sketches for which reconstruction cannot be correctly

performed. This limitation is in part because of its mathematical insolubility.

The choice of supporting only orthographic sketches is strategic. Orthographic projections preserve parallelism and congruence of lines in many cases, thereby providing more reliable 2D image regularities. Other projections like isometric, etc. that preserve such regularities may also be used. Chapter 4 extends this pipeline to perspective sketches in the context of images.

Although tablet devices have been around for a while, developments to support truly intuitive and useful interfaces for rough sketching for conceptual design leave much to be desired. My ongoing work includes several new ideas to design such interfaces and metaphors. Along with tablet devices, I am also interested in leveraging other commodity devices like web cameras and pen-top computers. Commodity cameras have been used in computer vision for several 3D tracking applications. I believe such work and easy access to such camera devices can be used effectively to conceptualize 3D sketching and modeling. I envision a "poor man's desktop VR setup" that can be used for desktop 3D sketching and sculpting applications, with emphasis on affordability over immersion.

Most of the work in computer graphics focuses on rapid prototyping (quickly building a plausible and complete 3D model) rather than truly supporting conceptual design which mixes abstract freeform sketching with ambiguous and incomplete geometry. During my doctoral study, I interacted with students and faculty at the Department of Architecture and researchers from the automotive design industry. I observed how both domains meticulously follow similar underlying rules and practices for sketching and interpreting conceptual designs that appear very abstract to outsiders. In contrast to the aims of much of today's research to facilitate design, conceptual designers need tools that do not attempt to refine their sketches or distract their users from the actual design process by feedback and interaction geared solely towards better reconstruction. There is an overwhelming need to create assisting non-interfering tools that adhere to domain practices, rather than those that either create new metaphors that must be learned or apply CAD-like refined metaphors to conceptual design. I wish to continue working closely with different domains to create such practical and professional sketching applications.

Chapter 4

Image-based Modeling: 3D Models from a Single Image

4.1 The Problem and the Applications

Sketch-based modeling as discussed in the previous chapter concentrated on orthographic sketches. These types of sketches have the advantages of being easier to draw for untrained users, and somewhat easier to reconstruct 3D models from. However many designers are trained to draw in perspective projection because it represents what our visual system sees in the real world. Perspective projections also enable the designer to depict a larger 3D scene more compactly and unambiguously. Therefore there is a need for sketch-based modeling systems to support perspective sketches.

The applications of perspective-sketch-based modeling systems go beyond actual hand-drawn perspective sketches. Images in the form of photographs are the most common examples of perspective projections of 3D models. Reconstructing a 3D model from photographs is a very old and popular problem in computer vision, and lots of diverse solutions with varying degrees of success have been proposed for it. Tracing objects from an image produces a line drawing in perspective projection. Such line drawings could potentially be fed to a sketch-based modeling system to reconstruct a 3D model. This chapter discusses an image-based modeling system that is based on this idea.

Reconstructing 3D models from images has many applications, ranging from entertainment and modeling to surveying. Many methods that concentrate on solving it better or

quicker or both have been proposed over the years[42, 21, 47, 72, 46, 23, 85, 108, 96, 50, 133]. Most of these methods concentrate on producing accurate models from single or multiple images. However, most of these methods also suffer from an "esoteric" interface, i.e. the user is assumed to have some technical knowledge of perspective projections and projective geometry. My work targets ordinary users who are interested in spending a small amount of time on some images and quickly generate a 3D model using a simple and interactive user interface. Thus my work concentrates on a crude but "navigable" 3D model instead of an accurate one, while creating an easier user interface in the process. The resulting prototype creates a sense of "being there" for a user by allowing him/her to fly through (a rough approximation of) the 3D scene that a photograph captures.

4.2 Challenges

Extracting 3D models from images is a very challenging task. As a single method is unlikely to reconstruct various entities contained in photographs like buildings, vegetation, etc., most approaches concentrate on particular classes of entities (faces[69], trees[100], etc.). Observing that many photographs largely capture buildings and other forms of architecture, my system is designed to create 3D models from such casually captured/created images.

Construction of a 3D model from a single image poses the same challenge as that in sketch-based modeling: mathematical insolubility. While projective geometry provides many hints about the captured 3D model in an image, it also lacks several features that orthographic projections preserve. This makes the problem easier in some aspects and challenging in others.

Central to reconstructing a 3D model from a single perspective projection are concepts from projective geometry like vanishing points, vanishing lines, homologies, etc. A set of mutually parallel lines lying on parallel 3D planes intersect at a point called the *vanishing point*. Various sets of such parallel lines (differing in their orientation) form *vanishing lines*. A perspective projection is usually defined by one, two or three different vanishing lines depending on the position and orientation of the camera. Projective geometry identifies

relationships between points on various planes in the 3D environment from their projections. Such relationships between projections of planes are called *homologies*. Homologies are matrices that transform points on one plane to another within the image, without explicitly converting them into 3D coordinates and re-projecting them into the image. Positions of vanishing points, orientations of vanishing lines and homologies derived from known point correspondences form the basis of typical computer-vision methods to reconstruct 3D models from a single image. A detailed overview and analysis of such methods can be found in [42, 21].

The main drawback of these methods is that the user interface is laborious and assumes that the user is knowledgeable about these concepts. However to an untrained user, the concept and logic behind identifying point correspondences and vanishing lines may not be obvious. Moreover an untrained user may not be as motivated to spend time on a single image to reconstruct an accurate 3D model. In fact, an accurate 3D model may not even be necessary for applications like a better image-navigation experience by flying through them, etc. This possibility of creating an easier user interface by compromising on the accuracy of the obtained model is the inspiration behind *Peek-in-the-Pic*, a tool that summarizes my work in image-based modeling. Another motivation is that a solution to this problem is intimately related to that of sketch-based modeling for perspective line drawings.

4.3 Related Work

4.3.1 Image-based Modeling

Image-based modeling reconstructs a scene from multiple photographs taken from various viewpoints by identifying correspondences between points in different images. If a sufficient number of photographs of a scene are available, reconstruction is a mathematically soluble, albeit semi-automatic problem. The *PhotoModeler* program[1] works on multiple images and concentrates on getting precise architectural measurements. The *ARBA3D* program[2] uses only two images, but their absolute positions and correspondence between them have to

[1] http://www.photomodeler.com
[2] http://www.arba3d.com

be manually specified. Software like *SilverEye*[3]are limited to reconstructing geometry from satellite images.

3D models captured in images can be acquired by *instantiation* (associating and aligning pre-defined building blocks with the image), or *reconstruction* (actually reconstructing the captured geometry). Instantiation is the concept behind the *Canoma*[6] program which works only on one image. Instantiation may work well in many situations, but conceptually it offers a very technical user interaction, unlikely to be intuitive for the typical user.

If multiple images are available, photogrammetry and computational stereopsis[78, 117] can be effective. Herman *et al.*[44] attempt to reconstruct 3D models of large scenes using multiple aerial images. Debevec *et al.*[24] interactively build an approximate model of the photographed scene that is then refined by locating image correspondences from multiple images. Reconstructing 3D models from single images, being mathematically insoluble, usually involves making assumptions about the apparent geometry in an image[56], or attempting to fit simple 3D objects (cubes, wedges, etc.) of known topology to a line drawing obtained from the image[101]. Seminal work by Huffman[48] on the notion of labeling contours in an image to infer their geometric nature form the basis of many approaches to reconstruct 3D geometry from 2D inputs[104, 61, 57]. Tour-Into-The-Picture[47] provides a spidery-mesh interface for the user to manually locate vanishing points of the image. Automatic Photo Pop-up[46] automatically creates billboards and "folds" from a single image to create a "pop-up" effect. Both these approaches generate very crude approximations of geometry in the scene, which limits the types of scenes they can navigate and the freedom of navigation itself. Typical vision-based approaches[72, 18, 22] perform reconstruction of a single image by calculating camera parameters, using vanishing lines and point correspondences on each plane. Such a "reconstruct-plane-by-plane" interface may be excessive for our application where accurate geometry is not desired. Also, such methods may not work correctly without a lot of interaction for planes that do not contain parallel edges from which vanishing lines can be computed (like pyramids). Oh *et al.*[85] regard reconstruction as a

[3] http://www.geotango.com/products/silvereye.htm
[6] http://www.canoma.com

Photoshop-like operation termed *depth painting*. They segment the photograph into layers and assign depth coordinates to every pixel of every layer to produce convincing depth images, but at the cost of heavy and time-consuming user interaction. *Peek-in-the-Pic* works at the object level instead of the pixel level.

4.3.2 Reconstruction of Line Drawings

Reconstruction of geometry from freehand 2D sketches is another related area of research and is related to modeling from images in many ways. Although many sketching metaphors exist (Teddy[50] for making rotund objects, SKETCH[133] for design by extrusion), approaches that use actual 2D projections to infer 3D geometry are more relevant to image-based modeling. Reconstruction of 3D models from line drawings is a very old research problem (Barrow *et al.*[10], Marill[76], Leclerc *et al.*[66], Varley[122] are excellent examples). Lipson *et al.*[73] and *SMARTPAPER*[106] formulate reconstruction as an optimization problem by evaluating the 2D input for various 2D image regularities like parallelism and orthogonality of lines and replicating their corresponding 3D properties in the geometry. Moreover while both reconstruct wire frame drawings (with hidden parts drawn), we reconstruct "line drawings" (with hidden parts unavailable). Most image regularities in many of these approaches are robust only if the underlying sketch is orthographic. We extend the optimization formulation to support perspective images and use gestures to provide geometric regularities that could otherwise be obtained implicitly from orthographic images. After the user "traces" out a building, we use this formulation to reconstruct its 3D geometry. Several other interesting approaches use a minimal set of lines (for example, only silhouettes and creases) to produce a plausible set of 3D curved surface models from them using simple mathematical constraints[96]. Kaplan *et al.*[57] propose an iterative user interaction process to progressively refine the constraint space and make the overall problem tractable.

Since we strive to devise a method that requires minimal *and* simple user interaction, *Peek-in-the-Pic* combines different disparate bodies of research in sketch-based geometry reconstruction, vision-based geometry reconstruction and image completion to provide a

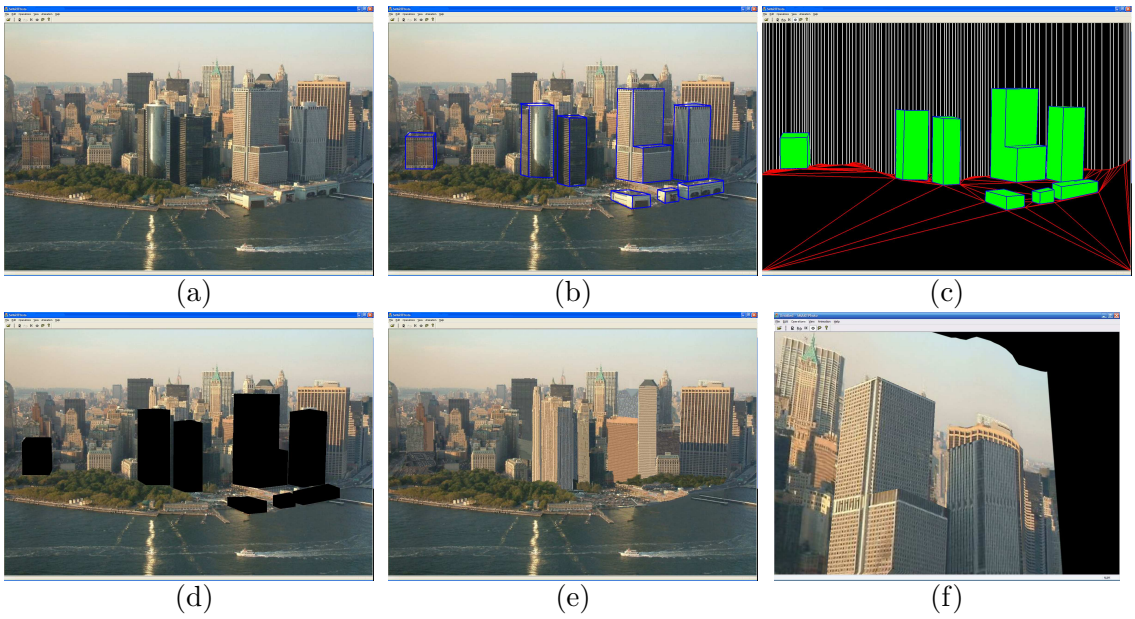


Figure 4.1: ***Partial geometry reconstruction of lower Manhattan, New York City from a single image.*** *(a) the original image. (b) all line drawings made by the user (figure shows all traced lines; actually, one building is traced and reconstructed at a time). (c) eight reconstructed buildings, with the ground relief and the background. (d) the original image with holes to be synthesized. (e) the synthesized image for background and ground geometry. (f) an alternate view of the city.*

complete image-to-navigable-geometry tool. Many operations of our pipeline are fully automatic, with interactive alternatives to improve the automatic results.

4.4 Peek-in-the-Pic: A primer

The general theme of *Peek-in-the-Pic* is "trace and reconstruct". Figure 4.1 illustrates its pipeline. An object is converted into a line drawing by tracing out its edges (Figure 4.1(b) shows all such tracings). Note that only the visible parts of an object can be drawn this way. They are consolidated to form a 2D graph and loops representing the faces of the object are determined. The graph is then analyzed to infer structural constraints on the object to be reconstructed. The user can also specify constraints via simple gestures (Section 4.6.1). The object is then reconstructed by solving an optimization problem (Section 4.6.2) that

considers various structural constraints placed on it and camera parameters obtained in Section 4.5. Hidden parts of the object can be completed automatically or by interactive sketching (Section 4.6.3). The ground geometry is obtained after all desired objects have been reconstructed (Section 4.7, reconstructed geometry shown in Figure 4.1(c)). Holes in the image resulting from construction (Figure 4.1(d)) are filled automatically and can be refined using a simple interface (Section 4.8, Figure 4.1(e)). Finally, textures are mapped on all constructed geometry using the original and the synthesized images for navigation.

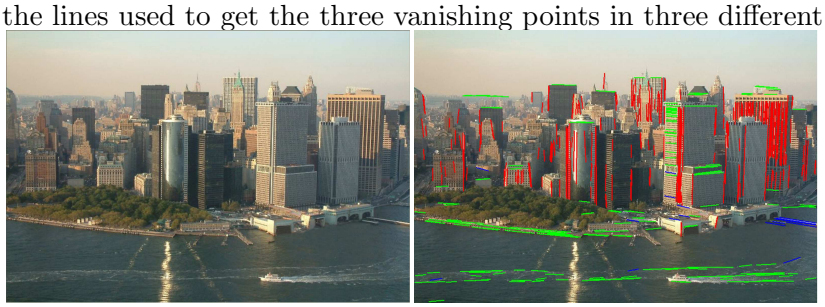
4.5 Camera Calibration

Camera parameters like focal length f of the lens and principal point p_0 are required to correct the perspective distortion in the image before it is reconstructed. Camera calibration by planting a known object in the scene[125] cannot be performed in our case as the photographs we strive to reconstruct are unplanned and casually taken. There are several existing methods to estimate camera parameters from a single image. We use the method by Cipolla *et al.*[18] to calculate camera parameters assuming a simple camera. This method requires knowledge of the vanishing points of an image.

Automatic detection of vanishing points in an image has been a subject of research in computer vision[102, 79, 29]. A common approach is to detect lines in the image and determine points in the image domain where large number of such lines intersect. Intersections can be determined by considering the domain to be the infinite image plane[102] or projecting them as great circles onto a Gaussian sphere and clustering their intersections[29]. We follow the former approach and use a RANSAC algorithm[32] to determine likely vanishing points.

We begin by finding all edges in the image using a Canny edge filter[15] and linking them to find line segments in the image. We find the three vanishing points using the RANSAC algorithm as follows: we randomly choose two lines and determine their point of intersection v. We then count the number of lines in the remaining set that this point is close to (we set the threshold subtended by v and one of the end points of the line onto the

other end point to be 5°). If this count is greater than that in the previous iteration, we select v as a candidate vanishing point. In the next iteration, we randomly pick another pair of lines and proceed similarly. After a sufficient number of iterations (one-third of the total number of detected lines in our implementation) we record the resulting point as a vanishing point and remove all lines that pass through or near this point from the set of lines. We repeat the algorithm twice to retrieve the remaining two vanishing points. If the randomly selected pair of lines is parallel to each other, the point is at infinity and we use their common direction to test the point against all remaining line segments. The figure below shows the lines used to get the three vanishing points in three different colors.



Previous work and our experiments indicate that this algorithm is prone to failures and errors in some cases: spurious edges, Canny thresholds and vanishing points approaching infinity leading to precision errors. When this occurs, we revert to a manual approach: the user selects three pairs of parallel lines that are mutually perpendicular to each other. This has to be done only once per image. These pairs give the three vanishing points.

The three vanishing points constitute a triangle T whose sides are the vanishing lines. Then, p_0 is the orthocenter of T, while f is a function of ${\lambda_1}^2$,${\lambda_2}^2$,${\lambda_3}^2$, where the λ_i's are the areas of triangles subdivided by p_0 in T. A perspective matrix P is constructed from these two parameters, that is then used during optimization. Please refer to [18] for further details.

4.6 Reconstruction of Object Geometry From Perspective Line Drawings

The user now traces[7] the visible edges of a building to be reconstructed from the image, forming a perspective line drawing (since photographs are perspective by nature). All lines are consolidated into a 2D graph G of vertices and edges. Clustering[113] is used for this consolidation. (Visible) faces of the object are then determined using the modified Dijkstra algorithm proposed earlier (Algorithm 2). All vertices of the graph that are on the ground are determined by starting at the lowest vertex of the graph (obviously on the ground). A breadth-first search of the graph is initiated from this vertex. Any edge from the current vertex that makes a small enough angle (say Θ) with the horizontal is assumed to lie on the ground. In our current implementation, we use $\Theta = 35^\circ$. In case the program does not select these vertices correctly, the user can manually specify them.

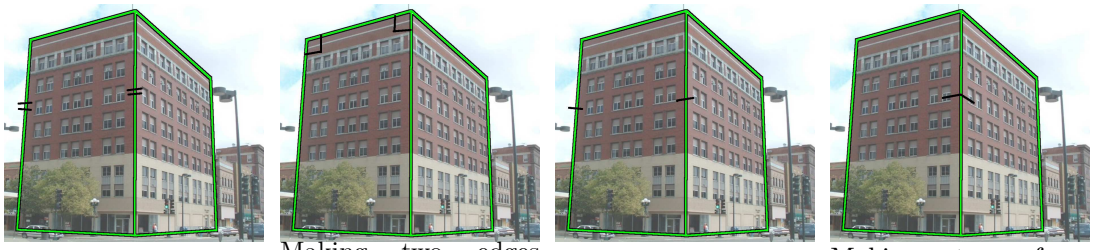
4.6.1 Constraint Specification

Once a 2D graph representing the traced object is compiled, constraints are imposed on its 3D structure that are used during optimization. Although similar constraints have been used to refine crudely approximated 3D geometry from photographs[16], the aim of our constraints is to approximate 3D geometry from 2D inputs. Techniques explained by Lipson *et al.*[73] or *SMARTPAPER* cannot be directly applied to perspective images because these images cannot provide the same 2D regularity cues. For example, two parallel 3D lines are almost never parallel in perspective projection. We assume some constraints *implicitly* and also allow the user to explicitly specify them.

Some general observations can be made about architectural objects: they are attached to the ground, the walls touching the ground *most likely* rise vertically upwards, many edges tend to be parallel or perpendicular to each other, etc. These observations can be *implicitly* used as constraints on the geometry of the object. As the line drawing is in perspective

[7] Our experiments indicated that detecting *only* the visible edges of a building is much more difficult to achieve through automatic edge detection than detecting all straight-line edges, as was needed to automatically infer the vanishing points. This is largely because of a large number of "spurious" edges produced by textures.

projection, only a few constraints can be obtained from image regularities. The user can specify more constraints *explicitly* through gestures. The three vanishing points (and hence the vanishing lines) obtained during camera calibration are used to derive more parallel lines. In any face, if two edges intersect at or near a vanishing line, they are assumed to be parallel to each other. It is also assumed that all edges that have exactly one vertex on the ground are perpendicular to the edges on the ground. This observation is not true for all objects–a notable example being a pyramid. But as these constraints are used as penalties in our optimization, they are not hard constraints.



Making two edges parallel to each other

Making two edges perpendicular to each other

Making two edges congruent to each other

Making two faces perpendicular to each other

Additional geometric constraints can be specified by the use of gestures. We currently support four types of constraints. These constraints can also be specified *after* the object has been reconstructed; the reconstruction algorithm is rerun in this case. Constraints assumed implicitly or imposed by the user are used to derive other constraints. For example, $(e_1 \| e_2) \wedge (e_2 \| e_3) \rightarrow (e_1 \| e_3)$.

4.6.2 Optimization for Geometry Reconstruction

Given a 2D graph G of the traced object and structural constraints, we reconstruct the 3D model that represents the object's projection in the photograph. Specifically, we "inflate" G by assigning suitable depth to each vertex.

This problem finds some common ground with that of reconstructing 3D geometry from 2D sketches in *SMARTPAPER*. However, *SMARTPAPER* assumes orthographic drawings with image regularities that provide most of the constraints used by their optimization process. Most of the assumed image regularities do not exist for perspective line drawings.

We extend *SMARTPAPER*'s pipeline by following a similar framework to reconstruct *perspective* line drawings through constraints explained in the previous section. All geometric constraints are incorporated into the compliance function as penalties.

It must be noted that inflation of the graph G by assigning a Z-coordinate to each vertex does not result in the actual correct geometry, as G originally represents a distorted projection of the object. Therefore, we use the perspective matrix P (Section 4.5) to undistort a candidate 3D graph *before* evaluating all characteristics. The resulting 3D graph projects onto the region occupied by it in the image.

The compliance function that the optimization attempts to minimize is of the form:

$$f = w_i * f_i$$

where $w = [w_i]$ is a weighting factor and f_i are various terms calculated as below. In practice, we use a weighting vector of $(\frac{1}{3}, \frac{2}{3})$ which was obtained empirically. The following notation is used henceforth:

v_i	:	i^{th} vertex of the graph G
f_i	:	i^{th} face in the graph G
$\vec{v_i}$	:	3D vector representing edge e_i
$\hat{v_i}$	:	Normalized 3D vector representing edge e_i
$\|\vec{v_i}\|$	:	Magnitude of vector $\vec{v_i}$
$\hat{n_i}$	:	Unit normal vector of face f_i
$n(G)$	:	Number of vertices in the graph G
$e(G)$	:	Number of edges in the graph G
$f(G)$	:	Number of faces in the graph G

The various terms f_i used are as follows:

1. Face Planarity

This constraint ensures that all faces of the objects are planar. A plane is fit on all vertices on each face f_i and the sum of distances of each vertex from its fit plane comprises this term.

$f_1 = \sum_{i=1}^{f(G)} \sum_{v_j \in face\ F_i} |a_i * x_j + b_i * y_j + c_i * z_j + d_i|$

2. Geometry Constraints

This set of terms is used to evaluate all the geometric constraints compiled earlier.

(a) Parallelism of edges

For all pairs e_i and e_j of edges (total $n_{parallel}$) that are supposed to be parallel to each other,

$t_1 = \frac{\sum_{e_i \| e_j} (1 - |\hat{v_i} \cdot \hat{v_j}|)}{n_{parallel}}$

(b) Perpendicularity of edges

For all pairs e_i and e_j of edges (total n_{perp}) that are supposed to be perpendicular to each other,

$t_2 = \frac{\sum_{e_i \perp e_j} |\hat{v_i} \cdot \hat{v_j}|}{n_{perp}}$

(c) Congruence of edges

For all pairs e_i and e_j of edges (total n_{cong}) that are supposed to be equal in length to each other,

$t_3 = \frac{\sum_{e_i = e_j} \frac{abs(\|\vec{v_i}\| - \|\vec{v_j}\|)}{max(\|\vec{v_i}\|, \|\vec{v_j}\|)}}{n_{cong}}$

(d) Perpendicularity of faces

For all pairs f_i and f_j of faces (total n_{fperp}) that are supposed to be perpendicular to each other,

$t_4 = \frac{\sum_{f_i \perp f_j} |\hat{n_i} \cdot \hat{n_j}|}{n_{fperp}}$

(e) Edge-face perpendicularity

For all pairs e_i and f_j (total n_{efperp})of such that edge e_i is perpendicular to face f_j,

$t_5 = \frac{\sum_{e_i \| f_j} |\hat{v_i} \cdot \hat{n_j}|}{n_{efperp}}$

$f_2 = 0.2 * \sum_{i=1}^{5} t_i$

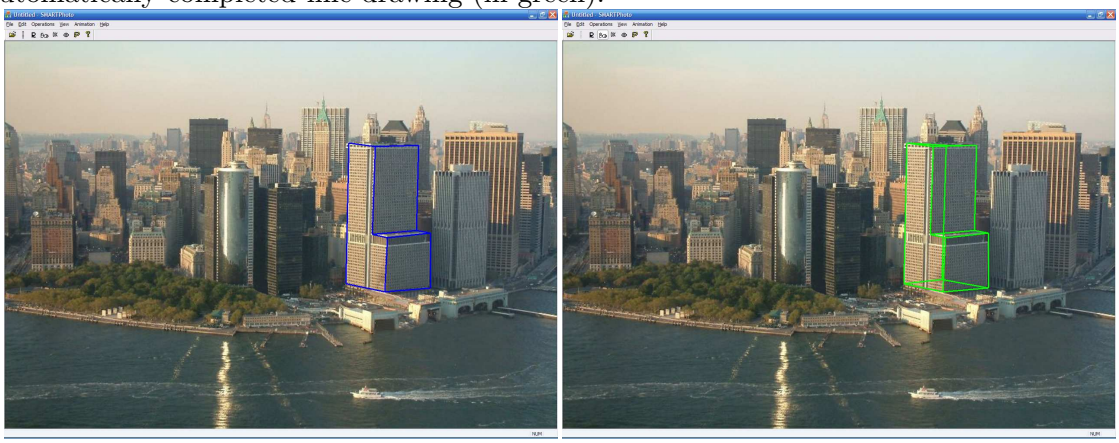
We use Brent's minimization[98] to solve the above optimization problem, as it offers a good tradeoff between speed and accuracy. We use the same layered method explained in Section 3.8.2 for a good initial guess: all vertices on the silhouette form a middle layer, visible vertices not on the silhouette form the front layer and hidden vertices not on the silhouette form the back layer. Although this initial guess works well for closed objects, it tends to fail in case of incomplete objects composed of facades, like Figure 4.4(j-l). In that case we resort to the trivial initial guess (all vertices with the same Z-coordinate) and rely more on implied and gestured hints to reconstruct the 3D model.

4.6.3 Completing Object Geometry

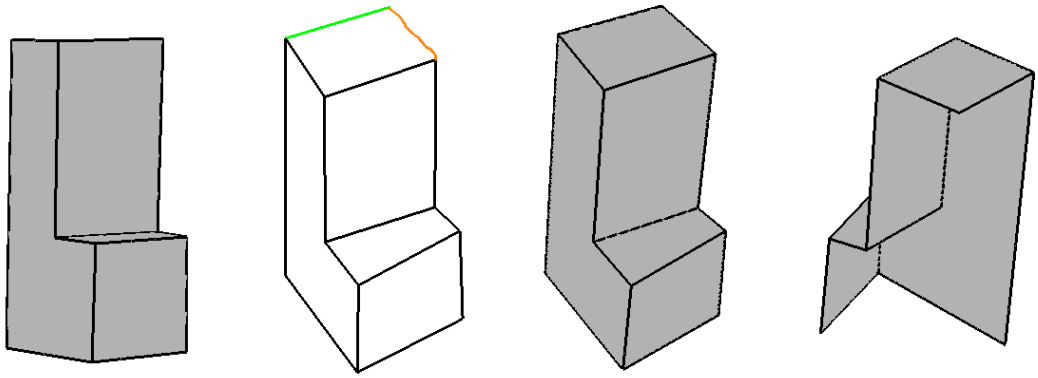
The user can trace only those parts of the objects that are visible in the image. In order to complete the 3D model, hidden edges must be estimated. In general, this is a difficult problem because there can theoretically be an infinite number of configurations for hidden geometry. However, in case of architectural buildings, a reasonable hidden topology can be estimated automatically.

If we assume that all buildings are trihedral, this problem can be solved automatically. Our solution is similar to that offered by Varley *et al.*[123], but for perspective line drawings. We first detect vertices that have degree two (i.e they have an edge missing). Then, we estimate the direction of the missing edge by pairing the visible edges at these vertices with the known vanishing points. Then, we iteratively select two incomplete vertices, and determine the intersection of the two rays along their respective hidden edge directions. In order to prune incorrect pairs of vertices, we constrain all these intersections to lie within

the convex hull of the building in the image (implicitly sketched by the user). Thus we keep eliminating vertices, until we are left with only two vertices (that we simply connect to each other). Below, the figures show the input sketched by the user (in blue) and the automatically completed line drawing (in green).



In case the trihedral assumption does not hold for a building, the user can resort to completing its geometry manually. In this case, although the user could "guess" the invisible edges while tracing an object, our initial experiments indicated that such guesses can affect the overall geometry reconstruction adversely. This is mainly because invisible edges also define vertices on the ground, and so guessing them can adversely affect the resulting ground geometry. Also, it is difficult to draw accurately in perspective and it may be easier to specify these edges from a different view.



The figures above provide an example of the L-shaped building in Figure 4.1(c). The object is rotated and completed progressively. The user sketches strokes for missing edges.

When the program detects a new face formed by them, it reconstructs its 3D geometry. The user can then rotate the partially completed object and continue sketching. This effectively removes the Closed Object assumption in Section 3.6.

It may be observed that this functionality of progressively constructing an object can be used independently as a design tool. In fact, the input image is used only as a guide for setting up the camera (Section 4.5), tracing out buildings (Section 4.6) and texturing the reconstructed buildings (Section 4.8). Thus, 3D models can be progressively reconstructed from perspective sketches once a perspective camera is set up.

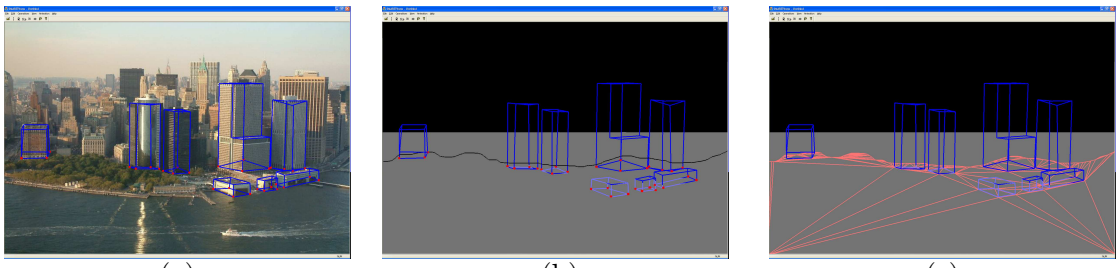
4.7 Ground and Background Geometry

Once all the buildings have been constructed, the ground geometry must be determined to complete the environment. All the edges of the buildings that should be on the ground are heuristically determined as discussed in Section 4.6. Figure 4.2(a) shows these vertices in red. Let the set of such vertices be denoted by S. A least-squares plane P (shown in grey in Figure 4.2(b)) is fit through points in S. It can be seen that P may pass through some objects.

The user then sketches out a pseudo-horizon curve (intersection of the ground and the background, see Figure 4.2(b)). This curve should be drawn just under the buildings in the background, so that the image does not "fold" in the middle of a building. The curve is then projected on P to obtain its 3D coordinates. These projected points are added to S, along with two points on the bottom corners of the screen. All points in S are triangulated in P. These form the ground relief (shown in orange in Figure 4.2(c)). Points of the pseudo-horizon curve are raised up to form the background, as seen in Figure 4.1(c). This completes the geometry of the 3D scene.

4.8 Image Completion

As a building is constructed, it has to be removed from the original image, creating a hole (Figure 4.1(d)). Such textures are usually filled manually by cloning (such as in [85]), a



(a) (b) (c)

Figure 4.2: ***Determination of ground and background geometry.*** *(a) the image with reconstructed objects in wire frame; the vertices marked in red lie on the ground. (b) a mean plane is constructed from these vertices as shown in grey. As it is a mean plane, it passes through some objects (shown in light blue). A horizon curve is sketched by the user. (c) points on the horizon line are projected on the mean plane and are used with the red vertices to get the ground geometry shown as an orange mesh. The background relief is raised from the horizon curve as shown in Figure 4.1(c).*

user-intensive and cumbersome process. Many image completion methods exist in computer graphics and vision literature. Many texture synthesis algorithms exist that complete a given "target region" of an image from source regions contained within the same or different images. Recent research has even led to real-time[71] and controllable[68] texture synthesis. However, texture synthesis primarily works well for amorphous images (flowers, stones, fuzzy background scenery, etc.) and its matching techniques tend to break down when synthesizing well-defined geometric structures like buildings. Image inpainting[12] works well for filling small holes in images, but it is not as effective on large missing regions. Although automatic, both texture synthesis and image inpainting are too slow for the size of images in this paper, mitigating most advantages of the automation.

From our initial experiments with manual cloning, we observed that buildings tend to be filled up using regions immediately surrounding them, not from distant image regions. Intuitively a building must be replaced by the background it occludes, which can be approximated from the image regions adjacent to the building. We use this observation to define the source region for a given hole. We calculate the bounding box of the hole to be filled, and scale it by 25%. We use the region in this scaled bounding box (minus the hole) as the

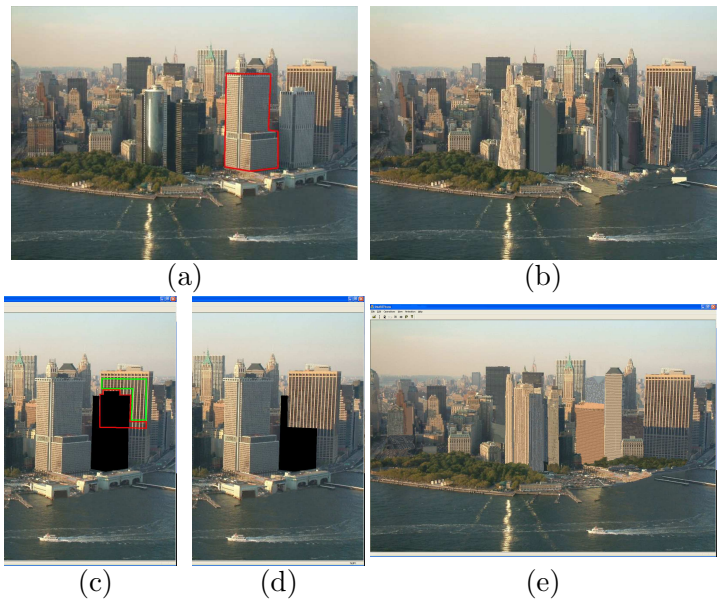


Figure 4.3: ***Hole-filling by texture synthesis.*** *Top row: automatic texture synthesis (a) when a building is sketched by the user, its outline is used as the region for synthesis. The bounding box of this region is scaled up and used as the source region for the texture synthesis. (b) Holes from Figure 4.1(d) filled using this method. Note that although this image looks patchy, the reconstructed buildings overlap these regions for most view points. Row 2: interactive texture synthesis (c) the source and target regions are specified in green and red respectively. (d) the synthesized output for (a). (e) Holes from Figure 4.1(d) filled using this interactive method.*

source for the texture synthesis process. We use the texture synthesis algorithm proposed in [26, 71]. As the hole is automatically derived from the user's sketch, the synthesis is done with no extra input. Row 1 of Figure 4.3 shows how this method produces a synthesized image. Each hole was filled in less than 30 seconds.

Although the image in Row 1 contains very obvious patches and discontinuities, the reconstructed buildings occlude large parts of these patches from most view points, and thus they are not as disturbing. However, because of the limitations of texture synthesis discussed earlier, the results produced may sometimes be too disturbing. In these cases, the user can resort to an interactive process. We take inspiration from the paint-by-numbers interface[41] by augmenting this process with a simple interface to interactively synthesize

these holes (see Row 2 of Figure 4.3). The user marks the source and target regions in the image with poly-lines to start texture synthesis. As the user marks small regions at a time, this synthesis is fast.

4.9 Implementation and Results

All the following results were produced on a desktop system with a 2.6 GHz Pentium Xeon processor and 1 GB RAM, with an external tablet device (Wacom Cintiq PL-550).

Figure 4.4(c) shows a part of Manhattan, New York reconstructed using Figure 4.4(a)[8] as input. Figure 4.4(b) shows how the constructed 3D environment looks from a distant view point. Eight buildings have been reconstructed in this example. The total time from (a) to (c) took about 15 minutes (including interaction), while the image completion took another 10 minutes. The actual optimization time is 2-3 seconds per building.

Figure 4.4(d) shows a drawing of Foshay Tower in Minneapolis, MN (USA)[9] from the 1930s. Figure 4.4(f) shows an alternate view obtained by reconstruction (note the correctly reconstructed pyramidal top of the tower). Three buildings were reconstructed in this view. The total time taken for geometry reconstruction was 5 minutes (including interaction), while the actual optimization time is 2-3 seconds per object.

Figure 4.4(g) shows a drawing of the 1305 Church of Aston-Cantlow, Warwickshire, England[10]. Figure 4.4(i) shows a zoom and change of angle towards the church. The image synthesis in this case was done by cloning, as the background is fairly homogeneous.

Figure 4.4(j) shows a photograph from Liebowitz *et al.* [72] (used with permission from the authors). This photograph is challenging because of some radial camera distortion and because it breaks the layered initial guess assumption made earlier[11] (the corner between the walls and ground is further away from the viewer than the vertices on the silhouette, instead of being nearer to it). The models in (k) and (l) were produced in two steps: first reconstructing the two walls and then reconstructing the roof and ground (1 second each).

[8] Source:http://www.pilotlist.org/balades/manhattan/manhattan.html

[9] Source:http://www.minneapolishistory.com/marriott3.htm

[10] Source:http://www.holoweb.net/ liam/pictures/oldbooks/OldEngland/ pages/1305-Church-of-Aston-Cantlow/

[11] Section 3.8.2

4.10 Remarks and Future Work

Peek-in-the-Pic performs image navigation using a single photograph by amalgamating camera calibration techniques and work done in the area of sketch reconstruction. Since it takes mostly "tracing" operations from the user and works on general photographs taken casually, it is suitable for non-technical users as well. *Peek-in-the-Pic* works best for modern architectural buildings that have polyhedral shapes. Reconstructing more involved architecture like buildings with ancient carvings is more difficult. However, as the final 3D model is textured, artifacts are not noticeable even if a building with details is approximated by simpler geometry, unless one zooms in closely near a building.

Peek-in-the-Pic uses a mathematical approach to reconstructing 3D models from a single image with (minimal) user input. Nevertheless, in order for the optimization framework to be applicable to a large set of images and line drawings, parameters like Θ and those required for clustering line drawings may have to be tuned by the user based on their interpretation. These parameters, and indeed even the reconstruction process can be inferred by machine learning or statistical analysis. Lipson *et al.*[74] and Automatic Photo-Pop[46] use probabilistic learning models to infer 3D geometry from a single 2D input. Similar approaches can be used to *learn* characteristics of building geometries and their correspondences to projections in an image. However some user interaction is unavoidable for any method to work for general cases.

Although tracing out buildings is simple and easy, it may still be a tedious task. Instead, segmenting out buildings and automatically constructing their profiles may reduce user interaction further. Extending methods like *Lazy snapping*[70] to segment buildings automatically is worthy of further study.

My work in *Peek-in-the-Pic* was an attempt to create an explorative experience for images that worked for small sets of photographs of a specific type. However organizing, exploring and searching in large sets of photographs is a much tougher problem that is relevant to any camera user today. The advent of cheap digital cameras has created a broad base of end users who grapple with this problem on a regular basis. It is common to forget

who a particular person or what a particular place in a photograph is. Users struggle with quickly browsing through photographs or presenting them to others, because selecting the appropriate photographs from a large collection is cumbersome. Moreover it is frustrating to remember a photograph but go through thousands of them because one cannot remember where the photograph is stored. Most users are too impatient to manually organize and tag newly taken photographs so that searching for them later is easier.

This problem is, in many ways, the graphical equivalent of organizing text and data files on a computer. However there are several unique challenges. Firstly, interpreting images to organize them better is much more difficult than interpreting text for the same purpose. Secondly, compiling images into a slide show or presentation is a much more common task that users are willing to devote very little time for. Thirdly, processing and maintaining images for search-and-query purposes is both time and data intensive. Fourthly, there are many ways in which a user may want to look for an image (sketching queries by drawing shapes and their relative positions, approximate color distributions, providing (parts of) a similar photograph, or a combination), as supposed to a simple text query.

Available commercial software that work on image collections can be divided into many categories like image editing (Adobe Photoshop, Gimp, etc.), organization and presentation (Google Picasa, etc.), or some combination (Microsoft Digital Image Suite, etc.). A quick trial of these software is enough to realize that no satisfactory solution for image organization and search exists that works on a large collection of arbitrary, untagged photographs in an efficient and automatic way. Research related to image collections is very popular because of the above reasons. Recently, there have been promising research developments like Microsoft Research's *PhotoTourism*[114] and *PhotoSynth*[2] that work on a variant of this problem: photographs that densely sample a place or an event. Many solutions that work with particular sets of input metaphors (sketching, image similarity, etc.) exist. However a holistic solution that works efficiently in space and time and correctly on large image collections using different query metaphors is still elusive. The focus of my future work will be to develop algorithms to organize, search and present personal image collections that sparsely sample events, places and people.

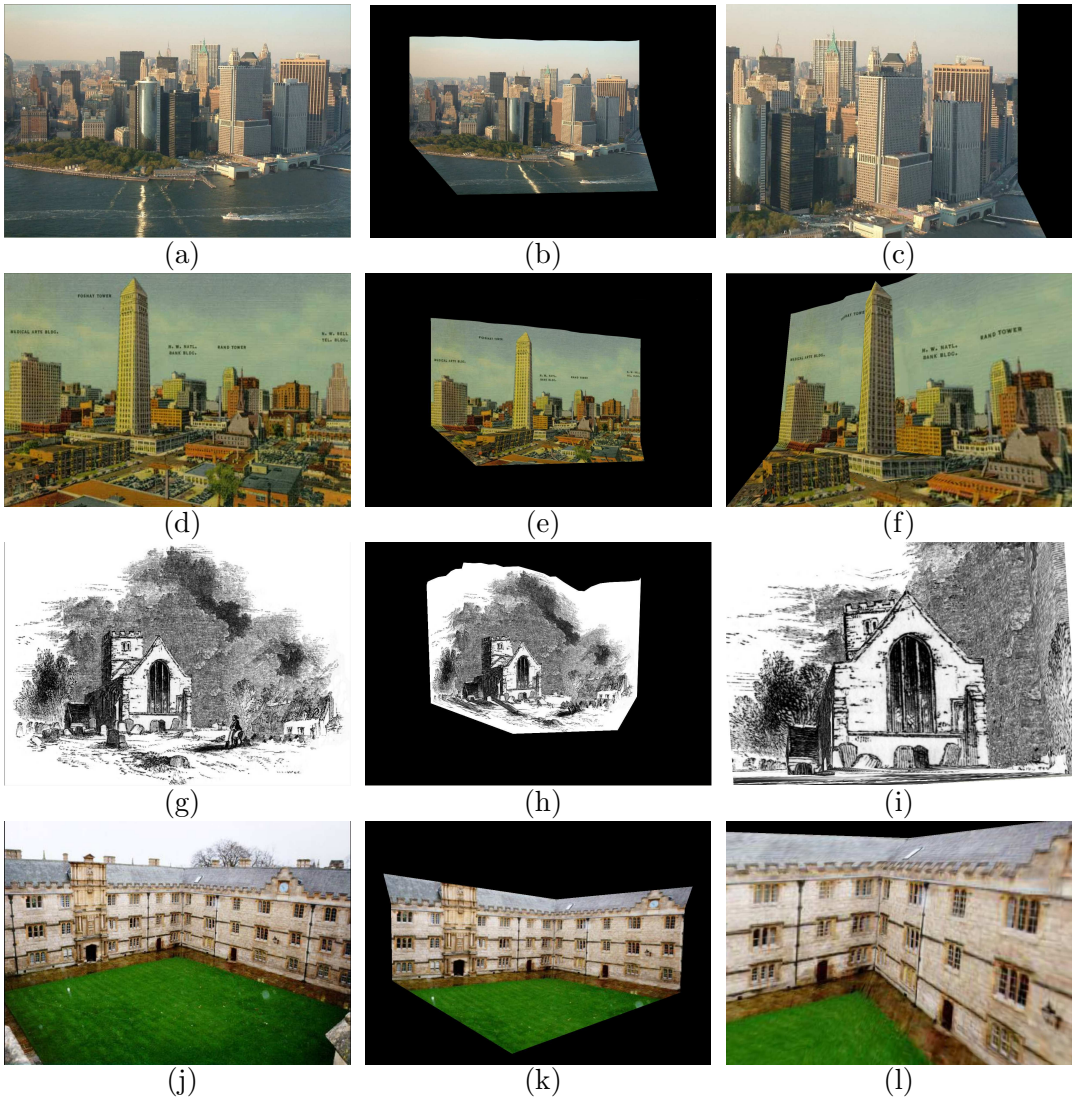


Figure 4.4: ***Results.*** *(a) photograph of Manhattan, NY (b) view from a distant point. (c) a unique view of the reconstructed 3D scene. (d) painting of 1930s downtown Minneapolis, MN. (e) view from a distant point. (f) a unique view of the reconstructed 3D scene. (g) painting of 1305 church (h) view from a distant point. (i) a unique view of the reconstructed 3D scene. (j) the original photograph of Merton College, Oxford, taken from [72]. (k) view from a distant point. (l) a unique close-up view of the 3D reconstructed scene.*

Chapter 5

Sketch-based Inverse Lighting: using sketches to design lighting

While the previous two chapters of this dissertation concentrate on creating 3D models from sketches, this chapter and the next concentrate on other applications directly related to or greatly aided by hand-drawn sketches.

5.1 The Problem and the Applications

Lighting is an integral aspect of any application or process that concerns the appearance of objects. Lighting in concert halls, meeting rooms, living rooms, etc. are specifically designed to suit a particular purpose. In any modeling in computer graphics, lighting is used to give models a desired look. Lighting is a critical tool in cinematography and other performing arts. But how is lighting designed?

In the real world, artists and technicians use their knowledge of the subjective and interpretative aspects of lighting to design it for a particular place and purpose. Animators and artists fulfill a similar role in motion pictures and animations. Irrespective of the application, the designer has a specific "look" in his/her mind which must be realized using lights of various kinds and colors. Technically, this problem of translating desired lighting effects into actual lighting configurations is termed *inverse-lighting*.

One of the most important aspects of solving inverse lighting problem computationally is the metaphor used to communicate "desired lighting effects". Traditionally artists

and designers express lighting effects similar to contours and geometry–via sketching and painting. The most expressive and intricate lighting effects have been illustrated through sketching, shading and painting by artists. Thus, specifying desired lighting effects to a computer using sketched input seems a natural extension of what artists have been always doing. This chapter deals specifically with solving a simplified version of the inverse lighting problem using interactive sketched input. The direct applications of our prototype are confined to model visualization in modeling systems that support basic lighting but no satisfactory way of designing it, and applications like ray tracing where designing lighting interactively is a challenge. However we propose an optimization framework that can be scaled to accommodate complex lighting models.

5.2 Challenges

The problem of inverse lighting by nature is complex even if the most basic lighting model is chosen. This is mainly because the relationship between various lighting effects like highlights, colors, intensities, etc. and lighting parameters like positions, angles, directions, etc. is not always obvious. While relationships between lighting parameters and highlights can be expressed mathematically, that between parameters and shadows cannot always be. Shadows add great complexity to the problem because of their dependence not only on the lighting parameters but also the nature of models (occluders).

In the absence of a more sophisticated work-flow, a typical lighting design session may proceed as follows: the user places some lights around the model randomly and then tweaks their positions and other parameters to achieve the desired look in largely a trial-and-error fashion. The user tries to decide where to place lights, where to point them, how to change their colors and intensities and whether to add/delete lights simultaneously. 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 (cause) 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. A good work-flow for lighting design would be goal-oriented, alleviating the user from "thinking technically" about the physics of lighting. This chapter realizes this goal for a simplified lighting model.

In terms of sheer speed (for interaction), the problem of inverse lighting is at least as complex as that of rendering itself. If the rendering model is complex, designing lights for the model becomes even more difficult. The lighting design process is interactive by nature, and hence an interactive solution to computational inverse lighting is highly desired. The use of sketches as an input metaphor for inverse lighting, while very suitable and natural, also raises the expectation of the overall system being interactive. Secondly, as will be obvious in later sections, formulating this problem mathematically is relatively simpler than solving it correctly and interactively because of a vast solution space. Lastly, as the user has complete control over the sketched input, nothing prevents him/her from providing *infeasible* lighting effects[1].

As the next section summarizes, many solutions for inverse lighting have been proposed over the years. 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. This work addresses 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.

This chapter of this dissertation presents my work in this area in the form of *Crayon Lighting*, a prototypical tool that solves the sketch-based inverse-lighting problem efficiently

[1]The definition of *infeasibility* itself depends on the nature of the lighting model and the actual computational work-flow of solving the inverse lighting problem.

for a simple point-light-based model by utilizing the computational power of the Graphics Processing Unit (GPU).

5.3 Related Work

Inverse lighting is a sought-after area of research. This section shortly summarizes work done in this area. Patow *et al.*[87] provide a more comprehensive survey.

Inverse lighting has been tackled in various contexts ranging from interior design to cinematography. Barzel[11] proposes a very general lighting model with a lot of degrees of freedom in the context of cinematographic lighting. Radiooptimization[62] and Painting-with-Light[103] 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.

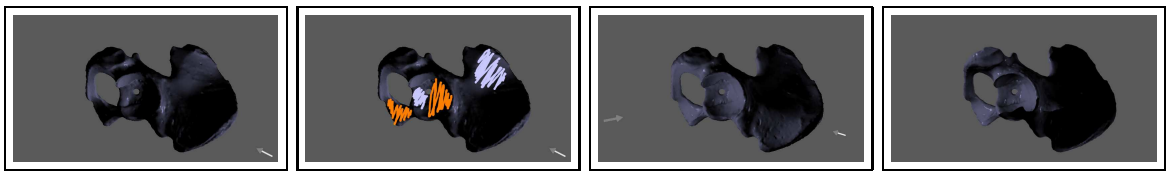
Various user models have been attempted in the context of lighting design. Kristensen *et al.*[64] concentrate on real-time rendering so that the user may interactively place lights and examine their visual effects. Automatic techniques to infer lighting are based on analysis of the scene geometry[105]. In contrast, many approaches allow the user to specify desired lighting effects in some intuitive manner and design lighting that produces them. *Crayon Lighting* belongs to this category. A popular approach is to directly "splatter" the model with desired colors[103, 95]. Poulin *et al.*[93, 94] 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[77, 90, 89] 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[62]) while others are more generalized. Gumhold[40] formulates it as an entropy minimization. Costa *et al.*[20] 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 Design-Galleries[77] where the user selects between various configurations through an interface, and Light Collages[67] which formulate the problem as an efficient greedy problem that infers possibly globally inconsistent lighting. Shacked *et al.*[105] 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 pre-computed 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[89], 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.

5.4 Crayon Lighting: A Primer

Crayon Lighting can be used to design lighting that uses local point-light based illumination



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

Figure 5.1: ***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 ray-traced image of the same model under the same lighting conditions, with shadowing effects.*

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. The current prototype of *Crayon Lighting* 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. The underlying optimization framework is general enough to be applied to a wide range of lighting models, although all results in this chapter use only one such model. Interactivity is achieved 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 5.1(a)). The user uses an orange highlighting pen to sketch highlights and a blue darkening pen for darkening parts of geometry by contrast (Figure 5.1(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 5.6.4). When the optimized lighting is presented(Figure 5.1(c)), 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(Figure 5.1(d)).

5.5 Sketching Interface

A wide range of user interfaces can be used in the context of inverse lighting systems; Painting-with-light[103] and the work done by Poulin *et al.*[94] 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.*[6] 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 as an ID buffer[63] to identify triangles may be faster, our volumetric grid is useful for other purposes as well, as explained in Section 5.6.3.

5.6 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.

5.6.1 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.

5.6.2 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

$$f(X) = w_1 * f_{change} + w_2 * f_{retain} + w_3 * f_{barrier} \tag{5.6.1}$$

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 shown here. We consider seven lighting parameters as degrees of freedom (DOFs): position (in polar coordinates, explained later in Section 5.6.3) (θ, ϕ), direction (in polar coordinates)$(\theta_{dir}, \phi_{dir})$, diffuse (k_d) and specular (k_s) intensities and spot angle ψ.

$$\begin{aligned} f_{change} &= \frac{\sum_{v_i \in V_h} |c_i - t_i|}{|V_h|} \\ f_{retain} &= \frac{\sum_{v_i \in V_{other}} |c_i - t_i|}{|V_{other}|} \\ f_{barrier} &= \sum_{lights L_i} max(0, -I(L_i)) + max(0, I(L_i) - 1) \end{aligned}$$

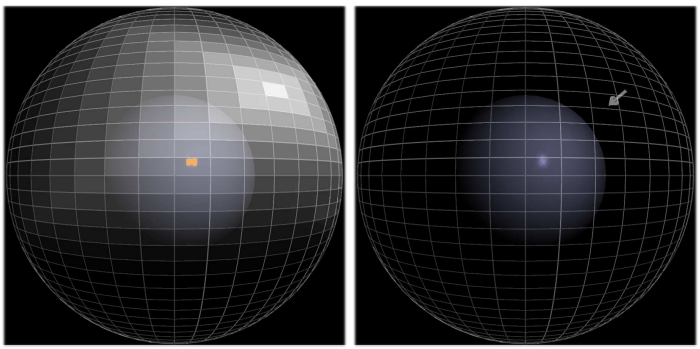
f_{change} minimizes the sum of differences between the candidate and target intensities of vertices in V_h. This term is similar to that in Lighting-with-paint[89]. f_{retain} minimizes the change of intensities of vertices in V_{other}. $f_{barrier}$ prevents light intensities from falling below 0 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 (and hence none of the terms in $f(X)$) 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 framework 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 towards converging to the correct minima. We use this method to also automatically add or delete lights from the system. 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.

5.6.3 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 5.5). 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.

5.6.4 Solving the optimization

In order to achieve automatic light addition/deletion and to keep the problem tractable, we perform the optimization in several stages (Figure 5.2). At any stage, if $f(X)$ falls below a certain threshold $T_{f(X)}$ (0.2 for all the results shown here), the system declares success. In

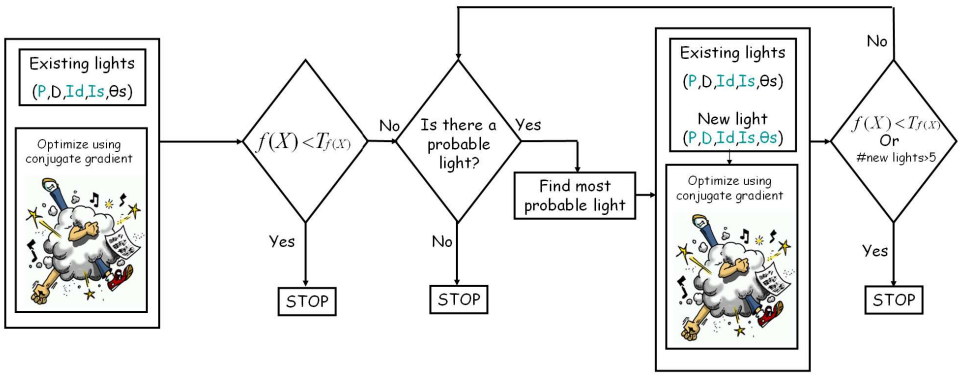


Figure 5.2: ***Flow chart describing the optimization procedure in Crayon Lighting***

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.

5.7 Implementation and System Features

Crayon Lighting 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.

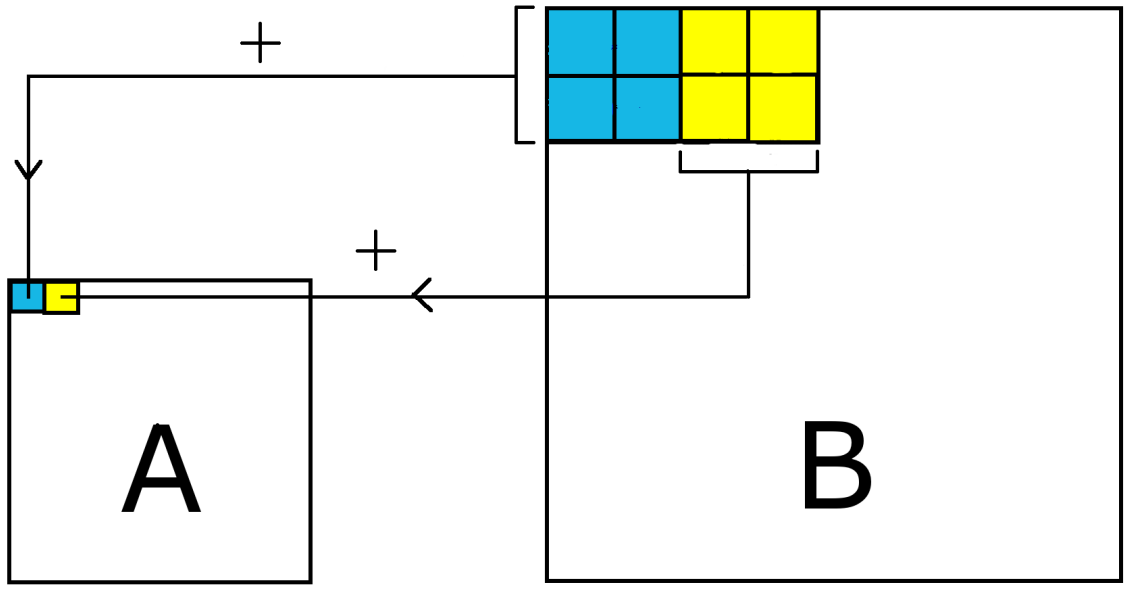
5.7.1 Calculating the minimization function

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.*[128], 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}, $V = V_{other}$.

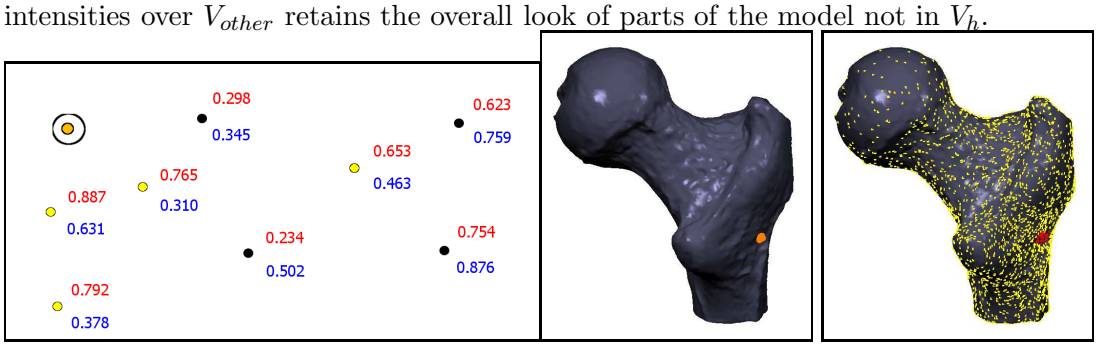


To calculate f_{change} and f_{retain} we have to find the sum of all texels in its difference texture. We use a multi-pass approach to achieve this. We create two float buffers and render DT in one of them. We use them alternately as the rendering context and input texture in various passes as follows: 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.

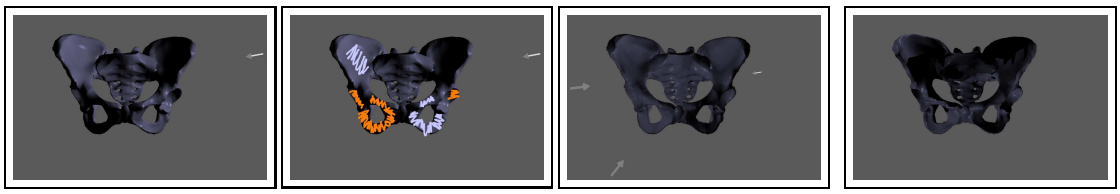
5.7.2 Sampling vertices

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) (b) (c)

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



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

Figure 5.3: ***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.*

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[82], volume rendering[131] and point-based rendering[99].

5.7.3 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.

5.8 Results

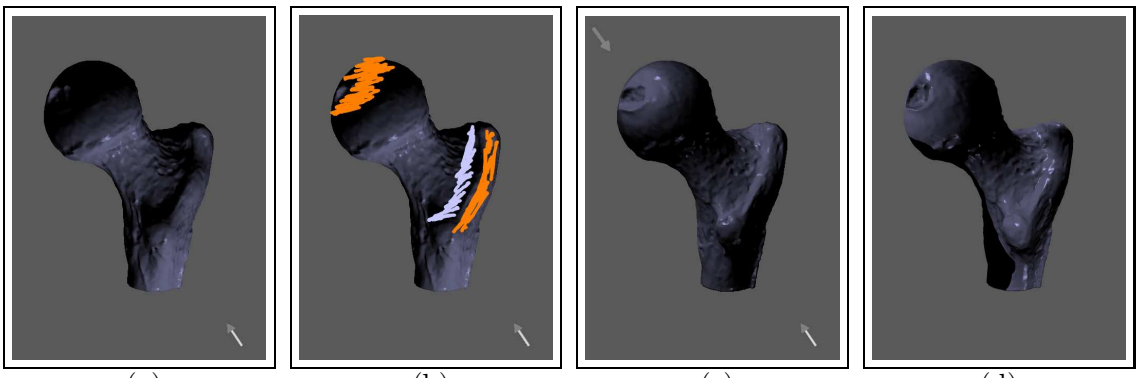
Figures 5.1, 5.3, 5.4, 5.5 and 5.6 show results on some models. All of these results were produced by a computer science professional who does not primarily work in computer graphics. In general several inputs could be required progressively to arrive at the desired result. Each figure shows lights in the form of arrows; the base of the arrow indicates its position and the direction shows its direction. Figure 5.1(a) shows an example of the medical model of a hip with 40,000 triangles, lit with default lighting. Figure 5.1(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 5.1(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 5.1(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.

Figure 5.3(a) shows a pelvis model[2] with a slightly contrived user input. The aim is to switch the opposing lighting conditions on the two lower cavities of the pelvis. Figure 5.3(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 Figure 5.3(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 5.4 shows results on the ball joint model having 35,000 triangles. The goal of the input in Figure 5.4(a) is to highlight the ball region. The result of this input is shown in Figure 5.4(b-c). This result took 4.16 seconds.

Figure 5.5[3] shows how lights can be obtained by progressively transforming a model and sketching on it. Figure 5.5(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 5.5(b) to obtain the lighting of Figure 5.5(c-d). Again, the object is rotated

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



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

Figure 5.4: ***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).*

and some more hints are sketched (Figure 5.5(e)) to increase the contrast, as shown in Figure 5.5(f-h).

Figure 5.6 exemplifies an interesting application of our system. Figure 5.6(a) shows a pre-existing image of an NPR rendering of the same model that shows lit and darkened regions[4]. 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 5.6(b) shows one such input). Figures 5.6(c-d) show the result in which most of the effects in Figure 5.6(b) have been reproduced. Thus, our system can be used to *reverse-engineer* lighting of a model, given an example image of its rendering.

5.9 Remarks 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. I envision this tool being used by modelers or researchers in computer graphics and visualization as a simple tool

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

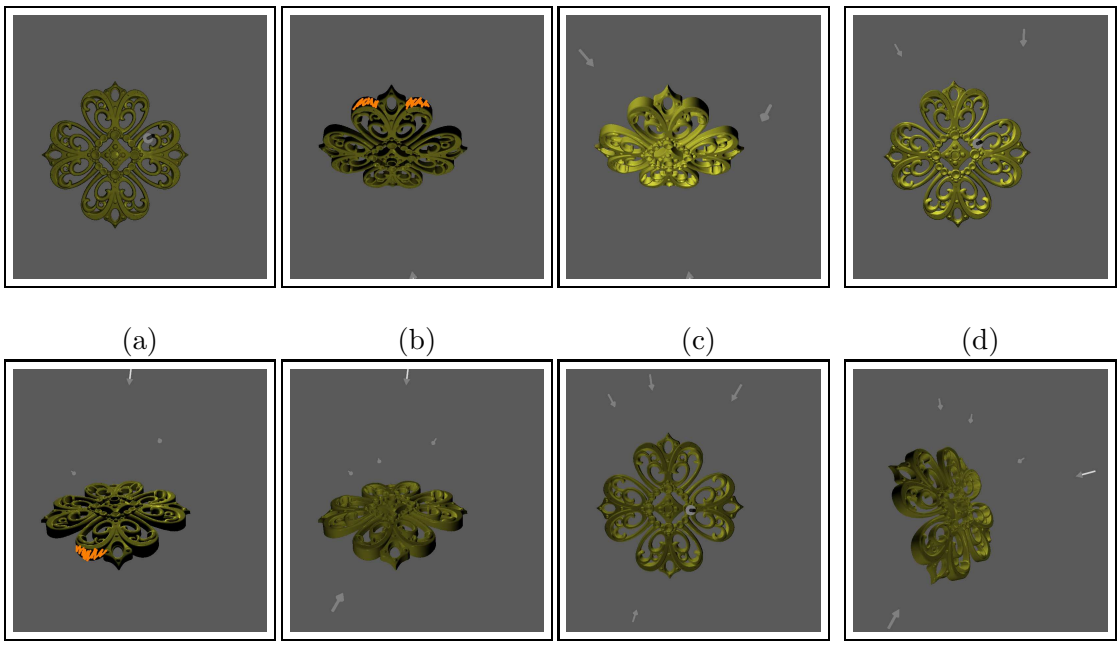
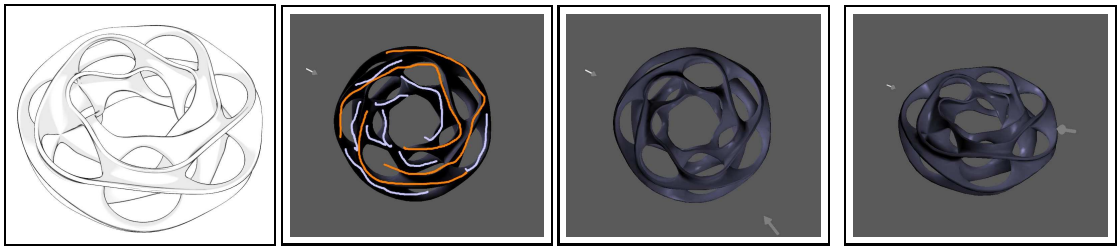


Figure 5.5: ***Progressive inverse lighting.*** *Figures show a filigree model having 177,000 triangles. The aim is to contrast opposite faces with light and darkness. (a) the model with default lighting. (b) the model is rotated and some input is given. (c) result of input from (b). (d) the lighting 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.*

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. *Crayon Lighting* is easy to use, works at interactive rates and restricts user input to drawing highlights and shadows.

The following are some practical issues with *Crayon Lighting*:

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,



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

Figure 5.6: ***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.*

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 5.6.1) and not the quality of the final result. If S_{lights} (Section 5.6.3) 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.

Crayon Lighting 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.

In the future, I would like to investigate 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 sketch-based inverse lighting like Illumination Brush[86], interactive global illumination[81, 43] targeted towards real-time lighting design[64]. Many of these alleviate the costs of performing global illumination by using approximation methods like wavelets or smart matrix approximations. My interest lies in investigating whether such methods that make rendering efficient can also be leveraged to design lighting based on intuitive sketch-based input.

Specifically, consider the approach of compressing a light field using wavelets[81]. Such a formulation expresses the intensity of a vertex as the dot product of two vectors: a row of the transport matrix and all possible light positions on a surrounding cube in terms of their wavelet coefficients (the light vector). Because of the sparsity of the transport matrix, the two vectors are also sparse. Physically, the wavelet coefficients of the transport vector correspond to the contributions of various square regions that are created when each side of the cube is recursively sub-divided. Thus the coefficients provide a per-quadrant distribution of light as one descends the hierarchy formed by the sub-division. The goal of inverse lighting is to predict the corresponding coefficients of the light vector. The physical interpretation of the coefficients may be used to design a hierarchical optimization that converges quickly to the desired local minimum.

Chapter 6

Sketches as Non-Photorealistic Renditions

6.1 The Problem and the Applications

Sketching is a natural activity for us. We use sketches every day and everywhere–an artist sketches abstract illustrations, a designer sketches initial design ideas, etc. Research in sketch-based computer graphics often proceeds orthogonal to this aspect of sketches. The role of sketches in such research remains passive–a means to an end. Traditionally, sketched inputs are either parsed to identify domain-specific entities[65, 34], interpreted to create the implied, more concrete form of data[133, 39] or used to annotate existing data[55]. Thus, most work in sketch-based systems (including that in the previous chapters of this dissertation) is geared towards interpreting or refining sketches.

While this goal is useful in many applications and has its challenges, it is best for a lot of applications involving sketches to leave interpretation to the creator. For an artist, sketches are abstract means of expression, that do not necessarily have refined forms in geometry. The initial conceptual sketches made by a designer are often too abstract for any algorithmic interpretation. This is reflected in the fact that many conceptual designers do not approve of computer programs changing their original sketches into refined forms because that is considered detrimental to the design process itself. Thus, computer-assisted design in such scenarios involves rendering these sketches with *fidelity*, i.e. satisfying aesthetic requirements but retaining their overall shape and abstract nature. This creates the

need to recognize **lines and curves** as active **rendering primitives**.

Such a characterization is in many ways, complementary to the field of non-photorealistic rendering (NPR). NPR deals with creating artistic and abstract renditions from various forms of 2D/3D geometry. Many of the challenges concerned with such sketch-based applications are also shared by NPR, with possibly one important addition: interactivity. While most forms of NPR are generated from static geometric models, sketch-based applications are inherently dynamic and interactive. Strokes are added, deleted and transformed on-the-fly as the renditions are viewed. The overall problem is to be able to create aesthetic, clutter-reducing, smoothly transforming and efficient static (images) and animated renditions in an environment where strokes are added, deleted and transformed arbitrarily.

There are many applications for such a rendering pipeline. First, such a pipeline can act as a front-end for many sketch-based applications[53, 59, 25, 51]. Secondly, such a pipeline addresses an important problem in NPR: creating efficient, temporally consistent, aesthetically pleasing static and animated renditions in an interactive and dynamic setting. Thus it can be used to render different types of geometric models non-photorealistically.

6.2 Challenges

Like its applications, this problem shares most of its challenges with non-photorealistic rendering.

Simplifying and rendering sketches (line drawings in general) on-the-fly obtained from 2D/3D data in general presents two unique challenges. First, as line data may be subject to an arbitrary sequence of rigid, deforming and projective (for 3D lines) transformations, simplifying them efficiently on-the-fly is difficult. The second challenge concerns the issue of visibility. Many techniques to create line drawings from 3D geometry in the form of meshes[115, 37, 127, 52, 45], points[130] and volumes[13] have been proposed in the past. Most previous work use attributes of the underlying surface geometry to address visibility. However in many cases, using an underlying surface to determine visibility between strokes may be inappropriate (e.g. freehand strokes drawn on planes, as in Mental Canvas[25]

and 3D6B[53]) or even impossible (interactively created wire frame CAD models). Thus appropriate visibility cues must be generated to disambiguate the depth between strokes.

There are two generic principles governing the process of line simplification: *proximity* (strokes near each other should affect each other) and *continuity* (the integrity of each stroke must be preserved) as described by Barla *et al.*[9]. To adhere to these principles, some form of a level-of-detail hierarchy of strokes can be built based on proximity so that continuity is maintained during rendering and animation. For static 2D line models (from vectorized images or previously drawn digital sketches), such a hierarchy can be generated off-line and only once. To support progressive sketching and editing sessions or to render 3D line models, this hierarchy *must* be created dynamically and interactively whenever strokes move on the screen (e.g. view point changes) or are added/deleted. Detecting *proximity* and simplifying to maintain *continuity* in such a dynamic setting is challenging.

6.3 Related Work

The issue of line simplification that is central to this work has been addressed in several ways in NPR. Strokes are simplified in a line drawing to maintain tone and overall shape and possibly for rendering efficiency. Tone is quantified by measuring local screen-space density[37], often for static line drawings[127]. Simplification is typically done by creating a level-of-detail (LOD) hierarchy of lines. Lines are prioritized and rendered according to this hierarchy. Creation of these levels of detail under various settings is the subject of many papers. Tonal art maps are used by Praun *et al.*[97] as an image-based method to create LODs in hatching. In the field of illustration, Winkenbach *et al.*[129] introduce the concept of indication to prioritize lines in textures to achieve control over density, albeit semi-automatically. With a sketch drawn in real time as its input, WYSIWYG-NPR[55] takes a user-centric approach by relying on manual specification of LODs when they cannot be extracted automatically. All these approaches simplify strokes using an accept/reject scheme, i.e. removing strokes to simplify the drawing. An interesting approach by Cole *et al.*[19] uses *priority buffers* to modulate line density for localized simplification effects.

Their focus however, is on user-assisted customization of line drawings created from static models, and hence may not be suitable for automatically simplifying interactively sketched line drawings.

This work is partly inspired by that of Barla *et al.*[9] who merge strokes by using screen-space proximity and color-based clustering. However there are three primary differences between their work and ours. First, their hierarchy can be created only once for a given drawing (possibly off-line) as they assume that drawings can only be zoomed in or out. We create and manage a time-coherent hierarchy on-the-fly to allow an arbitrary sequence of **2D/3D** transformations in an interactive and dynamic setting. Secondly, they rely on a manual classification of strokes as contour and hatching and employ different simplification methods for them. We present a unified and automatic simplification strategy, whose advantages are discussed in Sections 6.5.2 and 6.7. Lastly, a change in their input parameters requires the hierarchy to be completely re-calculated (their ϵ-lines and ϵ-groups change with ϵ). Our implementation does not suffer from this restriction. This is an advantage because such parameters are model-dependent and often have to be determined interactively by trial-and-error.

6.4 Overview

Stroke Proximity: Given a set of strokes projected on the screen, we first determine which strokes are (at least partially) "near enough" to other strokes to affect their appearance. We define a parameter δ as the maximum distance between two strokes for them to affect each other. This parameter can be changed interactively. Section 6.5.1 explains stroke proximity in detail. Proximity calculations are also used to generate visibility cues to mitigate depth ambiguity and visual clutter. This is explained in Section 6.5.4.

Stroke Pairing And Simplification: To disambiguate multiple interactions (a single stroke can affect and be affected by many strokes), we pair every stroke with one other stroke that it affects the most, according to proximity, color, local gradient and extent of overlap as a percentage ρ. This parameter can also be changed interactively. We merge two

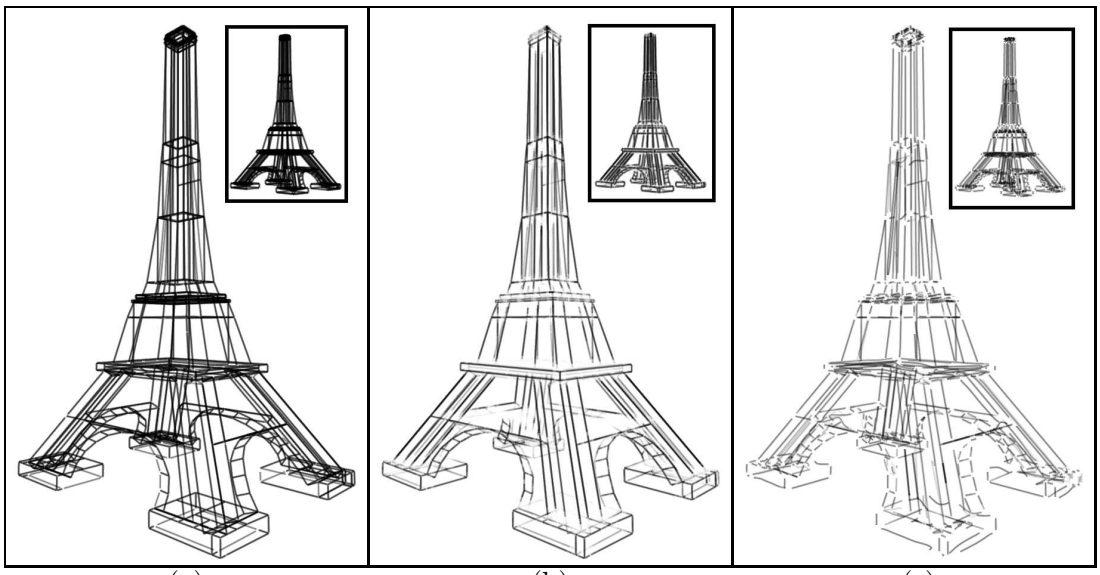


Figure 6.1: ***Line drawings of the Eiffel Tower.*** *Each column shows the Eiffel Tower zoomed out to the same level (thumbnail) and the thumbnail magnified. (a) the original wireframe model. (b) the tower rendered using only visibility cues that reduce some of the clutter. Without simplification, this figure is rendered at a very low frame rate. (c) the tower rendered after simplification and with visibility cues. Simplification further reduces the clutter when zoomed out to such a level in the thumbnail in (c) to retain the overall shape of the tower but hide the detailed truss structures.*

paired strokes into a single stroke. Section 6.5.2 explains pairing and simplification.

Hierarchy Maintenance: Stroke simplification results in a dynamic binary tree hierarchy. Whenever points move on the screen, this hierarchy must be updated and an appropriate level in it must be chosen for rendering. Section 6.5.3 discusses how this hierarchy is maintained.

Rendering: Every point in the sampled stroke is rendered as an oriented, alpha-textured quad with the stroke's thickness and color. Each point's opacity modulates that of its texture to produce the final result. The texture can be changed to produce various styles.

6.5 Simplification and Rendering

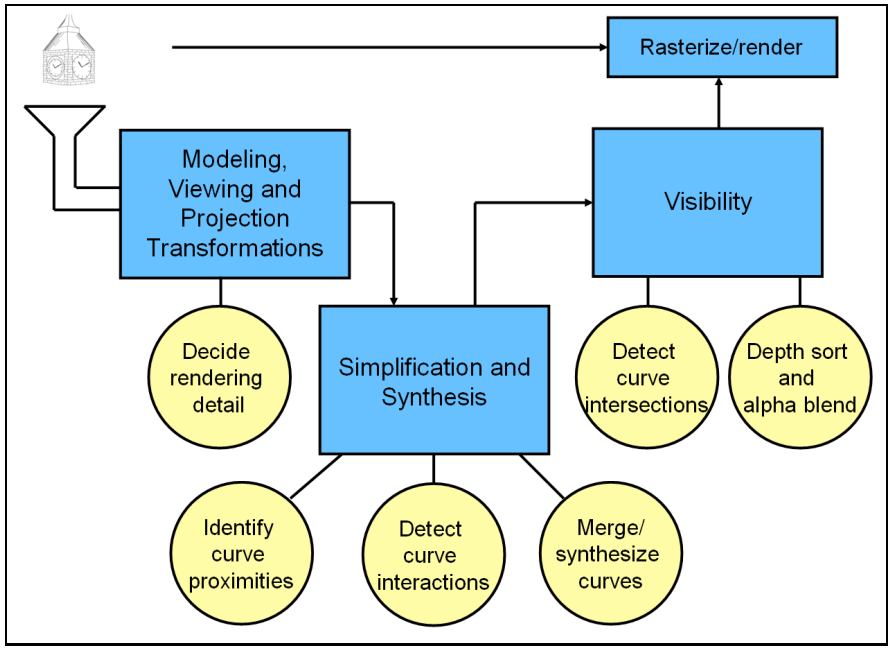


Figure 6.2: ***Rendering pipeline for strokes.***

6.5.1 Proximity

Line simplification can be thought of as a localized process, where strokes are affected only by other strokes near them. If a shape-defining feature like a contour is specified using many small strokes (a common sketching style), then proximity groups such small strokes together so that they may be approximated by a single long stroke. A local group of hatching strokes may be approximated by fewer strokes to maintain stroke density as the user zooms out. Thus, proximity is an effective measure of identifying groups of strokes that may be approximated by a different set of strokes.

To determine which strokes to simplify, one must determine which strokes are near each other in screen space. Data structures employed for this purpose must support adding and removing strokes efficiently. Moreover, as the strokes move incrementally on the screen, proximity should be re-calculated quickly to maintain interactive rendering rates.

Given a set of curves, it is difficult to efficiently determine proximity as they move. This problem can be solved in a variety of ways:

Representing strokes as a collection

An obvious solution would be to divide the 3D space using an octree. Each cell of the octree would store (parts of) all the curves that pass through it. Thus, a single stroke is divided into multiple cells that are arranged hierarchically.

The octree can be supplemented with additional data about each cell's siblings to make the proximity query efficient. However the octree responds poorly to moving strokes. During every frame, all the strokes would need to be re-distributed within the octree. Thus, although an octree would produce an efficient proximity query, it would not be suitable for our dynamic and interactive setting.

Representing a stroke individually

An alternative would be to represent each stroke separately, and then check which strokes are near each other. This may be viewed as a "collision detection" test between strokes under movement. Collision detection is a very popular problem in computational geometry, and many solutions exist for it. However most of these methods apply to objects that can be represented as points.

A strip-tree[8, 31] is a hierarchical data structure to represent a curve. In a strip-tree representation, every curve is recursively sub-divided along its arc length, until arcs of sufficiently small length are obtained. Thus the leaves of a strip-tree are line segments that represent the curve at a particular resolution. This representation has the advantage that a stroke may also *self-simplify*, i.e. reduce in resolution as its screen size becomes smaller.

Collision detection is a standard operation on a strip tree because it can be determined hierarchically[132]. This is based on the (loose) assumption that if two nodes of two strip trees do not collide, none of their children collide with each other either[1]. Thus it is efficient to check whether two strip-trees collide with each other. Another advantage is that since

[1]This is not always true as a node in a strip tree is not constrained to fully contain both its children. Due to this property, the collision query will realistically be more expensive than logarithmic time

every stroke is represented as a single tree, updating strip-trees when strokes move simply amounts to transforming the whole tree.

Unfortunately, in the absence of a "higher" data structure, all pairs of strip trees will have to be exhaustively checked for collision detection in order to determine proximity. Another data structure that manages this "strip forest" more efficiently potentially has the same drawbacks as the octree above. Thus although the strip tree representation is efficient under addition, deletion and movement, its proximity query is inefficient.

Divide-and-Conquer: Strokes ↔ Points

We resort to a divide-and-conquer approach–we pool the underlying points of every stroke, solve the proximity problem for points and then interpret the results at the stroke level. Each stroke is in general a one-dimensional (parametric) curve. We first sample each stroke uniformly (in terms of its arc length) into a set of points and then represent the input model at two levels: strokes consisting of points and an independent set of points from all strokes. Every point stores the tangent to the stroke at that point, its screen-space position and orientation (projected tangent), and rendering attributes like thickness, color, transparency, stroke texture id, etc. Our processing pipeline works fully in screen space.

In general, we estimate the expected "pairability" $E(S,T)$ between strokes S and T as

$$E(S,T) \propto Co(S,T) * \sum_{(p,q):p\in S,q\in T,d(p,q)\leq\delta} (\vec{p}.\vec{q}) \tag{6.5.1}$$

where p and q are screen points with screen-space (normalized) tangents $\vec{p}$ and $\vec{q}$ and $d(.)$ is the Euclidean distance. Co(.) is the color similarity between S and T, but may include other suitable metrics as well. If two strokes are very near each other, then more pairs will be found, leading to a greater expected value. Thus $E(S,T)$ measures how "near", "locally parallel" and similar in color S and T are.

Many solutions for point proximity problems exist in computational geometry[14, 30]. As strokes can be added, deleted, or dynamically modified, we require a data structure that supports efficient nearest-point queries under motion, dynamic insertion and deletion. Hence, we use the $(1+\epsilon)$-deformable spanner by Gao *et al.*[33].

$(1+\epsilon)$-deformable Spanner

For a set of points in $\mathbb{R}^d$, an s-spanner is a graph on the set such that any pair of points is connected via some path in the spanner whose total length is at most s times the Euclidean distance between the points. A $(1+\epsilon)$-deformable spanner (for any $\epsilon > 0$) is a sparse spanner that is suitable for dynamic sets of points. Because of its hierarchical construction and sparseness, a $(1+\epsilon)$-spanner efficiently "repairs" itself incrementally whenever points move *continuously*. Thus, in the context of rendering, it works on the notion of using results from the previous frame to determine those for the current frame. Moreover it supports dynamic insertion and deletion of points, making it suitable for our setting. Specifically, for a set of n points in $\mathbb{R}^d$ bounded by an aspect ratio α (the ratio of the maximum and minimum distance between two points), the $(1+\epsilon)$-spanner supports insertion and deletion of a point in $O(\frac{h}{\epsilon^d})$ time, where $h = O(\log_2 \alpha)$. The nearest-point query – given a set of points, for each point p, enumerate all points within a distance δ from p is defined as a standard operation on this spanner. For k such pairs, the $(1+\epsilon)$-spanner supports this query in $O(k+n)$ time. Our pipeline uses this data structure *as-is* (like a black box). We refer the reader to Gao *et al.*[33] for further details.

We project all strokes onto the screen and build the $(1+\epsilon)$-spanner on these projected 2D points. After any screen-space movement, we update(repair) the spanner and query it to return all pairs of points that are within a distance δ (Section 6.4) from each other. As long as the movement is *incremental* (points do not move by a large distance abruptly), the updating operation is efficient irrespective of the actual nature of movement. Thus our pipeline is not limited to simple rigid transformations–*any* incremental motion can be handled.

The parameter ϵ controls the extent of approximation of level i in its parent level $i-1$ (we chose $\epsilon = 16$ empirically for all results shown here). Our experiments indicated that increasing the value of ϵ increases the time to "repair" the spanner during every frame.

Henceforth, all screen-projected points are represented in lower case while all strokes are represented in upper case. A screen-projected point p has a screen-space orientation $\vec{p}$.

Point Pairing

All pairs returned by the spanner are not useful in determining stroke-stroke proximity (e.g. the spanner returns pairs of adjacent points along the same stroke due to the absence of connectivity information in it). Intuitively, we try to select pairs that are near to each other, have similar colors and screen-space orientations.

During every frame, we build two tables from all the pairs returned by the spanner: $C(p, S)$ that maintains the "closest" point in every stroke S to a given point p, and $Q(S, T)$ that maintains the geometric likelihood that strokes S and T will be paired together (Q is the summation part of Equation 6.5.1). Let $p \rightarrow q(p \in S, q \in T)$ be a candidate pair of points returned by the spanner. If $S = T$, we reject the pair. Otherwise, we determine a score $Sim(p, q)$ proportional to the Euclidean distance and color difference between p and q. We update the entries $C(p, T)$ and $C(q, S)$ with $Sim(p, q)$ if necessary. Then we add $\vec{p} \cdot \vec{q}$ to $Q(S, T)$ and $Q(T, S)$. Thus $Q(S, T)$ maintains a score that is proportional to the number of pairs between points in S and T respectively, modulated by the similarity between their local gradients. It may be noted that C and Q are sparse as they contain data only for points and strokes that are near each other.

6.5.2 Stroke Pairing and Simplification

Given updated sparse tables C and Q, we now consolidate the results to pair strokes. Simplification happens at the stroke level, maintaining continuity wherever appropriate. We maintain continuity geometrically by preventing strokes from disintegrating. Although *apparent* continuity (interpreting multiple short strokes as a single long stroke) is maintained, it is not guaranteed (please refer to Section 6.7)[2].

Intuitively, we pair stroke S with a stroke T if $\rho\%$ of the points of S are paired to some point in T with comparable corresponding local gradients. We select an ordered pair of strokes (S, T) for simplification if $Co(S, T)Q(S, T) \geq \rho\,|S|$[3]. Our formulation allows us to

[2]Maintaining the integrity of special strokes like dashed lines depends on how they are represented in the application–if passed to the pipeline as one long stroke, continuity is maintained.

[3]It may happen that $Co(S, T)Q(S, T) \geq \rho\,|S|$ but $Co(T, S)Q(T, S) \leq \rho\,|T|$. This happens if S is shorter than and so overlaps only a small portion of T.

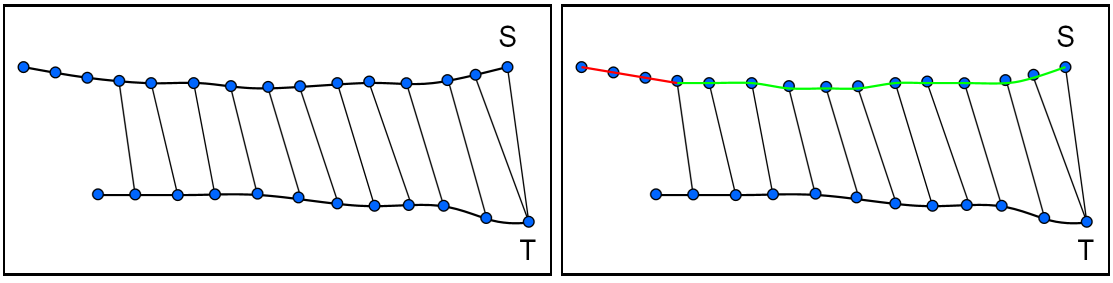
handle fork cases (neither of the end points of S pair with T) seamlessly. Since entries in Q are averaged over a stroke, they vary smoothly as points move continuously. Thus simplification is also incremental and smooth, providing temporal coherence.

It is important to note that once (S, T) are paired and simplified to U in a particular frame, no other strokes can pair with either S or T until U again separates into them. Thus in the case that multiple strokes have sufficient overlap with a single stroke S to trigger simplification, the first of these strokes (in the order in which they were created) is paired with S, ruling out any other pairs with S during that frame.

Stroke Simplification

The problem of simplifying two strokes into one is difficult in general, because some correspondence between their points must be known for any interpolation scheme. This problem occurs in many applications: simplifying strokes in NPR, representing rough over-traced sketches by a representative stroke, etc. It is often addressed by classifying strokes and employing multiple simplifying strategies.

If the two strokes were parameterized in a common domain, the parametrization would serve as an implicit correspondence. There are existing methods to fit curves to noisy point data [36] that parameterize points in a common domain. The strategy is to define a reference parameterized curve. Then for every point, the point on the curve that is nearest to it is determined and parameterized accordingly. While this seems inefficient for an interactive setting, we reuse the point pairs from the spanner to determine this correspondence.



Noting that at least $\rho\%$ of points of S are paired to some point in T for a pair (S, T), we

choose T as our reference curve and parameterize it in $(0, 1)$ using arc-length parametrization. For every point $p \in S$, if $C(p, T)$ exists, the parameter value at p is trivially known. This scheme divides S into a set of alternating parameterized (green above) and (possibly) unparameterized (red above) segments. We parameterize these segments using interpolation (if it lies between two parameterized segments) or extrapolation (otherwise). Another pass through S makes this parametrization consistent (monotonically increasing/decreasing). Then we sort the points in S and T according to the common parameter and fit a curve through this point set (a similar technique for parameter propagation was used by Kalnins *et al.*[54] to produce coherent silhouette animations).

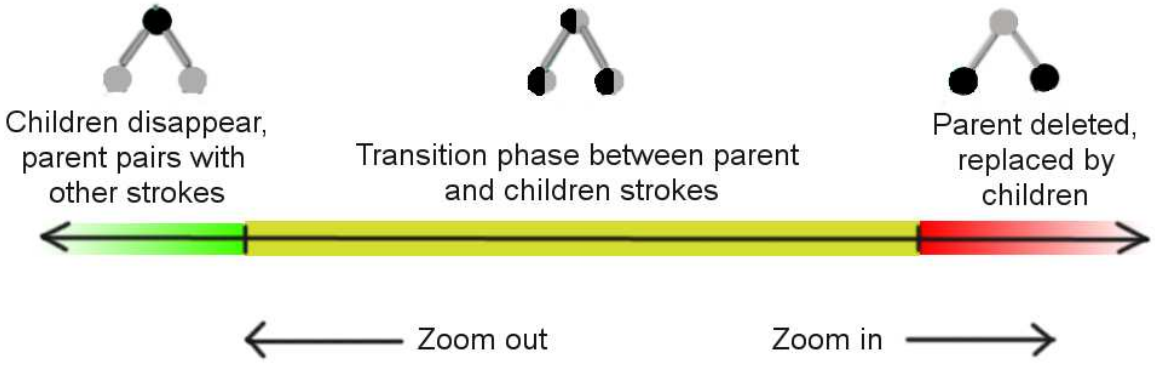
In regions where S and T do not coincide, this point set may cause a zig-zagged curve. To minimize this effect we choose Rational Gaussian curves[35] for interpolation. RaG curves work by centering a Gaussian function at every control point and using these weights to calculate interpolating points on the curve. They offer good control over local and global smoothness by changing the standard deviations of the interpolating Gaussians (we use $\sigma = \frac{3}{|S+T|}$ for all models). We use pre-computed Gaussian tables to improve speed. Figures 6.3 and 6.5 show how this single simplification scheme works well for both contour strokes (maintaining shape in the Gandhi and boat sketches) and hatching strokes (hull of the boat).

For a pair (S, T) simplified to a stroke U, U records a representative pair of points $(p_S \in S, q_T \in T)$ and the Euclidean distance $d_{first}(p_S, q_T)$ between them at the time of simplification. They are used to decide whether to move down the hierarchy (replace U with S and T).

6.5.3 Hierarchy Maintenance

Simplification creates a localized hierarchy of strokes. Strokes should smoothly merge into their parent, or should smoothly separate into their children for coherent animation.

Based on strokes S and T simplified into stroke U with the parameters (p_S, q_T) and $d_{first}(p_S, q_T)$ as explained previously and distance $d_{curr}(p_S, q_T)$ in the current frame, a life cycle of U can be defined in terms of the following phases:



Separation phase

(red phase;right)

In this phase characterized by $d_{curr}(p_S, q_T) > l * d_{first}(p_S, q_T)$, S and T are apart enough to exist as separate strokes. Thus we descend one level down the hierarchy and render them instead of U. Points of S and T are added to the spanner, and those of U are deleted. $l > 1$ provides a wider hysteresis loop (so that points are not repeatedly added to/deleted from the spanner between frames).

Transition phase

(yellow phase;middle)

In this phase characterized by $k * d_{first}(p_S, q_T) < d_{curr}(p_S, q_T) \leq l * d_{first}(p_S, q_T)$, U, S and T are rendered as if in transition between the two levels. Their opacity is governed by $d_{curr}(p_S, q_T)$, such that $\alpha(S) = \alpha(T) \propto \frac{d_{curr}(p_S, q_T)}{d_{first}(p_S, q_T)}$, $\alpha(U) = 1 - \alpha(S)$. $0 \leq k \leq 1$ creates a smooth transition.

Simplified phase

(green phase;left)

In this phase characterized by $d_{curr}(p_S, q_T) \leq k * d_{first}(p_S, q_T)$, S and T are too near each other to exist as separate strokes. If they were in the transient phase in the previous frame (they would have had low opacity then to be in this phase now), we stop rendering them and set U at full opacity. Points of S and T are removed from the spanner, and those

of U are added.

We implement this scheme using two lists. At every frame, one list is marked as the current list. Any strokes created during that frame (by simplification) are added to this list. We check every stroke in the list, and either push it (yellow phase) or its children (red phase) into the other list. The lists are swapped in the next frame. Any stroke that has already been paired (i.e. its ancestor exists in the list) is neglected.

To summarize, the following high-level operations are performed in order during every frame:

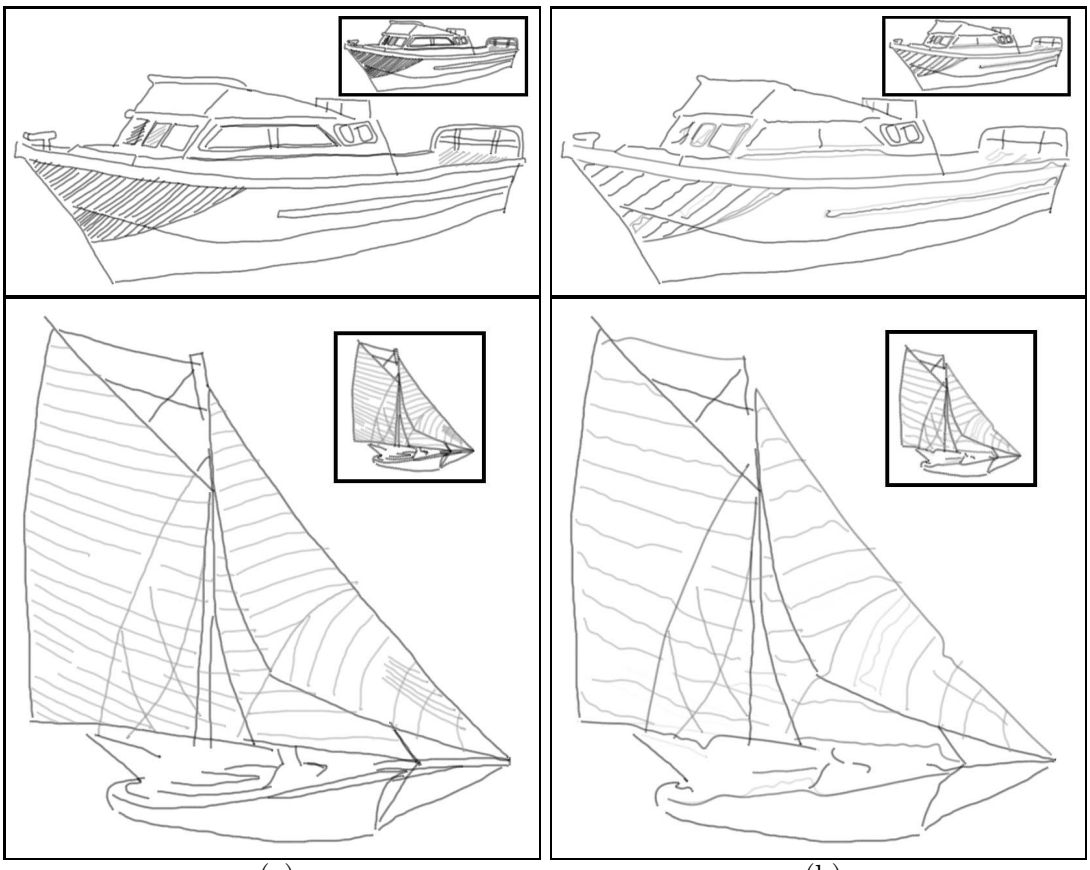
1. Verify the hierarchy. (Section 6.5.3)
2. Update the spanner, get and process all pairs. (Section 6.5.1)
3. Determine strokes pairs and simplify them. (Section 6.5.2)
4. Render the strokes.

6.5.4 Visibility Cues

If surfaces are available in case of 3D models, visibility cuing by occlusion may be performed easily in some cases. However in the general case of 3D curves, such occlusion may not be possible or desirable even if the surfaces that they lie on are available. For example, consider two 3D curves that lie on two known surfaces. How does one decide the surface bounds to get occluding polygons? Is it always conceptually sensible for one curve to occlude another?

We generate cues to alleviate depth ambiguity and visually de-emphasize distant parts of the model by further modulating the transparency of every stroke. Our technique is similar to the *haloed line effects* proposed by Appel *et al.*[7] and Elber[28]; however our pipeline implements it at no extra cost. For every intersection point of two strokes on the screen, we locally increase the transparency of the stroke that is behind by a certain amount. Thus, the stroke appears "lighter" or hidden behind the stroke that is in front. These intersection points can be approximated from the pairs of points returned by the spanner. Such local modulation of transparency values results in a global visibility cuing effect. In Figure 6.4(b),

the right wheel of the cart appears occluded due to these cues. In Figure 6.1(b), these cues reduce clutter when the Eiffel Tower is rendered from different distances.



(a) (b)

Figure 6.3: ***Results on 2D sketches of objects.*** *Column (a) shows the original sketch zoomed out to a certain level (thumbnail), while column (b) shows the sketch zoomed out and simplified to the same level (thumbnail) and the thumbnail magnified. The top row shows a sketched boat ($\delta = 4, \rho = 85$), while the bottom row shows a sketched sailboat($\delta = 8, \rho = 85$).*

6.6 Results

Figure 6.3 shows how our system can be used to render 2D sketches. These sketches were drawn on a tablet PC, using a simple program that allowed the user to draw strokes by tracing a photograph. Row 1 shows the 2D sketch of a boat. Notice how the hatching strokes on the hull and the contours of the boat are simplified correctly with our unified

simplification technique. Row 2 shows another 2D sketch of a sailboat. The thumbnail shows how the wrinkles on the sail were simplified upon zooming out.

Figure 6.5 shows how our system can be used to render creative sketches directly drawn on computer by an artist. Typically, such sketches are ambiguous and are drawn by over sketching with many small strokes. The first row shows a character sketch of Mahatma Gandhi. Notice how various silhouettes are drawn with multiple smaller strokes. Column (b) shows our simplified rendering where such strokes are consolidated into fewer strokes, thereby preventing them from thickening and darkening. The second row shows a sketch of a lamp. While sketching, the artist went over the same regions with strokes of multiple colors, due to which there is some color mixing as the sketch is zoomed out (thumbnail in (c)). Some parts (yellow shading on the lamp shade) were drawn using a few long, winding strokes. The lamp simplifies into (d) if such strokes are taken in their entirety. It can be seen that such strokes are not simplified significantly as their length mitigates sufficient overlap with any other stroke to trigger simplification. To improve this, we first segment such strokes by monitoring abrupt curvature changes. Simplification after such segmentation is shown in (e). Notice how the lamp is now simplified to a greater extent and resembles the original sketch more (the two thumbnails), in terms of stroke placement and density.

Figure 6.6 shows a comparison with a result from Barla *et al.*[9] by using the data of one of their sketches. Our pipeline simplifies the hatching strokes similar to theirs (without explicitly labeling them so). Result (c) was obtained by fixing δ and ρ so that zooming out to the same size as the thumbnail in (b) results in an image that looks similar to the original sketch in terms of shape and tone. Notice how the overall appearance of the thumbnail of (c) matches that of the original sketch. In particular, notice the locally darker hatches on the big branch on the right that are somewhat preserved in our thumbnail. This is reflected in the sparse dark stroke on this branch in the magnified thumbnail in (c).

Figure 6.4 shows how 3D sketches (here drawn using the 3D6B interface[53]) can be rendered using our system. Notice how our visibility cues correctly depict the depth by "occluding" the right wheel of the cart (first row). In the second row , notice how the tone of the dense strokes on the roof, the shingle pattern and the shape of the clocks are

Model	$N_{strokes}$	N_{pts}(initial)	$FR_{avg}(fps)$
Sailboat	281	4817	7.7
Boat	359	6000	9.24
Gandhi	757	7197	12.52
Lamp	2559	31444	3.16
Tree	2376	34424	11.3
Clock tower	961	21445	2.69
Cart	162	8851	5.45
Eiffel	2742	22119	5.11

Table 6.1: ***Performance results.*** *$N_{strokes}$: number of strokes in the model, $N_{pts}(initial)$: number of points initially added into the spanner. All frame rates were captured at a* 950×950 *resolution.*

retained. Figure 6.1 shows how our system can be used to render 3D wireframe CAD models ($\delta = 3$,$\rho = 80$). This Eiffel Tower model was obtained from Google 3D Warehouse[1] and rendered without any surface information.

Table 6.1 shows the average frame rates for the results shown in this paper. These frame rates were calculated for a resolution of 950×950 on a desktop machine with a 3.0 GHz Intel Pentium 4 processor and 1 GB RAM. The implementation is fully CPU-based. As the screen size decreases, the number of points in the spanner decreases due to simplification, leading to faster frame rates.

6.7 Qualitative Analysis and Limitations

The main highlight of our pipeline is that it works in a dynamic setting for an arbitrary sequence of transformations. Towards this goal, the most expensive operation is to determine proximity between strokes, which we perform efficiently with the $(1+\epsilon)$-spanner. The cost of dynamism principally comes in the form of per-frame updates of the spanner, which is why the performance of our pipeline is interactive, but not real-time. It may be possible to improve the performance of the spanner if only specific transformations are allowed. Implementation of the spanner *as-is* on the GPU will greatly boost performance without

compromising on generality. I have reserved this for the near future.

The main advantage of a unified simplification strategy (Section 6.5.2) is that strokes need not be annotated (e.g. "contour" and "hatching") explicitly. In a sketching session, this may have to be done manually which would seem contextually unnecessary. However the notion of *continuity* may be compromised in some cases. Consider a (implied) long stroke that is drawn using two strokes A and B overlapping end-to-end. If a third stroke C pairs with A to create a simplified stroke D, then D and B may not look like a continuous long stroke as A and B were meant to be. As small values of δ are normally used to prevent over-simplification, such an effect is usually not obvious. However it is theoretically possible.

The two parameters δ and ρ offer limited control over subjective simplification effects. Both δ and ρ are applied globally, i.e. to the entire (projection of) model. Although this approach makes them intuitive to change, it makes local customizations difficult. For example, it is not possible to simplify a local region more while retaining the original simplification elsewhere. A possible solution could be to allow additional localized simplification effects like those by Cole *et al.*[19] as a post-process. Another issue concerns the "constancy" of δ, i.e. although δ can be changed by the user, it is independent of the actual scale of the model. This creates artifacts when the model is scaled down greatly. At that stage the model may be over-simplified. A better strategy could be to modulate the value of δ with the overall scale of the model, but achieving a proper control is difficult because this correspondence is often model-dependent. Such a strategy may also make δ less intuitive to change.

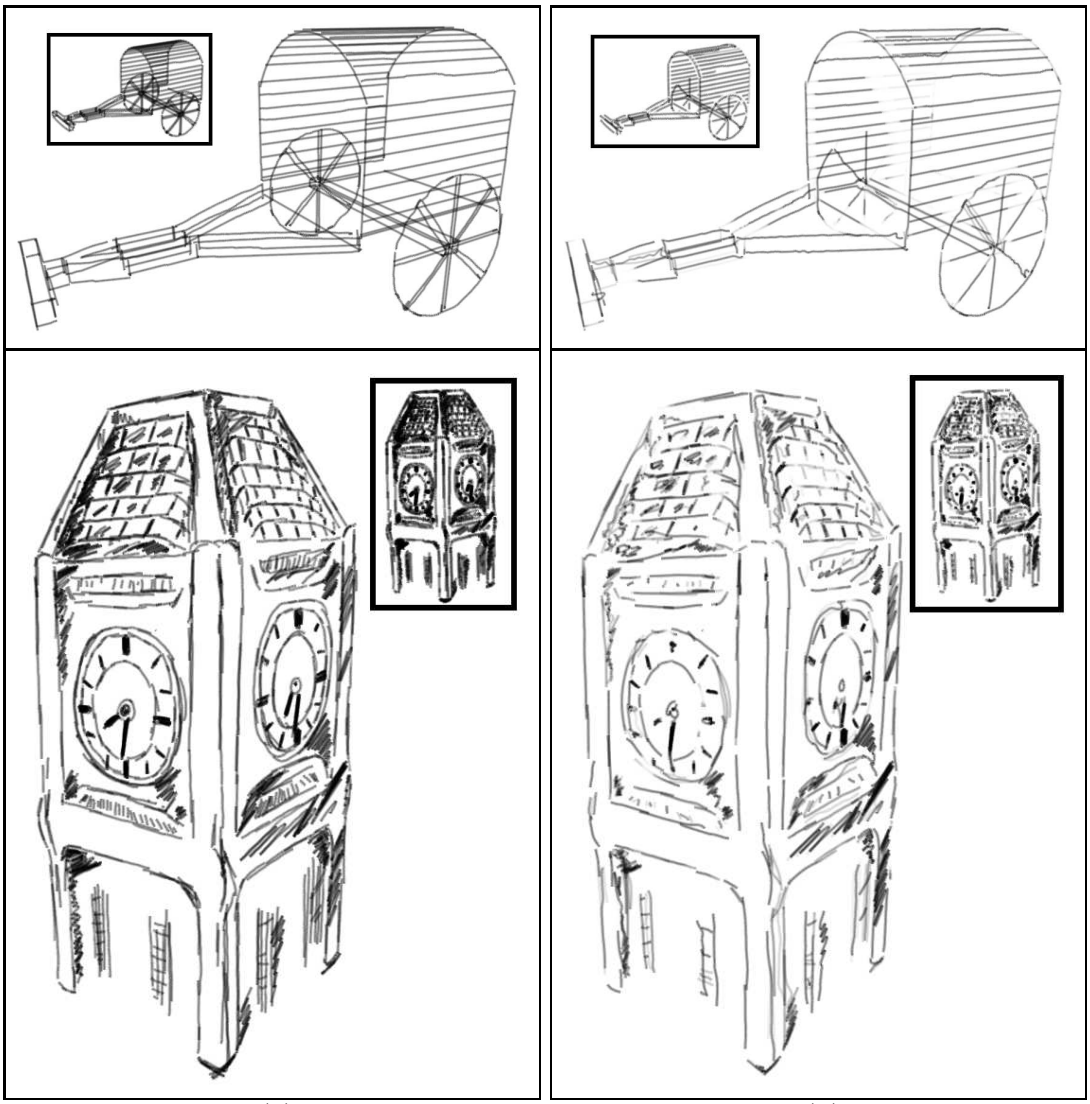
Another issue concerns our stroke simplification algorithm. Our parameter propagation when creating a merged stroke is based on a simple merging procedure. This may cause strokes to jump (spatially, not temporally) from the unpaired parts to those that are paired. This, along with our method of determining the standard deviations of the RaG curves sometimes creates wiggly strokes (some strokes in Figure 6.6(c)).

6.8 Remarks and Future Work

This work simplifies line drawings and generates visibility cues from purely 2D/3D line-based models obtained from various types of data. By using a $(1+\epsilon)$-spanner to determine stroke proximity, we dynamically build and maintain a stroke hierarchy based on the high-level principles of proximity and continuity. The line drawings thus created are geometrically meaningful and temporally coherent.

Synthesis of line drawings is a complementary and more difficult problem, in which strokes are synthesized (instead of simplified) to achieve the same goals of shape and tone preservation. In the future, I wish to extend our pipeline to synthesize line drawings. In addition to line-based NPR, I would like our pipeline to support other styles of abstract drawings like point stippling. Point-based and line-based NPR effects are often treated exclusively because of operations required to realize each of them. I envision a hybrid pipeline that feeds off a common efficient representation of underlying data, supports both styles simultaneously and switches between them interactively, and that offers intuitive controls for an artistic user.

Lastly, I am interested in working further at the overlapping areas of sketch-based applications and non-photorealistic rendering. So far, these two areas have been treated separately: the former concentrating on modeling while the latter on rendering models. However, as non-photorealistic rendering is principally aimed at reproducing the natural form of hand-drawn sketches, I feel that the two areas have great potential to reinforce each other. First, I am interested in investigating how the effectiveness and subjectivity of non-photorealistic renditions can be improved by using sketched inputs. This would facilitate artistic control over a computer-based process that is essentially algorithmic and/or statistical in nature. It would provide local and global "NPR-by-example" control to the artist instead of relying on parameters and sliders. Secondly, I am interested in exploring the role of sketch-based NPR in art education. I believe the collective knowledge of computer scientists in understanding and reproducing artistic renditions can be used to create very effective tools for teaching and refining the art of sketching and painting in art classes.



(a) (b)

Figure 6.4: ***Results on 3D sketches (created using the 3D6B interface [53]).*** *Column (a) shows the original sketch zoomed out to a certain level (thumbnail). Column (b) shows the sketch zoomed out and simplified to the same level(thumbnail) and the magnified thumbnail. First row: 3D sketch of a cart ($\delta = 5, \rho = 85$). Second row: two facades of a clock tower ($\delta = 5, \rho = 80$).*

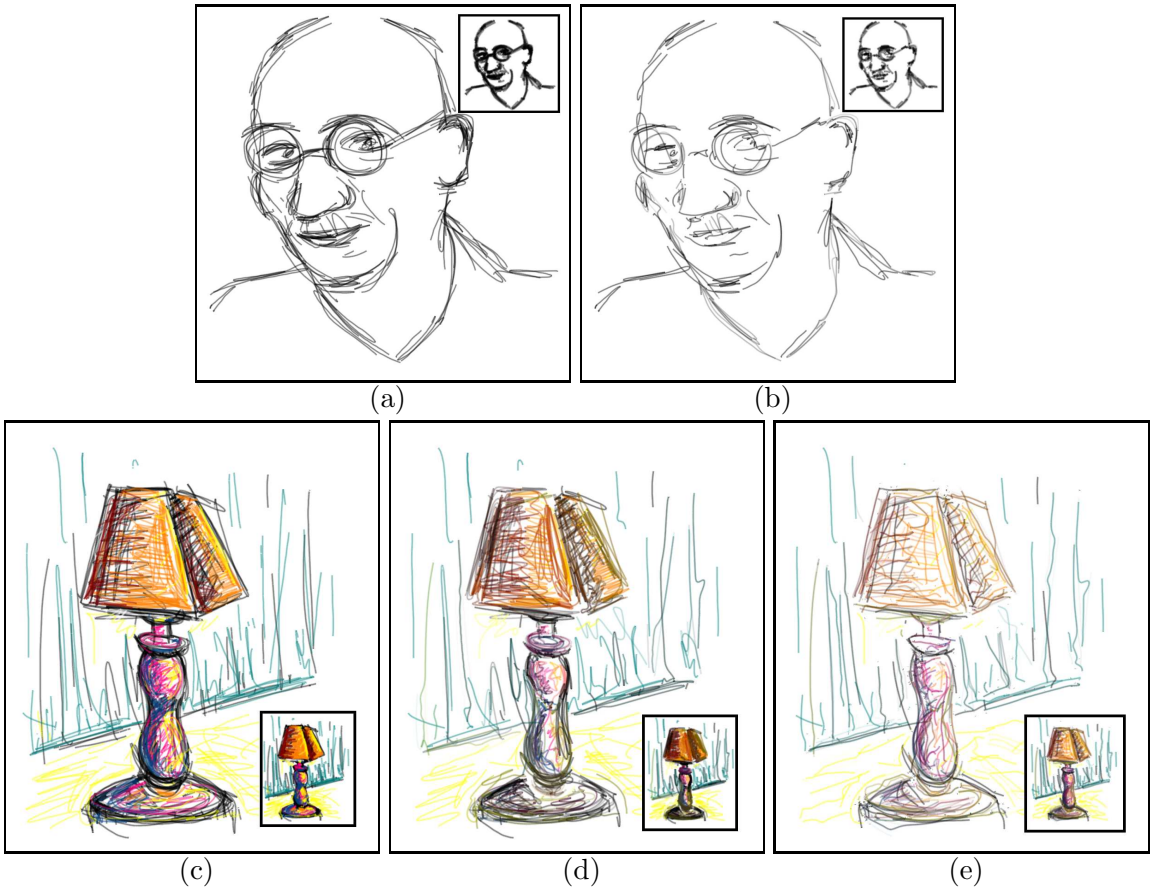
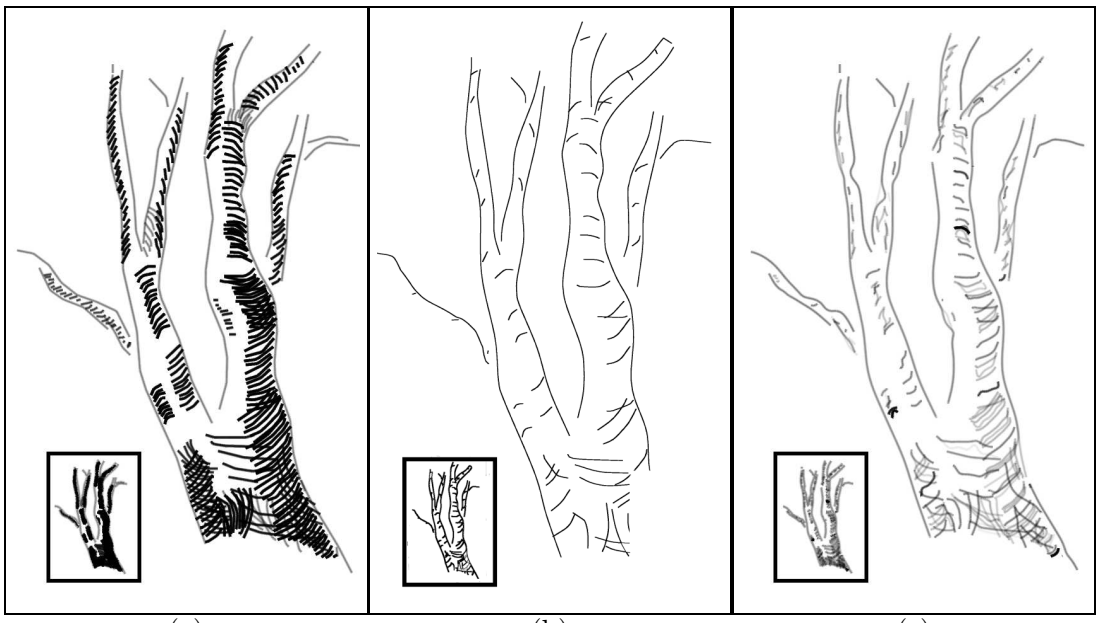


Figure 6.5: ***Results on artistic sketches.*** *First row: (a) a sketch of Mahatma Gandhi made by an artist, zoomed out to a certain level (thumbnail). (b) the simplified sketch zoomed out to the same level (thumbnail) and the thumbnail magnified ($\delta = 5, \rho = 80$). Notice the typical over-tracing sketching style in the sketch that is suitably simplified by our rendering. Second row: (a) an artistic sketch of a lamp, zoomed out to a certain level (thumbnail). This sketch was made by the artist by sketching repeatedly with overlapping strokes of various colors, giving it a composite appearance. Notice how parts of the lamp shade are sketched using a single winding stroke. If such strokes are taken in their entirety, the simplified result appears as in (b)($\delta = 6, \rho = 85$). To improve on this, we break the stroke into segments so that they may be simplified, producing the result in (c). Notice how the strokes in the lamp shade have been simplified correctly due to this segmentation.*



(a) (b) (c)

Figure 6.6: ***Comparison with previous work.*** *(a) A sketch from Barla et al. [9]. (b) Their result upon zooming out. (their result was darkened to better compare with our rendering style. The number and shapes of strokes were unchanged. Reproduced with permission from the authors) (c) Our result when zoomed out to the level shown in the bottom left thumbnails ($\delta = 6, \rho = 90$). Our simplification preserves the apparent tone produced by the hatching strokes as in their result, shown in (b). The result was obtained by fixing δ and ρ empirically and zooming out till a satisfactory image at the same size was obtained.*

Bibliography

[1] Google 3d warehouse. `http://sketchup.google.com/3dwarehouse/`.

[2] Photosynth. `http://labs.live.com/photosynth/`.

[3] Wacom bamboo. `http://www.wacom.com/BambooTablet/index.cfm`.

[4] Wacom cintiq. `http://www.wacom.com/cintiq/index.cfm`.

[5] Active image contours. *www.cs.wisc.edu/ pingelm/Algo.html*, Retrieved: Apr 6, 2004.

[6] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Proc. Eurographics '87*, pages 3–10, 1987.

[7] A. Appel, F. Rohlf, and A. Stein. The haloed line effect for hidden line elimination. *Proc. SIGGRAPH*, pages 151–157, 1979.

[8] Dana H. Ballard. Strip trees: a hierarchical representation for curves. *Commun. ACM*, 24(5):310–321, 1981.

[9] Pascal Barla, Joëlle Thollot, and Francois X. Sillion. Geometric clustering for line drawing simplification. In *Proc. EGSR*, pages 183–192, 2005.

[10] Harry G. Barrow and Jay M. Tenenbaum. Interpreting line drawings as three-dimensional surfaces. *Artificial Intelligence*, 17:75–116, 1981.

[11] Ronen Barzel. Lighting controls for computer cinematography. *Journal of Graphics Tools*, 2(1):1–20, 1997.

[12] Marcelo Bertalmio, Guillermo Sapiro, Vicent Caselles, and Coloma Ballester. Image inpainting. In *Proc. SIGGRAPH*, pages 417–424, 2000.

[13] Michael Burns, Janek Klawe, Szymon Rusinkiewicz, Adam Finkelstein, and Doug DeCarlo. Line drawings from volume data. *Proc. SIGGRAPH*, pages 512–518, 2005.

[14] Callahan and Kosaraju. Algorithms for dynamic closest pair and n-body potential fields. In *SODA: ACM-SIAM Symp. Discrete Algorithms*, 1995.

[15] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8:679–698, nov 1986.

[16] H. Cantzler, Robert B. Fisher, and Michel Devy. Improving architectural 3d reconstruction by plane and edge constraining. In *BMVC*, 2002.

[17] Joseph Jacob Cherlin, Faramarz Samavati, Mario Costa Sousa, and Joaquim A. Jorge. Sketch-based modeling with few strokes. In *Proc. SCCG*, pages 137–145, 2005.

[18] Roberto Cipolla, Tom Drummond, and D. P. Robertson. Camera calibration from vanishing points in image of architectural scenes. In *Proc. British Machine Vision Conference*, 1999.

[19] Forrester Cole, Doug DeCarlo, Adam Finkelstein, Kenrick Kin, Keith Morley, and Anthony Santella. Directing gaze in 3D models with stylized focus. In *Proc. EGSR*, pages 377–387, 2006.

[20] A. C. Costa, A. A. Sousa, and F. N. Ferreira. Lighting design: A goal based approach using optimisation. In *Rendering Techniques*, pages 317–328, 1999.

[21] Antonio Criminisi. *Accurate Visual Metrology from Single and Multiple Uncalibrated Images*. Springer Verlag, 2001.

[22] Antonio Criminisi, Ian D. Reid, and Andrew Zisserman. Single view metrology. *International Journal of Computer Vision*, 40(2):123–148, 2000.

[23] Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. Modeling and rendering architecture from photographs: a hybrid geometry- and image-based approach. In *SIGGRAPH*, 1996.

[24] Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. *Proc. SIGGRAPH*, pages 11–20, 1996.

[25] Julie Dorsey, Songhua Xu, Gabe Smedresman, Holly Rushmeier, and Leonard McMillan. The mental canvas: A tool for conceptual architectural design and analysis. *Proc. Pacific Graphics*, pages 201–210, 2007.

[26] Alexei A. Efros and Thomas K. Leung. Texture synthesis by non-parametric sampling. *IEEE Intl. Conf. Computer Vision*, pages 1033–1038, September 1999.

[27] Lynn Eggli, Ching yao Hsu, Beat D. Bruderlin, and Gershon Elbert. *Computed-Aided Design*, 29(2):101–112, 1997.

[28] G. Elber. Line illustrations in computer graphics. *The Visual Computer*, 11(6):290–296, 1995.

[29] E.Lutton, H. Maitre, and J.Lopez-Krahe. Contribution to the determination of vanishing points using hough transform. *IEEE Trans. Pattern Analysis and Mach. Int. (PAMI)*, 16(4):430–438, 1994.

[30] David Eppstein. Spanning trees and spanners. In Jörg-Rudiger Sack and Jorge Urrutia, editors, *Handbook of Computational Geometry*, chapter 9, pages 425–461. Elsevier, 2000.

[31] Luiz Henrique Figueiredo, Jorge Solfi, and Luiz Velho. Approximating parametric curves with strip trees using affine arithmetic. *Computer Graphics Forum,*, 22(2):171–180, June 2003.

[32] M.A. Fisher and R.C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Comm. of the ACM*, 24:381–395, 1981.

[33] Jie Gao, Leonidas J. Guibas, and An Nguyen. Deformable spanners and applications. *Comput. Geom. Theory Appl.*, 35(1):2–19, 2006.

[34] Leslie Gennari, Levent Burak Kara, and Thomas Stahovich. Combining geometry and domain knowledge to interpret hand-drawn diagrams. *Computers and Graphics*, 29(4):547–562, 2005.

[35] Ardeshir Goshtasby. Geometric modelling using rational gaussian curves and surfaces. *Computer-Aided Design*, 27(5):363–375, 1995.

[36] Ardeshir Goshtasby. Grouping and parameterizing irregularly spaced points for curve fitting. *ACM Trans. Graph.*, 19(3):185–203, 2000.

[37] Stéphane Grabli, Frédo Durand, and François Sillion. Density measure for line-drawing simplification. In *Proc. Pacific Graphics*, 2004.

[38] I. J. Grimstead and R. R. Martin. Creating solid models from single 2D sketches. In *SMA '95: Proceedings of the Third Symposium on Solid Modeling and Applications*, pages 323–337, 1995.

[39] Mark D. Gross and Ellen Yi-Luen Do. Ambiguous intentions: a paper-like interface for creative design. In *Proc. UIST*, pages 183–192, 1996.

[40] S. Gumhold. Maximum entropy light source placement. In *Proc. IEEE Visualization '02*, pages 275–282, 2002.

[41] Paul Haeberli. Paint by numbers:abstract image representations. *Proc. SIGGRAPH*, pages 207–214, 1990.

[42] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[43] Miloš Hašan, Fabio Pellacini, and Kavita Bala. Matrix row-column sampling for the many-light problem. *ACM Trans. Graph.*, 26(3):26, 2007.

[44] Martin Herman and Takeo Kanade. The 3D MOSAIC scene understanding system: incremental reconstruction of 3d scenes for complex images. *Readings in computer vision: issues, problems, principles, and paradigms*, pages 471–482, 1987.

[45] Aaron Hertzmann and Denis Zorin. Illustrating smooth surfaces. *Proc. SIGGRAPH*, pages 517–526, 2000.

[46] Derek Hoiem, Alexei A. Efros, and Martial Hebert. Automatic photo pop-up. *Proc. SIGGRAPH*, pages 577–584, 2005.

[47] Youichi Horry, Ken ichi Anjyo, and Kiyoshi Arai. Tour into the picture: using a spidery mesh interface to make animation from a single image. *Proc. SIGGRAPH*, pages 225–232, 1997.

[48] D.A. Huffman. Impossible objects as nonsense sentences. *Machine Intelligence*, 6:295–303, 1971.

[49] Takeo Igarashi and John F. Hughes. A suggestive interface for 3d drawing. In *Proc. UIST*, pages 173–181, 2001.

[50] Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. Teddy: A sketching interface for 3d freeform design. In *Proc. SIGGRAPH*, pages 409–416, 1999.

[51] Takashi Ijiri, Shigeru Owada, Makoto Okabe, and Takeo Igarashi. Floral diagrams and inflorescences: interactive flower modeling using botanical structural constraints. *Proc. SIGGRAPH*, pages 720–726, 2005.

[52] Kyuman Jeong, Alex Ni, Seungyong Lee, and Lee Markosian. Detail control in line drawings of 3D meshes. *The Visual Computer*, 21(8-10):698–706, September 2005. Special Issue of Pacific Graphics 2005.

[53] Kiia Kallio. 3d6b editor: Projective 3d sketching with line-based rendering. In *Proc. Eurographics Workshop on Sketch-Based Interfaces and Modeling*, pages 73–79, 2005.

[54] Robert D. Kalnins, Philip L. Davidson, Lee Markosian, and Adam Finkelstein. Coherent stylized silhouettes. In *Proc. SIGGRAPH*, pages 856–861, 2003.

[55] Robert D. Kalnins, Lee Markosian, Barbara J. Meier, Michael A. Kowalski, Joseph C. Lee, Philip L. Davidson, Matthew Webb, John F. Hughes, and Adam Finkelstein. Wysiwyg npr: Drawing strokes directly on 3d models. In *Proc. SIGGRAPH 2002*, pages 755–762, 2002.

[56] T. Kanade. Recovery of the three dimensional shape of an object from a single view. *Artificial Intelligence*, 17:409–460, 1980.

[57] Matthew Kaplan and Elaine Cohen. Producing models from drawings of curved surfaces. In *Proc. SBIM*, pages 51–58, 2006.

[58] O. Karpenko, J. F. Hughes, and R. Raskar. Free-form sketching with variational implicit surfaces. *Proc. Eurographics*, pages 585–594, 2002.

[59] Olga Karpenko, John F. Hughes, and Ramesh Raskar. Epipolar methods for multi-view sketching. In *Proc. Eurographics Symposium on Sketch-Based Interfaces and Modeling*, pages 167–174, 2004.

[60] Olga Karpenko, John F. Hughes, and Ramesh Raskar. Epipolar methods for multi-view sketching . In *Eurographics Workshop on Sketch-Based Interfaces and Modeling*, pages 167–173, 2004.

[61] Olga A. Karpenko and John F. Hughes. Smoothsketch: 3d free-form shapes from complex sketches. *Proc. SIGGRAPH*, pages 589–598, 2006.

[62] J. K. Kawai, J. S. Painter, and M. F. Cohen. Radioptimization: goal based rendering. In *Proc. SIGGRAPH '93*, pages 147–154, 1993.

[63] Michael A. Kowalski, Lee Markosian, J. D. Northrup, Lubomir Bourdev, Ronen Barzel, Loring S. Holden, and John F. Hughes. Art-based rendering of fur, grass, and trees. In *Proc. SIGGRAPH*, pages 433–438, 1999.

[64] A.W. Kristensen, T. Akenine-Moller, and H.W. Jensen. Precomputed local radiance transfer for real-time lighting design. In *Proc. SIGGRAPH '05*, pages 1208–1215, 2005.

[65] James Landay and Brad Myers. Interactive sketching for the early stages of user interface design. In *CHI '95: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 43–50, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[66] Y.G. Leclerc and M.A. Fischler. An optimization-based approach to the interpretation of single line drawings as 3d wire frames. *IJCV*, 9(2):113–136, November 1992.

[67] C. H. Lee, X. Hao, and A. Varshney. Light collages: Lighting design for effective visualization. In *Proc. IEEE Visualization '04*, pages 281–288, 2004.

[68] Sylvain Lefebvre and Hugues Hoppe. Parallel controllable texture synthesis. *Proc. SIGGRAPH*, 24(3):777–786, 2005.

[69] R. Lengagne, J.P. Tarel, and O. Monga. From 2d images to 3d face geometry. pages 301–306, 1996.

[70] Yin Li, Jian Sun, Chi-Keung Tang, and Heung-Yeung Shum. Lazy snapping. *Proc. SIGGRAPH*, pages 303–308, 2004.

[71] Lin Liang, Ce Liu, Ying-Qing Xu, Baining Guo, and Heung-Yeung Shum. Real-time texture synthesis by patch-based sampling. *ACM Trans. Graphics.*, 20(3):127–150, 2001.

[72] D. Liebowitz, A. Criminisi, and A. Zisserman. Creating architectural models from images. *Proc. Eurographics*, 18:39–50, 1999.

[73] Hod Lipson and Moshe Shpitalni. Optimization-based reconstruction of a 3d object from a single freehand line drawing. *Journal of Computer Aided Design*, 28(8):651–663, 1996.

[74] Hod Lipson and Moshe Shpitalni. Correlation-based reconstruction of a 3d object from a single freehand sketch. *AAAI Spring Symposium on Sketch Understanding*, pages 99–104, 2002.

[75] Martti Mäntylä. Boolean operations of 2-manifolds through vertex neighborhood classification. *ACM Transactions on Graphics*, 5(1):1–29, 1986.

[76] T. Marill. Emulating the human interpretation of line drawings as three-dimensinoal objects. *IJCV*, 6(2):147–161, 1991.

[77] J. Marks, B. Andalman, P.A. Beardsley, W. Freeman, S. Gibson, J. Hodgins, T. Kang, B. Mirtich, H. Pfister, W. Ruml, K. Ryall, J. Seims, and S. Shieber. Design galleries: A general approach to setting parameters for computer graphics and animation. In *Proc. SIGGRAPH '97*, pages 389–400, August 1997.

[78] Leonard McMillan and Gary Bishop. Plenoptic modeling: an image-based rendering system. *Proc. SIGGRAPH*, pages 39–46, 1995.

[79] M.J.Magee and J.K. Aggarwal. Determining vanishing points from perspective images. *Proc. CVGIP*, 26(2):256–267, 1984.

[80] Ferran Naya, Julián Conesa, Manuel Contero, Pedro Company, and Joaquim Jorge. Smart sketch system for 3d reconstruction based modeling. In *Smart Graphics*, pages 58–68, 2003.

[81] R. Ng, R. Ramamoorthi, and P. Hanrahan. All-frequency shadows using non-linear wavelet lighting approximation. In *Proc. SIGGRAPH '03*, pages 376–381, 2003.

[82] Minh X. Nguyen, Hui Xu, Xiaoru Yuan, and Baoquan Chen. Inspire: An interactive image assisted non-photorealistic rendering system. In *Proc. Pacific Graphics*, pages 472–477, 2003.

[83] Beom-Soo Oh and Chang-Hun Kim. Progressive 3d reconstruction from a sketch drawing. In *Proc. Pacific Graphics*, page 108, 2001.

[84] Beom-Soo Oh and Chang-Hun Kim. Self-correctional 3d shape reconstruction from a single freehand line drawing. In *Proc. ICCSA*, pages 528–538, 2003.

[85] Byong Mok Oh, Max Chen, Julie Dorsey, and Frédo Durand. Image-based modeling and photo editing. *Proc. SIGGRAPH*, pages 433–442, 2001.

[86] Makoto Okabe, Yasuyuki Matsushita, Li Shen, and Takeo Igarashi. Illumination brush: Interactive design of all-frequency lighting. In *Proc. Pacific Graphics*, pages 171–180, 2007.

[87] G. Patow and X. Pueyo. A survey of inverse rendering problems. *Computer Graphics Forum*, 22(4):663–688, 2003.

[88] P.Company, J.Conesa, and M.Contero. Tentative level-inflation in line drawings reconstruction. Technical report, February 2001.

[89] Fabio Pellacini, Frank Battaglia, R. Keith Morley, and Adam Finkelstein. Lighting with paint. *ACM Trans. Graph.*, 26(2):9, 2007.

[90] Fabio Pellacini, Kiril Vidimce, Aaron Lefohn, Alex Mohr, Mark Leone, and John Warren. Lpics: a hybrid hardware-accelerated relighting engine for computer cinematography. *ACM Trans. Graph.*, 24(3):464–470, 2005.

[91] João Paulo Pereira, Joaquim A. Jorge, Vasco A. Branco, and Fernando Nunes Ferreira. Towards calligraphic interfaces: Sketching 3d scenes with gestures and context icons. In *WSCG*, 2000.

[92] A. Piquer, R.R. Martin, and P. Company. Using skewed mirror symmetry for optimisation-based 3d line-drawing recognition. In *Proc. 5th IAPR International Workshop on Graphics Recognition*, pages 182–193, 2003.

[93] P. Poulin and A. Fournier. Lights from highlights and shadows. In *Proc. SI3D '92*, pages 31–38, 1992.

[94] P. Poulin, K. Ratib, and M. Jacques. Sketching shadows and highlights to position lights. In *Proc. CGI '97*, page 56, 1997.

[95] Pierre Poulin and Alain Fournier. Painting surface characteristics. In *Proc. EGSR*, pages 160–169, 1995.

[96] Mukta Prasad and Andrew Fitzgibbon. Single view reconstruction of curved surfaces. In *Proc. CVPR*, 2006.

[97] Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein. Real-time hatching. In *Proc. SIGGRAPH*, page 581, New York, NY, USA, 2001. ACM Press.

[98] William Press, Saul Teukolsky, William Vetterling, and Brian Flannery. *Numerical Recipes in C++: The Art of Scientific Computing*. Cambridge University Press, New York, NY, 2002.

[99] Lijun Qu, Xiaoru Yuan, Minh X. Nguyen, Gary Meyer, and Baoquan Chen. Perceptually guided texture mapping on points. In *Proc. IEEE/Eurographics Sym. Point-based Graphics*, pages 95–102, 2006.

[100] Alex Reche-Martinez, Ignacio Martin, and George Drettakis. Volumetric reconstruction and interactive rendering of trees from photographs. *Proc. SIGGRAPH*, pages 720–727, 2004.

[101] L.G. Roberts. *Machine Perception of 3-D Solids*. PhD thesis, MIT, 1963.

[102] C. Rother. A new approach for vanishing point detection in architectural environments. In *BMVC*, pages 382–391, 2000.

[103] C. Schoeneman, J. Dorsey, B. Smits, J. Arvo, and D. Greenburg. Painting with light. In *Proc. SIGGRAPH '93*, pages 143–146, 1993.

[104] Eric Schweikardt and Mark D. Gross. Digital clay: deriving digital models from freehand sketches. In *Proc. ACADIA*, pages 202–211, 1998.

[105] R. Shacked and D. Lischinski. Automatic lighting design using a perceptual quality metric. *Computer Graphics Forum*, 20(3), 2001.

[106] Amit Shesh and Baoquan Chen. Smartpaper–an interactive and user-friendly sketching system. *Proc. Eurographics*, 23:301–310, 2004.

[107] Amit Shesh and Baoquan Chen. User interface design and realization of a design-by-sketches system. Technical report, University of Minnesota (TR.04.01), 2004.

[108] Amit Shesh and Baoquan Chen. Peek-in-the-pic: Architectural scene navigation from a single picture using line drawing cues. *Proc. Pacific Graphics (Short Paper)*, 2005.

[109] Amit Shesh and Baoquan Chen. Lasels–line-based primitives for representing and rendering of 2d and 3d sketches with fidelity. Technical report, University of Minnesota (TR.06.04), 2006.

[110] Amit Shesh and Baoquan Chen. Efficient and dynamic simplification of line drawings. In *Proc. Eurographics*, pages 537–545, 2008.

[111] Amit Shesh and Baoquan Chen. Peek-in-the-pic: Flying through architectural scenes from a single image. *Computer Graphics Forum (accepted with major revisions)*, 2008.

[112] Moshe Shpitalni and Hod Lipson. Identification of faces in a 2d line drawing projection of a wireframe object. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 18(10):99–104, 1996.

[113] Moshe Shpitalni and Hod Lipson. Classification of sketch strokes and corner detection using conic sections and adaptive clustering. *ASME Journal of Mechanical Design*, 119(2):131–135, 1997.

[114] Noah Snavely, Steven M. Seitz, and Richard Szeliski. Photo tourism: exploring photo collections in 3d. *ACM Trans. Graph.*, 25(3):835–846, 2006.

[115] Mario Costa Sousa and Przemyslaw Prusinkiewicz. A few good lines. In *Proc. Eurographics*, pages 381–390, 2003.

[116] Ivan Sutherland. *Sketchpad: A man-machine graphical communication system*. PhD thesis, 1963.

[117] R. Szeliski. Image mosaicing for tele-reality applications. *WACV94*, pages 44–53, 1994.

[118] **Amit Shesh** and Baoquan Chen. Crayon lighting: Sketch-based illumination of models. In *Pacific Graphics 2006 (Poster paper)*, 2006.

[119] **Amit Shesh** and Baoquan Chen. Crayon lighting: Sketch-guided illumination of models. In *Proc. ACM GRAPHITE*, pages 95–102, 2007.

[120] Osama Tolba, Julie Dorsey, and Leonard McMillan. Sketching with projective 2d strokes. In *Proc. UIST 1999*, pages 149–157, 1999.

[121] Osama Tolba, Julie Dorsey, and Leonard McMillan. A projective drawing system. In *Proc. I3D Symposium on Interactive 3D Graphics*, 2001.

[122] P.A.C. Varley. *Automatic Creation of Boundary-Representation Models from Single Line Drawings*. PhD thesis, 2003.

[123] P.A.C Varley and R.R. Martin. Constructing boundary representation solid models from a two-dimensional sketch–topology of hidden parts. *Proc. First UK-Korea Wksp. Geometric Modeling and Computer Graphics*, pages 129–144, 2000.

[124] P.A.C. Varley, H. Suzuki, J. Mitani, and R.R. Martin. Interpretation of single sketch input for mesh and solid models. *International Journal of Shape Modelling*, 6(2):207–241, 2000.

[125] Ling Ling Wang and Wen-Hsiang Tsai. Camera calibration by vanishing lines for 3-d computer vision. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 13(4):370–376, April 1991.

[126] Donna J. Williams and Mubarak Shah. A fast algorithm for active contours and curvature estimation. *CVIGP Computer Vision Graphics Image Processing:Image Understanding*, 55(1):14–26, 1992.

[127] Brett Wilson and Kwan-Liu Ma. Rendering complexity in computer-generated pen-and-ink illustrations. In *Proc. NPAR*, pages 129–137, 2004.

[128] J. E. Windsheimer and G. W. Meyer. Implementation of a visual difference metric using commodity graphics hardware. In *Human Vision and Electronic Imaging IX, Proceedings of the SPIE*, volume 5292, pages 150–161, 2004.

[129] Georges Winkenbach and David Salesin. Computer-generated pen-and-ink illustration. In *Proc. SIGGRAPH*, pages 91–100, 1994.

[130] Hui Xu and Baoquan Chen. Stylized rendering of 3d scanned real world environments. In *Proc. NPAR*, pages 25–34, 2004.

[131] Xiaoru Yuan and Baoquan Chen. Illustrating surfaces in volume. In *Proc. IEEE/EG VisSym*, pages 9–16, 2004.

[132] G. Zachmann. The boxtree: Exact and fast collision detection of arbitrary polyhedra. In *SIVE Workshop*, pages 104–112, July 1995.

[133] Robert C. Zeleznik, Kenneth P. Herndon, and John F. Hughes. Sketch: An interface for sketching 3d scenes. In *Proc. SIGGRAPH 1996*, pages 163–170, 1996.