Modeling Architecture from Photographs

An Implementation of the Façade System

Zach Heath

A Masters Project

Submitted To

Computer Science Department

Oregon State University

Corvallis OR, 97331

Committee

Dr Eric Mortensen – Major Professor

Dr Bruce D'Ambrosio – Minor Professor

Dr Rajeev Pandey – Committee Member

June 18th, 2004

Abstract

The Façade photometric modeling system, developed by Paul E Debevec at Berkley, is capable of transforming a sparse set of camera images of an architectural scene into a photorealistic 3D model. Users define a rough model out of primitive building blocks and mark where a portion of the edges of the building blocks exist in the images. Façade then optimizes the parameters of those building blocks to create a model that best fits the images and the user input. This paper details my own implementation of the photometric modeling portion of Façade.

Table of Contents

Introduction

Architecture is a fundamental part of our society. It defines where we work, sleep, and play. It serves simultaneously as a functional work of art and a preserver of our history and culture. In the last fifty years computers have also become a fundamental part of our lives. They are powerful tools capable of expressing abstract ideas and allow for these ideas to be quickly communicated across great distances. A growing application provided by computers is the creation of virtual environments that allow a user to explore and interact with architectural models. Originally the creation of these models was time consuming and the resultant realism left much to be desired.

This paper describes an implementation of the image-based modeling system Façade [9], originally developed by Paul E. Debevec at Berkley, that accomplishes both of these goals. Façade allows users to realistically model a building from a sparse set of calibrated photographs. Users specify a set of primitive building blocks that the model should be composed of and also mark where these blocks are located within the photographs. A computationally intensive optimization process then finds the sizes and locations of these blocks. The images can then be used to fill in the missing details that were not modeled by the building blocks. Such uses for this system include architectural planning, special effects for entertainment, virtual tourism, and historical preservation.

Outline

This paper will explain how to use a series of 2D camera images to construct a 3D model of the buildings they depict.

Section 1 gives background information on what image-based modeling is and lists some techniques that can be used to create photorealistic models. Several other methods for modeling architectural scenes are also covered.

Section 2 overviews the camera calibration process. Calibration determines the function and parameters used by the camera to map points from the 3D world coordinates onto 2D images. This greatly simplifies the optimization process.

Section 3 gives an overview of the various parts of the Façade modeling system and how the optimization process works.

Section 4 specifies the set of primitives used to construct the architectural models, and how different types of primitives can be defined.

Section 5 explains how the user specifies what primitives exist in the model as well as how to constrain the parameters that determine the geometry of these primitives.

Section 6 defines how users setup camera views and their corresponding images. It also details the process for marking block edge locations within the input images.

Section 7 details how the resultant models can be viewed and checked for accuracy.

Section 8 details the math behind the optimization process.

Section 9 presents the results obtained by this implementation.

Section 10 provides suggestions on how to improve the overall system.

1. Background

What is image based modeling?

Computer vision and computer graphics are two complementary fields. In computer graphics, one takes a conceptual model and transforms it into a set of 2D images. Computer vision does the opposite by processing sets of 2D images to create a conceptual model. Image-based modeling (IBM) combines these two fields. Like computer vision, 2D images are first processed to produce a rough model. The model is then combined with the original 2D images to produce new images. IBM's greatest strength is its ability to produce photorealistic models since it uses actual photos in its creation. It also scales

well as an object becomes more complex because one can model the object using simple shapes but still display a lot of complexity by including the detail found in the images.

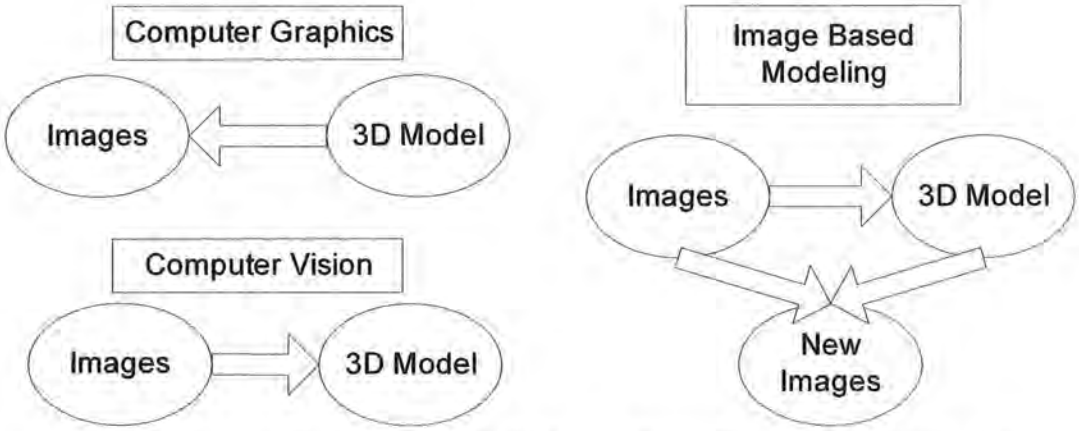

Figure 1.1: Computer Graphics, Computer Vision, and Image-Based Modeling Comparison.

Image based modeling does carry some drawbacks. One requires good images to produce good models. IBM also does not easily accommodate change once an environment has been established. One must either obtain images from a variety of different lighting conditions or capture a detailed photometric mapping for the model [8].

Modeling architecture takes full advantage of the power of image-based modeling. At the heart of most architectural designs lies a set of simple building blocks. These building blocks are easy to define and take very few parameters to designate the block's geometry. Constraints also exist which link the parameters of these blocks together since many will have the same dimensions or lay adjacent to one another. Most of the complexity of a building lies on the walls with intricate sculptures and designs. Human aided modeling can define the simple primitives while the images are used for optimization and detailing the complexities.

Previous Work

Stereo Correspondence

The traditional approach to constructing 3D models out of 2D images is through the use of stereo correspondence. A recent survey [13] documents the many varied techniques in

this field. Stereo correspondence finds the 3D location of the points in an object by comparing two images. When given a calibrated camera it is possible to determine exactly how a 3D object will be projected onto a 2D image. However, the inverse problem is not directly solvable with one image. Given a pixel location the corresponding 3D object is only constrained to lie on the ray emanating from the camera's center toward the pixel location. If we are given two images of the same object these rays will have an intersection point for the corresponding pixels that define the 3D location of the object. If we know the positions of our camera and we can create a correspondence for all of the points in the object we can build our model.

Humans are naturally good at picking correspondence between images, however this proves to be a difficult task for computers for several reasons. Camera locations need to be close together in order to produce views that are similar and thus easily matched, but close camera positions result in less accurate depth estimates. There are also problems with foreshortening where the relative size of the geometry in one image can be quite different than in the other due to the prospective geometry. This means that a large number of pixels in one image may map to a small set of pixels in another image. This combined with occlusions, where one image can see object points that another image cannot, results in pixels that have no correspondences. There are also cases where there is a lack of texture in the image resulting in a large number of pixels with the same intensity making the correspondence between neighboring pixels even more difficult.

Shape from Silhouette

Another modeling approach using multiple views [15] is finding the shape of an object from its silhouette. The process begins by modeling the object as a solid block. The block is then carved away until it more closely matches the actual object. This is done by first segmenting the input images, marking which pixels belong to the object and which pixels belong to the background. Each marked background pixel is then used to shoot a ray from the camera, through the pixel, and into the model. Each voxel that the ray intersects is removed from the current estimate of the model to make the model consistent with the images. Doing this from all of the camera views results in a convex hull of the

object that is composed of many voxel elements. This process has the advantage that it can model any type of convex surface. However, the accuracy of the model is limited by the size and number of voxel elements and can result in a "jaggy" model.

Voxel Coloring [14] is a similar approach. Instead of carving away voxels from the model, the model is built up out of voxels that are consistent with the input images. The model space is broken into several layers of potential voxels. The layers are traversed (outer layers first) having each voxel be projected into the camera images. Occlusions with other voxels are taken into account when doing the projections. The pixels that the voxel projects onto are then checked to see if they are consistent across all of the images. If the voxel is consistent it is added to the model. The model slowly grows into the object. Because voxel occlusions are checked the resultant model is not limited to a convex hull. Voxel Coloring is still limited by the same accuracy problems as Shape from Silhouette.

Range Scanning

A more mechanical approach [2] uses a laser range scanner to create a dense set of 3D position points for the object. Triangulation Based Range Scanners use a technique similar to stereo correspondence. The position of the camera and the laser are known helping constrain the geometry of the system. The camera tracks the observed position of the laser as it stripes across the object. Each observed point limits the 3D position to a ray extending from the camera center to the pixel. The location of the 3D point is the intersection between this ray and the ray extending from the laser. Time of Flight Scanners exploit the known velocity of the laser light. The scanner emits a laser pulse and records how long it takes before that pulse is reflected back to the sensor. The time needed for this path determines the 3D location of the point. Laser scanners are traditionally only used for smaller objects and do not apply well for large objects such as buildings. Both of these methods produce a very large set of independent 3D points and do not take into account the many natural constraints found in architecture.

Modeling Architecture

Several other techniques have been developed to model architecture from photographs. Cipolla and Robertson [6] exploit the properties of vanishing points to find the calibration parameters as well as the position and orientation of a camera. They take advantage of the parallel and orthogonal lines found in most buildings and use these to create a consistent coordinate axis. In projective geometry, parallel lines converge to what is known as a vanishing point. Since buildings exhibit many parallel and orthogonal lines along the three major coordinate axes, the corresponding vanishing points can be used to align the camera. The user of the system first marks a series of parallel and orthogonal lines. The vanishing points are defined as the intersection of the parallel sets of lines and provide three translationally independent, localized feature points that are used to solve a matrix representing the calibration parameters for the camera. Once the camera is calibrated and the images are aligned standard epipolar geometry constraints are used to find point correspondences. The system uses these point correspondences to fit a model of triangle meshes.

Other systems [1] are becoming more ambitious and are automatically optimizing models using the Hough transform for edge detection. The detected edges are then clustered to form groups of world parallel lines. These are then used as above to calibrate the camera and determine the model.

The use of piecewise planar models [10] which make use of the perpendicularity and verticality constraints found in architecture is another new technique used to automatically discover the architectural structure of a scene. In this method, feature detection is used to set up a mapping between images as well extract a series of 3D points. Optimized planes are randomly selected from groups of these 3D points and then checked to see if they fit the model using the Random Sample Consensus (RANSAC) estimation algorithm. The error is determined by projecting the planes into images and checking the intensity differences between the predicted values and the actual values. The best planes are selected and further optimized using a non-linear gradient decent algorithm. The use of planar constraints helps produce realistic models and removes the need for user input.

2. Camera Calibration

The majority of information used to reconstruct the models in the Façade system comes from the input images. In order to perform this reconstruction it is necessary to understand how the camera produced these images. The camera calibration parameters define a direct mapping between the 3D world coordinate system to the 2D image plane. This determines what a 3D object appears like in the photograph. The Façade system uses this mapping by first projecting a possible model onto the image and then checks if the projection is consistent with the actual image.

A pinhole camera model [3] is used to represent the settings for a given camera. This model has the following parameters:

$\mathbf{p_x}$,$\mathbf{p_y}$ define the principle pixel points for the camera. This is the optical center for the image where the pinhole projects onto the image plane. For most cameras this is located directly in the center of the image. However, this can change if the image is cropped or misaligned during scanning.

$\mathbf{f_x}$,$\mathbf{f_y}$ define the focal length (distance between the camera center and image plane) in pixels. The ratio between f_y and f_x defines the aspect ratio of the image. This is 1 for standard images but can vary if the pixels in the camera are not square. The calculated parameters for focal length are only correct if the camera maintains the same focus for all images using this calibration. Some cameras come with auto focus as well as a variable zoom lens. Each change of the auto focus and zoom lens changes the focal length of the camera so much care must be taken to keep these constant. This can be done by setting the zoom to a fixed position and the focus to infinity.

s defines the skew of the camera. This is the angle between the x and y axis for the image. This is 90 degrees for most cameras and is ignored in the calibration process.

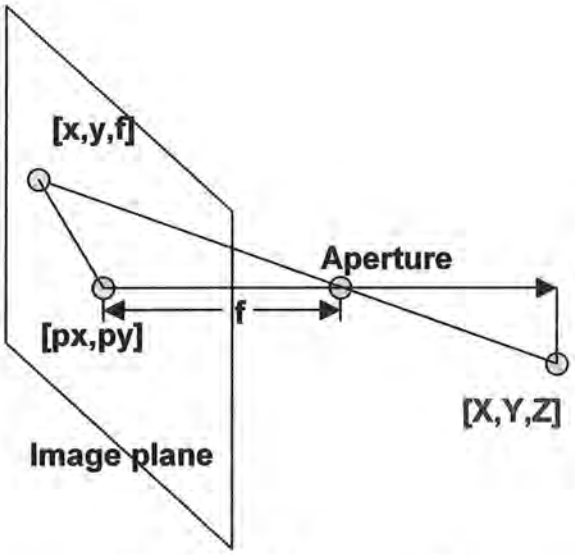

figure 2.1: pinhole camera model

A 3x3 matrix M_c can be use to produce the required mapping between 3D coordinates and the 2D pixel coordinates. Note that there is not a direct inverse mapping from the 2D coordinates back to 3D world coordinates since the z position of the object is lost. Each pixel coordinate defines a ray from the camera center to the object that produced its value.

$$M_c \cdot \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} f_x & s & p_x \\ 0 & f_y & p_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} x \\ y \\ Z \end{pmatrix}$$

Most cameras also differ from the ideal pinhole model and use a lens that is not infinitely small. This results in the blurring of objects that do not lie on the focal plane. Cameras can also carry radial distortions that cause straight lines to appear bowed within the image. These distortions are modeled with a 5-parameter polynomial and appropriate transformations can be applied to remove such distortion

$$\left(1 + k_1 r + k_2 r^2 + k_3 r^3 + k_4 r^4 + k_5 r^5\right)$$

My implementation of the Facade system does not perform the camera calibration directly. Instead the Matlab Camera Calibration Toolbox [3] should be run for all image sets. This toolbox finds and removes the radial distortion from images. The toolbox also

finds the other intrinsic camera parameters which can then be entered into the Façade system for each camera view as needed.

figure 2.2: camera calibration pattern

The camera calibration toolbox uses a standard checkerboard calibration pattern as a reference point. To calibrate a camera one takes several pictures of the calibration pattern at different angles and positions. As mentioned earlier, the internal parameters of the camera must be kept constant to insure proper calibration. Corner extraction is done on each of the reference images using the Harris corner detector [7] to locate the positions of all corners in the checkerboard grid. These corners are treated as feature points [11] and are used to setup a matrix defining the correlation from the real world checkerboard coordinates to the image coordinates. Given enough feature points an analytical solution to this matrix can be solved. The accuracy of the calculated values is limited due to image noise and radial distortion. An equation using the camera calibration parameters and the radial distortion is then used and optimized to minimize the error between the actual transform and the transform predicted by the parameters. This equation is treated as a least mean square problem and is optimized by modifying the parameters in a gradient decent search. The gradient decent is initialized with the analytical solution found earlier to minimize the chance of getting stuck on a local minimum.

We now have found the intrinsic parameters for the pinhole camera as well as the radial distortion. 3D points can now accurately be reprojected onto images if the rotation and position of the cameras are found accurately.

3. Optimization process overview

Once the input images are calibrated they can be used to create the desired models. The first step in creating a model requires decomposing the building into a series of blocks. These blocks are standard geometric shapes such as boxes, planes, and wedges. The optimization process finds the values for the parameters of these objects. Users can put constraints on these parameters to help limit the optimization's search space.

The user must also setup camera objects representing each input image. The camera objects define the input image, camera calibration parameters, as well as the location and orientation of the camera. The user must give a very rough guess of where the camera is located and which direction it is facing. The user also marks lines found in the images that correspond to the block edges defined earlier.

Model optimization is done by making educated guesses at what the various camera and block parameters should be. This guess is then checked by projecting the guessed geometry of the model onto the images. Projected edges of the model should line up with the edges drawn by the user. Based on how the edges line up the parameters are changed accordingly and the new guess is checked. This process repeats iteratively until it is deemed that a solution is found or the iteration count exceeds a specified number. The process of adding new images and new blocks does not need to be done all at once, blocks can be added and optimized one at a time. If the optimization process does not work, the user can choose to add further marked edges on the camera images. This allows a user to slowly build the model until it reaches a satisfactory level of accuracy.

4. Block Types

The models produced by the Façade system are built from a collection of primitive block types. The use of these blocks helps simplify the process of modeling architecture, because most buildings are built on a standard set of shapes with a lot of natural constraints such as parallel and orthogonal lines. Image-based modeling's ability to use the input images as texture maps also removes the necessity of modeling all the complex

detail found on most buildings. In other modeling approaches such as laser range scanning, shape from silhouette, and voxel coloring the exact model of the building is being built without knowledge of these standard primitives and this results in a lot of superfluous shapes due to error and noise. Restricting the system to primitive blocks produces cleaner models.

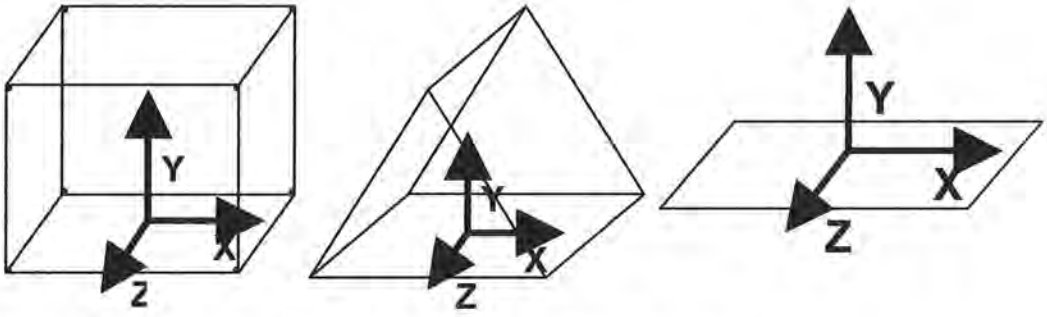

figure 4.1: box, wedge, and plane block types

In Façade, users can select from a list of block types. Currently the block types offered are box, wedge, and plane. A specially formatted XML file defines the geometry of these primitive block types. Other block types can be created with a block file using the XML format and placing it in the block type directory. The block type format helps define the geometry of the block as well as the variable parameters that are used to modify this geometry. The format is divided into 5 sections:

Block Parameters: A list of parameters for the block is given. These parameters determine the position of the vertices of the block geometry with respect to the block's local coordinate system. Blocks have an implied set of 6 extrinsic parameters that set the context for the block's local coordinate system. The first three define the block's orientation using Euler angles and the final three determine the blocks position.

Point Locations: This is a list of the vertices that define the geometry for a block. Each point is named and given a location based on the local parameters.

```
<Point>
```

```
        <PointName>P0</PointName>
        <PointLocation>[.5*width,0,.5*length]</PointLocation>
</Point>
```

This defines point P_0 which will be found at location (.5*width,0,.5*length) with respect to the blocks local coordinate system. The location of this point changes as the parameters of the block change. Point locations may be composed of a linear combination of their internal parameters. This is done to simplify the differentiation of point locations with respect to block parameters. This does not limit Façade's capability as most needed shapes can be defined using a linear combination of variables.

Extents: This helps define a bounding box around the geometry. The first value given defines the minimum values that any given point can take on in each of the coordinate axis. The second point given defines the maximum values that any given point can take on in each of the coordinate axis. These are used to simplify setting up constraints where a box sits on top of another or is directly adjacent to another block.

Line Locations: The location of each line in the block is defined using a pairing of two of the above points. The line name for a given block is used to create a correspondence with the user drawn lines on the input pictures.

Face Locations: The geometry for each face of the block is needed to visualize the actual models. A face is defined using a list of point names. The list should be given in standard counter clockwise order to determine which side of the polygon should be visible. Currently faces can only be composed of three or four points. If a given face has more points the user should break it up into triangles and add each of these triangles as a separate face.

With the above information a block with a given block type can fully define its geometry with respect to its own coordinate system when given values for its intrinsic parameters. It can further define its world geometry when given the values for its extrinsic rotation and translation parameters.

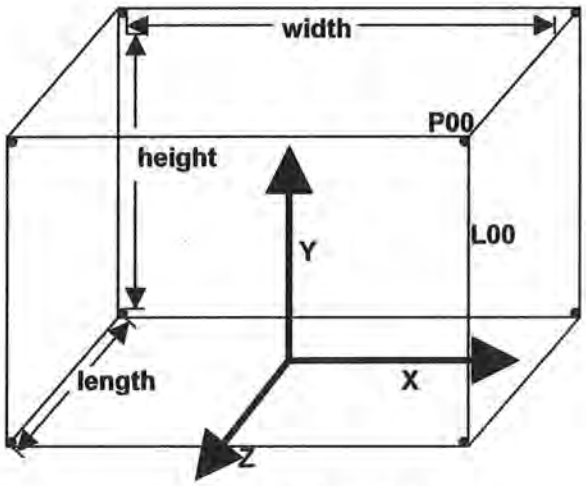

figure 4.2: box block type

The XML block type file format and an example block type defining the geometry for a standard box can be found in Appendix A. The box's geometry varies by the three internal parameters of length, width, and height.

5. Blocks

Façade users incrementally build their model by creating named instances of block objects. A block object consists of several parts.

Block Name: Used to identify the block. This name is needed for making block correspondences between other block parameters and user marked block edges.

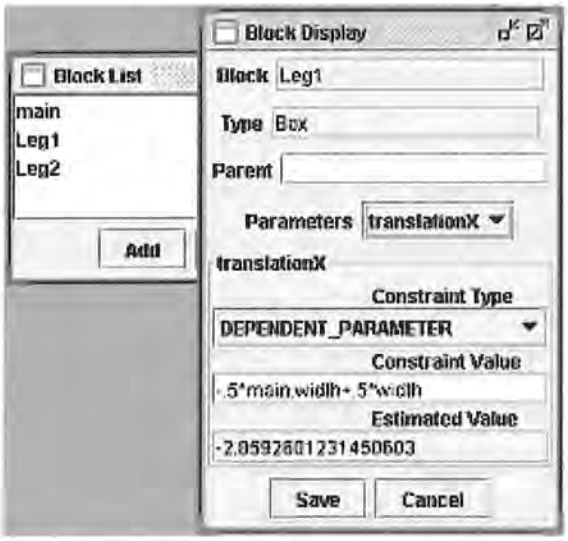

figure 5.1: block display dialog

Block Type: This specifies the geometry of the block.

Parameter Constraints: This lists all of the parameters for the block (both intrinsic and extrinsic). Each parameter has a name, constraint type, constraint value, and a current estimated value. There are three different constraint types and each determine the meaning of the given constraint value that is to be entered:

Free Parameter: The actual value of this parameter is unknown and will be determined by the optimization process. The constraint value given sets the current estimated value for the variable and will be used in the optimization initialization process.

Constant Parameter: The value for this parameter remains fixed throughout the optimization process. Since its value is already known it is never fed into the optimization process, however it is still needed to find the point positions for the corresponding block. The value given for the constraint value should be the numerical value for that parameter.

Dependent Parameter: The value for this parameter is limited to a linear combination of other parameters come from any block. This is useful when it is known

that two blocks have the same dimensions or if one block is known to be directly adjacent to another block. The requirement that this dependence be a linear combination is to aid in the calculation of the derivatives of how this block's point positions change with the perturbation of other block parameters. Since this block's point position is a linear combination of its own variables and its own variables can only be linear combinations of other variable it follows that a given block's point position is limited to a linear combination of block variables. This greatly simplifies the derivative equations needed for optimization. The constraint value given is a string representing the linear combination of variables that defines this variable's dependences. When using a parameter from another block, that block must be given followed by a period and then the parameter name.

Example Dependent constraint setting for blockA's length :

BlockA.length = .5*blockC.width+ .5*blockD.height

When performing the optimization, a given dependent value is never used directly. Instead its value is set using the variables it depends on. The value of the parameter is then used to reproject the guessed geometry to test its fitness.

Before the optimization process begins it is necessary to reduce the dependences of the parameter since the given dependencies may be other dependent variables. Since each dependence is simply a linear combination of other variables, this flattening can be done using a recursive algorithm. Using a simple depth first search one can determine if there are any cyclical dependencies. If a cycle is found, optimization must be aborted and the user is alerted to the problem.

Besides the intrinsic parameters defined in the block type, each block is also given 6 extrinsic parameters that define its position and orientation with respect to world coordinates. The rotational parameters represent Euler angle rotations in degrees. The block to world transform is applied as:

$$P_w = T \cdot R_z \cdot R_y \cdot R_x \cdot P_b$$

P_b = Block's local position

P_w = Block's world position

For my implementation of the Façade system the rotations must be set as constant constraints. This does not really limit the system because most block components generally have a known rotation such as 90 or 45 degrees. The translation parameters are treated any other block parameters in the system.

Parent Block:

Each block in the system also has the option of having a parent block. When a block is given a parent the blocks translation and rotational parameters are applied with respect to the parents local coordinate system. This means that any rotations or translations applied to the parent or the parent's ancestors are also applied to this block.

$$P_w = T_g \cdot R_g \cdot T_p \cdot R_p \cdot T_b \cdot R_b \cdot P_b$$

T_g,R_g = Grand parent's transform

T_p,R_p = Parent's transform

During the initialization phase a depth first search is done to make sure there are no cycles in a block's family tree.

Before optimization occurs the parameter constraints are processed. Only the free parameters are fed into the optimization. These free parameters form a large array of variables. Each block initializes its structure to keep track of the current location of all of its geometry points and how they vary according to dependence on other free parameters including variation due to a block's ancestors' transformation. Given the current value of the free parameter array a given block can report the position of each of its points (or lines) with respect to the world coordinate system. This is used to help reproject these points (or lines) back onto the reference images.

The 3D point location d of each vertex in a block's geometry with respect to the world coordinate system can be calculated as:

$$f(x_0 + \varepsilon) = f(x_0) + f'(x_0)\varepsilon + f''(x_0)\varepsilon^2 + \ldots$$

$R_{ax}|T_{ax}$ = Rotation and Translation for ancestor block x

$R_{ap}|T_{ap}$ = Rotation and Translation for parent block

$R_b|T_b$ = Rotation and Translation for block

P(x) = point defined by block parameters X (as found in the block type file)

The representation for a block edge v can be defined as a 3D vector and a point

Line $l = \langle v,p \rangle = \langle d_1 - d_2, d2 \rangle$

Where d_1 and d_2 are the two 3D coordinates for the points that define block edge l.

6. Cameras

Each image used for modeling in the Façade has a corresponding camera object. The camera object defines all of the camera's intrinsic and extrinsic parameters for the corresponding image taken. The intrinsic parameters are those found by calibrating the camera using the Matlab camera calibration toolbox and are directly entered by the user. In most cases the same calibration set will be used for all images. The extrinsic parameters define the position and orientation of the camera with respect to the world coordinate system. A rough initial guess must be entered by the user to make sure the camera is pointed in the correct direction. If this is not done, the camera will still reproject the geometry but may find the model parameters such as width and height to be negative. The rotation angles defining the camera's orientation are Euler angles. The transform is applied in the same manner as the block transforms. It is also possible to transform a point from world coordinates to a camera's coordinates as follows:

$$P_{camera} = R^{-1} (P_{world} - T)$$

One can further transform a point from 3D camera coordinates to the camera's 2D image plane using the camera calibration parameters found earlier. This is necessary when it comes time to check the accuracy of a model.

$$P_{image}(x,y) = M_c \cdot P_{camera}(X,Y,Z)$$

The Camera object allows for the user to mark edges found in the image that correspond with the primitive blocks specified earlier in the system. One perk of the Façade system is the use of lines to mark edge correspondences. Traditionally in computer vision, point markers are used to mark 3D correspondences, which can be difficult to mark accurately. Marking with lines is more forgiving to user error since a series of points are actually used. The lines marked by the user also do not need to begin or end at the actual end points of the line in the image. Instead, the equation of the line marked by the user is used to make the correspondence. The weight that the line carries in the optimization process is determined by the length of the line drawn by the user. Longer lines carry more weight which is beneficial since longer lines are easier to localize accurately. Lines are specified using the mouse to mark two pixels on the image. Before these pixel coordinates are used in the optimization process they must first be normalized to the camera coordinate system. This takes into account the camera calibration parameters found earlier. We find where these pixels would lie on an image plane set at z = -1.

$$z = -1$$

$$y = \frac{-(i_y - p_y)}{f_y}$$

$$x = \frac{(i_y - p_y) - s \cdot y}{f_x}$$

Where i_{xy} is the marked pixel location and x,y,z is the normalized pixel location in camera coordinates.

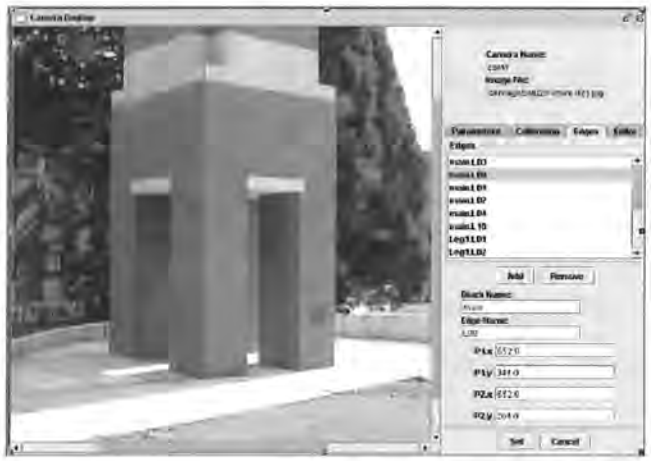

figure 6.1: camera edge marking

The Façade system also provides a fitting algorithm to help accurately place the lines drawn by users. The fitting algorithm takes a gradient-based approach based on the Intelligent Scissors algorithm [12]. When the original image is loaded it is preprocessed to find the image gradient in the x and y directions using convolution filters. The magnitude and direction of this gradient are then calculated. The Laplacian zero crossing for the image are found as well.

$$G_{mag} = \sqrt{x_g^2 + y_g^2} \qquad G_{orient} = \mathrm{Tan}^{-1}(y/x)$$

Pixels that are located on an image edge should have high gradient magnitude and be a zero crossing since an edge represents a drastic change in pixel intensity. Pixels along an edge should also have a gradient orientation that is similar to other pixels along that edge. We would like to traverse along our edge from our starting line point to our stopping line point. We can make a guess of this traversal by following what we think is the edge. This is done by placing weights between pixels. We define these weights such that the path from one pixel to then next is low if the gradient is high, the pixel is a Laplacian zero crossing, and the orientations are similar. We perform a shortest path search between the starting and stopping pixels. It is unlikely that very many pixels will need to be searched to find our destination since our line is presumably on a strong image edge. The pixel locations that are found along this shortest path are then used to find a line of best fit. This should locate the line to sub pixel accuracy. The seed points can then be

moved to the closest points that lie on this line. Since the seed points may not have been placed accurately the search can be performed again until the desired accuracy is reached. It may be the case that the given line does not lay on a strong image edge. The user has the option of skipping this fitting process or fitting more seed points to the line.

Once the seed points for the line are selected the user must enter in the corresponding block line that this line represents. This is done by entering in the block name as well as the line name as defined in the block type file. Users make line correspondences as they see fit as they run the optimization process. They may continue to add edge lines until the model's desired accuracy is reached.

7. Viewing the Model

There are several ways to view the accuracy of the model produced by Facade. The final numeric fitness values found by the optimization process may be used. There are three levels of optimization which are explained later in the paper. Each produces a metric based on how well the geometry of the model matches the lines marked by the user. After each optimization, the number of iterations taken, the final error value, and the reason the optimization was halted is outputted to the user. The actual numbers outputted only make sense with experience but the reason the optimization process was halted is usually a good indicator.

The second method to view the accuracy of the model involves the camera views. After each optimization run the final model found is reprojected back onto the input images as a wire frame drawing. The user can check to see if the model's wire frame properly aligns with the block edges found in the images. This gives a good idea of the accuracy of the model.

figure 7.1: projected model wire frame

The third method is to use the 3D model viewer. The final model optimized by the system is placed in a 3D virtual environment. The user can navigate throughout the environment to check out the model's geometry. A list of the various camera angles and positions is also given to allow the user to jump to the view that a given image was taken at. A VRML file is exported that represents the model and can be explored using a standard VRML viewer.

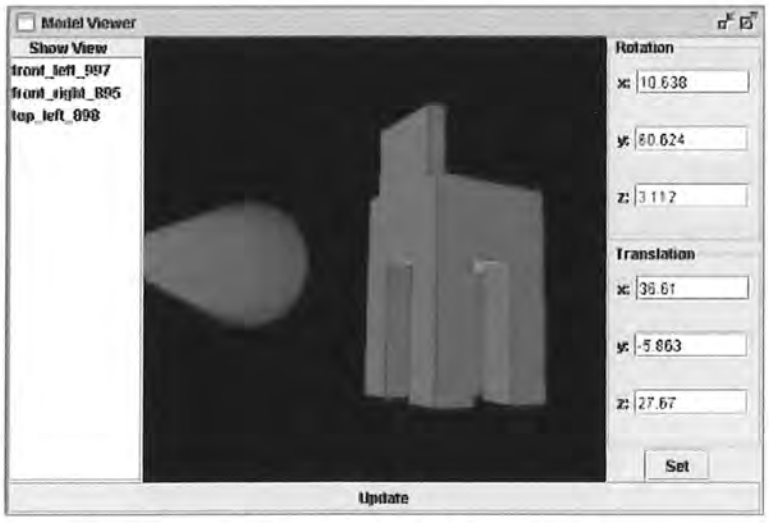

figure 7.2: model viewer of partially reconstructed tower

8. Model Optimization

So far the user has done most of the work in finding the model. We take advantage of the user's high-level expertise in decomposing the model into blocks, providing constraints, and providing a way to test the accuracy of the model. Now the computer steps in and performs the repetitive task of optimizing all of the parameters to produce a model that best fits the lines marked by the user. The model optimizer finds the values of all of the free block parameters (other constrained parameters can be calculated from these) as well as the position and orientation of each of the cameras. To estimate these values the Façade system uses an intelligent guess and check method. A guess is made at what the free parameter values should be. These values are fed into the various block objects which then determine the location of all of the block points. The location of block edges can be found from these block points. Each 3D edge that has a corresponding edge marked in the input images is projected back into the camera's image space. This projection is then checked to see how well it matches the line drawn by the user. The level of difference between these lines defines our error function, which then is differentiated with respect to each free parameter giving the direction in which we should modify parameters in order to reduce our error. The process repeats until we have reduced our error to a desired level, exceed the iteration limit, or found a local minimum. The final free parameters are fed back to the blocks and cameras and determine the final model.

Line Reprojection

Our first task is to reproject block edge e_i of our model back into the input images. The intersection of the camera's image plane P_{im} with the plane P_c formed by e_i in camera$_j$'s coordinates and the camera$_j$'s origin provides this intersection [5]. We can define the equation for plane P_c using normal vector m_{ij}.

$$m_{ij} = R_j^{-1}(v_i \times (d_i - t_j))$$

where v_i and d_i for block i come from equations defined in section 5 and R_j and t_j are the transformations from camera j space to world space.

$$P_c \rightarrow m_x \cdot x + m_y \cdot y + m_z \cdot z = 0$$

We now must find the intersection between P_c and P_{im}. As part of the camera calibration process all images are normalized such that the image plane lies at z equal to –1 and thus

$$P_{im} \rightarrow z = -1$$

The intersection of these to planes yields the line

$$m_x \cdot x + m_y \cdot y - m_z = 0$$

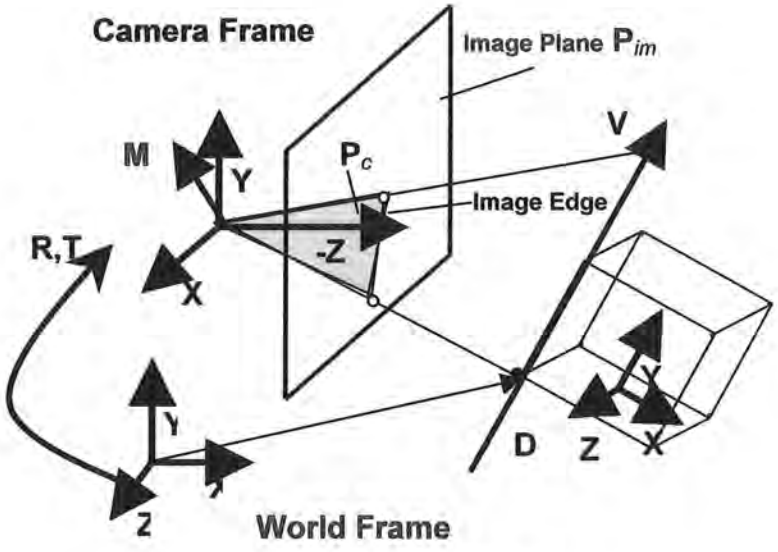

figure 8.1: block reprojection

We now must compare this line to the line marked earlier by the user which is defined by the points

$$\{(x_1,y_1,-1), (x_2,y_2,-1)\}$$

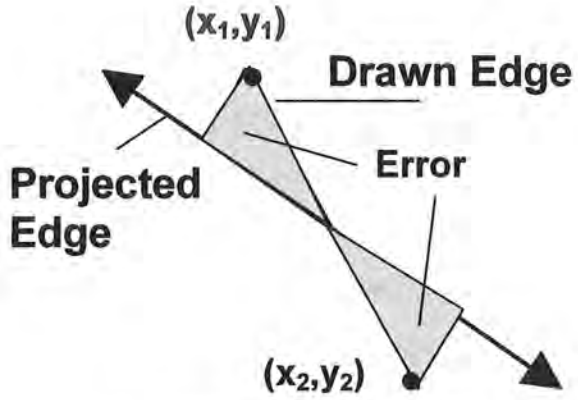

figure 8.2: block reprojection error

The error function [5] for the shortest distance between a point on the observed line and a point on the projected line can be parameterized by a scalar variable s.

$$h(s) = h_1 + s\frac{h_2 - h_1}{l}$$

$$h_1 = \frac{m_x \cdot x_1 + m_y \cdot y_1 - m_z}{\sqrt{m_x^2 + m_y^2}}$$

$$h_2 = \frac{m_x \cdot x_2 + m_y \cdot y_2 - m_z}{\sqrt{m_x^2 + m_y^2}}$$

$$l = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

We then define our error function for one marked edge as

$$Err = \int_0^l h^2(s)ds = \frac{l}{3}(h_1^2 + h_1 h_2 + h_2^2) = m^T(A^T BA)m$$

$$m = (m_x, m_y, m_z)^T$$

$$A = \begin{pmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{pmatrix} \qquad B = \frac{l}{3*(m_x^2 + m_y^2)}\begin{pmatrix} 1 & .5 \\ .5 & 1 \end{pmatrix}$$

The true error function is the summation of the error function for each marked edge.

We now have the non-linear function we wish to optimize. We minimize this function using a variant of the Newton-Raphson gradient search technique taken from [5]. The search is done by approximating our error function with the first three terms of its Taylor series expansion evaluated at our current estimate. This requires the gradient as well as Hessian of the error function with respect to every camera and free block parameter. We desire an error of zero so we find the increment step necessary to give us this using our approximated function. Differentiating the above equation remains simple due to the fact that we only allowed our block geometry vertexes to vary by a linear combination of parameters. Single derivatives of these parameters yield a constant. Double derivatives of these parameters yield zero. At each iteration step each model edge that has a corresponding marked image edge evaluates its error. These are added together to produce the actual error function. The gradient and Hessians of this function are then calculated with respect to each free parameter and are given to the optimization process. The optimizer then calculates an update step that needs to take place for each free parameter. This update is fed back to the various blocks and cameras which take care of the change. The process repeats until the error function is sufficiently minimized, a local minimum is reached, or the maximum iteration limit is exceeded.

One of the more difficult parts of this optimization process is the involvement of the camera rotations. The set of rotation matrices specified by Euler angles is in the SO(3) lie group [4] which is not isomorphic with $\Re^n$. This means there exists singularities when dealing with gradients of the rotation which do not allow us to directly use the rotations in the optimization process. Instead we must parameterize these rotations to a system that is isomorphic to $\Re^n$ in the local neighborhood of R_0 (our current rotation).

$$R(\omega) = R_0 \exp\{J(\omega)\} \quad \omega \in R^3, \sqrt{\omega^t \omega} < \pi$$

$$J(\omega) = \begin{pmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{pmatrix}$$

All rotations for cameras and blocks are represented as quaternions. When a free rotation variable needs to be differentiated its parameterized form is used instead. The above equation can be differentiated as so:

$$\frac{d}{d\omega_x}(R_0 \exp\{J(\omega)\})\Big|_{w=0} = R_0 J(\hat{x})$$

The incremental step found by the optimization algorithm must be unparamaterized back to a quaternion.

$$q_s = (\cos(\theta/2), (\sin(\theta/2)/\theta)\ \omega)$$

The rotation can then be updated by:

$$q_0 = q_0 \cdot q_s$$

Since this is a nonlinear function it is possible that the optimization process can get stuck in a local minima. To prevent this we must provide the optimization process with a good initial guess to the solution. We first optimize 2 other error functions to provide this good guess. Going back to the definition of our original error function we can see that vector m is orthogonal to the rotated block edge v. This means their dot products should be zero giving:

$$m_{ij}^T R_j^{-1} v_i = 0$$

The line drawn by the user gives us the predicted value for m'_{ij}:

$$m'_{ij} = (x_1, y_1, -1)^T \times (x_2, y_2, -1)^T$$

Since our block rotations are constrained to constants, most block edge values v are known. With our predicted value for m'_{ij} we can optimize the equation above to get the initial rotation values for all of the cameras. The actual equation optimized is

$$0_1 = \sum_i \sum_j \left(m'^T R_j^{-1} v_i \right)^2$$

The next goal is to establish an estimated value for the camera translations and block parameters. We note that the vector between the cameras origin and block edge points are also orthogonal meaning their dot products should be 0. This allows us to define our second error function as:

$$0_2 = \sum_i \sum_j \left(\left(m'^T R_j^{-1} \left(P_i(x) - t_j \right) \right)^2 + \left(m'^T R_j^{-1} \left(Q_i(x) - t_j \right) \right)^2 \right)$$

P(x) and Q(x) are the location of the block edge endpoints based on the current guesses for the block parameters x. This equation helps predict the values of all of the free block parameters and the camera translations.

By optimizing the above two equations we have a very good guess as to what the final models should look like. We then perform our full optimization on the error function presented earlier to determine our final model.

9. Results

The results of my implementation of the Façade system are promising. The projected wire frame models match the input images well but there are definitely some pixel correlation errors. I am also disappointed with amount of line segments that are needed to be marked to produce good optimizations. It is difficult to tell if this is a limitation of my system or if the results provided by Debevec are overly optimistic.

One must be careful when relying only on one input image. The wire frame results can be misleading and appear to match the input image perfectly. This often is just an illusion of the projective geometry since there may be several combinations of translations and size parameters that can produce the same images which results in a model that matches the image but does not match the real geometry. One must be sure to use images from multiple views to properly constrain the parameters.

The initial camera rotation estimation step is very good. The further optimization steps change these values very little. Most of the model optimization is done in the second initialization phase where the block parameters and camera translations are calculated. This generally requires tens of thousands of iterations to minimize. The final optimization stage takes very few (5-50) iterations to reach its minimum. This is good because it is the most computationally expensive. Optimization times run between a couple of seconds and a couple of minutes depending on the complexity of the model and the number of edges marked.

Pape Memorial Tower

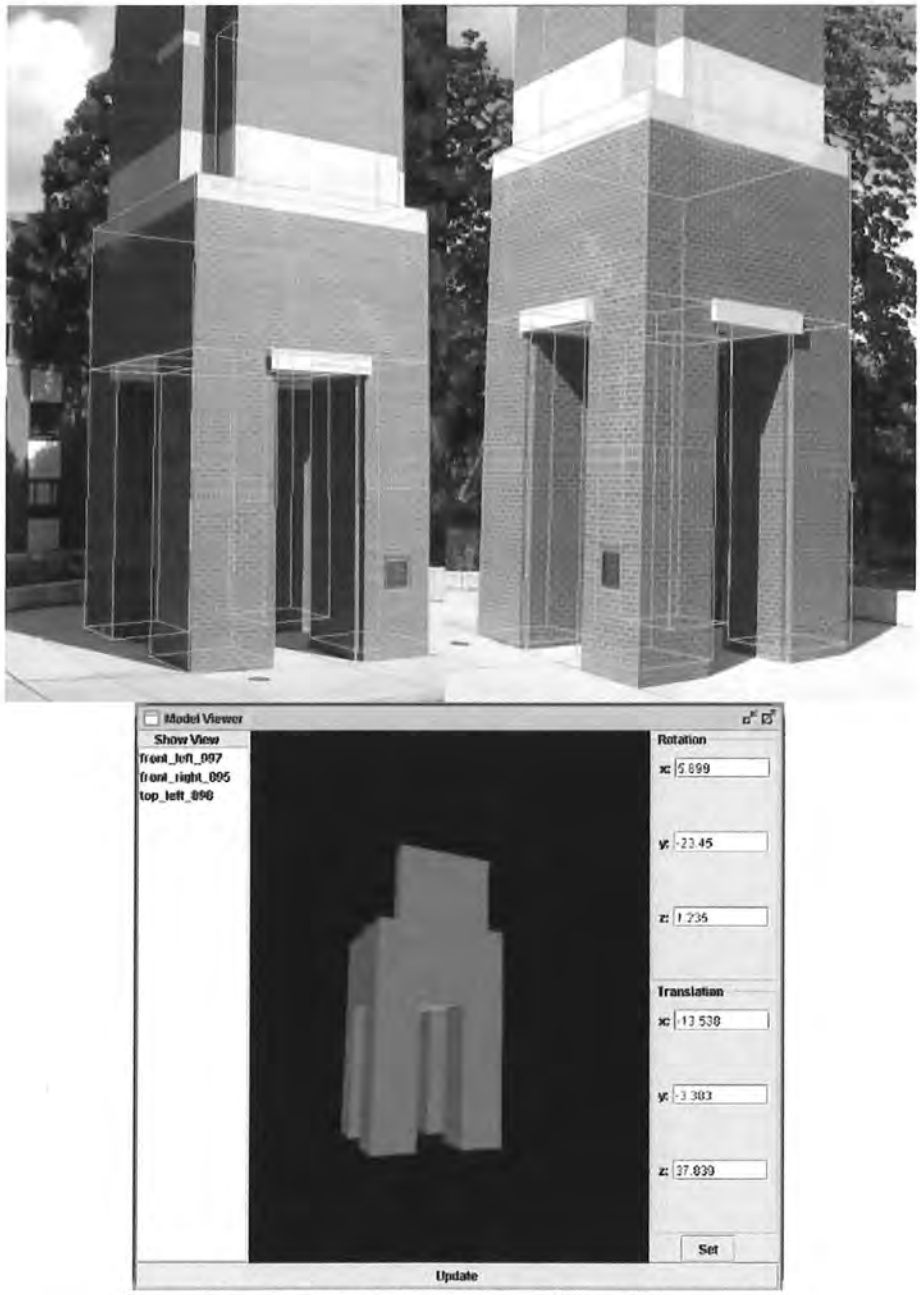

figure 9.1 a,b,c: partial reconstruction of library tower

The projected lines match the image edges very well. The discrepancies at the bottom of the tower legs is due to the fact that the tower is on slanted ground but the tower legs were all constrained to be the same size in order to have a symmetric model. This model was composed of 6 blocks, 54 block parameters, 18 camera parameters, and required 28 marked edges. Due to the constraints in the model there were only 8 free block parameters. The tower legs were all constrained to be of the same size and to be placed

symmetrically below the main block. It required 15 minutes to draw the line correspondences and to specify the block constraints. The total optimization time took 45 seconds.

Waldo Hall Tower

figure 9.1 a,b,c,d: partial reconstruction of Waldo Hall tower

The projected wire frame image once again matches closely to the real image edges. There are places where the lines deviate from the true edges by a couple of pixels. The most notable error is the front right corner on the second image. This model was composed of 3 blocks, 24 block parameters, 18 camera parameters, and required 24 marked edges. Note that the small block between the two main blocks has a larger width and length than the others which is true of the real structure. Due to the constraints in the

model there were only 8 free block parameters. The y translations of the blocks were constrained to produce a stacking of blocks. It required 20 minutes to draw the line correspondences and to specify the block constraints. The total optimization time took 2 minutes, which was much greater than the previous example. This is perplexing since the models share the same relative complexity.

10. Further Work

The process described so far is only the first stage in Debevec's Façade system. To accurately model a building one needs more than just the shape. Debevec goes on to realistically texture map each of the primitive block's faces. This is done in a two-part process. First a depth map is found for each of the primitive's faces using a process of model-based stereo. As mentioned before, the standard stereo correspondence problem can be difficult to solve due to foreshortening and occlusions, however this process is simplified by the use of the predicted model. The various images are reprojected onto the model removing all foreshortenings but still maintaining the principles of epipolar geometry. The task of finding point correspondences becomes much easier. These depth maps help model much of the complexity missed by the actual primitives. Each face can then be accurately texture mapped using combinations of input images. Debevec uses a process of view dependent rendering where the texture map changes depending on the location of the viewer. Images that were taken more closely to the viewer's current position are weighted more heavily than the others.

The photometric modeling system presented in this paper could also be further improved. The system still requires a lot of input from the user. Marking block edges and setting the correspondences is a tedious process. One suggestion is to provide a tool allowing the user to drag a wire frame version of the block onto the image. The user then makes a rough estimate of where the corners of that block should be positioned. The block lines are then fit to the 2D image using the gradient information found in the image. The user could then remove the lines that are not visible in the image or better fit the ones that are visible if necessary. This would greatly decrease the amount of input required by the user.

Work can also be done to automatically find the existence of blocks in images using feature point extraction. If the user were to specify whole block location such as explained above we would have an idea of what each block face looks like. We then can perform feature point detection on those faces. The other images can then be searched for these same features. This could be used to predict the existence of the block in other images as well. Combing this information with the Hough transform such as in [1] may lead to the automatic detection of the lines as well as setting the proper edge correspondences. The user would then only be required to add extra block constraints to limit the amount of necessary work for the optimizer.

The system can also benefit from the implementation of features of other modeling programs. The use of the Hough transform to find vanishing points such as in [6] could reduce or even remove the need to have calibrated images. When the user marks a block consistent with a box a search can be done for vanishing points. The constraints given by the user could then be used in conjunction to solve for some of the camera calibration parameters.

Conclusions

The Façade system is a powerful tool for modeling architecture. Its expertise can be attributed to its proper division of labor between the user and the computer as well as its exploitation of constraints found naturally in architecture. However, the continued addition of features to automate the model creation process will greatly aid in its usability allowing people to simply and quickly create photorealistic 3D virtual environments.

Bibliography

[1] M. Antone and S. Teller, "Automatic recovery of relative camera rotations for urban scenes.," in *CVPR*, 2000, pp. 282--289.

[2] F. Arman and J.K. Aggarwal. "Model-based object recognition in dense-range images – a review". *ACM Computing Surveys*, 25(1):5–43, 1993.

[3] Camera Calibration Toolbox for Matlab. 2004. http://www.vision.caltech.edu/bouguetj/calib_doc/.

[4] Camillo J. Taylor and David J. Kriegman. "Minimization on the lie group so(3) and related manifolds". Technical Report 9405, Center for Systems Science, Dept. of Electrical Engineering, Yale University, New Haven, CT, April 1994.

[5] Camillo J. Taylor and David J. Kriegman. "Structure and motion from line segments in multiple images". *IEEE Trans. Pattern Anal. Machine Intell.*, 17(11), November 1995.

[6] R. Cipolla, D. Robertson and E. Boyer, "PhotoBuilder - 3D models of architectural scenes from uncalibrated images". In *Proc. IEEE Int. Conf. on Multimedia Computing and Systems*, Firenze, volume I, pages 25-31, June 1999.

[7] C. Harris and M. Stephens, "A combined corner and edge detector", *Fourth Alvey Vision Conference*, pp.147-151, 1988.

[8] Debevec, P. "Rendering Synthetic Objects into Real Scenes: Bridging Traditional and Image-Based Graphics with Global Illumination and High Dynamic Range Photography". In *SIGGRAPH 98*, July 1998.

[9] Debevec, P., Taylor, C. & Malik, J. 1996. "Modelling and Rendering Architecture from Photographs: A Hybrid Geometry- and Image-Based Approach", *Proc. SIGGRAPH.* 11.20.

[10] A. Dick, P. Torr, and R. Cipolla. "Automatic 3d modelling of architecture". *British Machine Vision Conference*, Bristol, pages 372–381, 2000.

[11] Heikkila J., Silven O., "A four-step camera calibration procedure with implicit image correction,'' *Proc. CVPR '97*, IEEE, 1997, pp. 1106-1112.

[12] E. N. Mortensen and W. A. Barrett, "Intelligent Scissors for Image Composition," in *Proc. of the ACM SIGGRAPH '95: Computer Graphics and Interactive Techniques*, pp. 191-198, Los Angeles, CA, Aug. 1995.

[13] Daniel Scharstein and Richard Szeliski. "A taxonomy and evaluation of dense twoframe stereo correspondence algorithms". Technical Report MSR-TR-2001-81, Microsoft Corporation, Redmond, WA 98052, USA, November 2001.

[14] S. M. Seitz and C. R. Dyer, "Photorealistic Scene Reconstruction by Voxel Coloring", *Proc. Computer Vision and Pattern Recognition Conf.*, 1997, 1067-1073.

[15] R. Szeliski, "From images to models (and beyond): a personal retrospective", *Vision Interface '97*, Kelowna, British Columbia, Canadian Image Processing and Pattern Recognition Society, pp. 126-137 (1997).

[16] Zhengyou Zhang, "Flexible Camera Calibration by Viewing a Plane from Unknown Orientations". *ICCV 1999*: 666-673

Appendix A

Block Type Template

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
- <BlockTypes>
  - <BlockType>
      <BlockTypeName>Box</BlockTypeName>
      <NumParameters>3</NumParameters>
    - <Parameters>
      - <Parameter>
          <ParameterName>width</ParameterName>
        </Parameter>
      - <Parameter>
          <ParameterName>height</ParameterName>
        </Parameter>
      - <Parameter>
          <ParameterName>length</ParameterName>
        </Parameter>
      </Parameters>
      <NumPoints>8</NumPoints>
    - <Points>
      - <Point>
          <PointName>P0</PointName>
          <PointLocation>[.5*width,0,.5*length]</PointLocation>
        </Point>
      - <Point>
          <PointName>P1</PointName>
          <PointLocation>[.5*width,height,.5*length]</PointLocation>
        </Point>
      - <Point>
          <PointName>P2</PointName>
          <PointLocation>[-
            .5*width,height,.5*length]</PointLocation>
        </Point>
      - <Point>
          <PointName>P3</PointName>
          <PointLocation>[-.5*width,0,.5*length]</PointLocation>
        </Point>
      - <Point>
          <PointName>P4</PointName>
          <PointLocation>[-.5*width,0,-.5*length]</PointLocation>
        </Point>
      - <Point>
          <PointName>P5</PointName>
          <PointLocation>[-.5*width,height,-
            .5*length]</PointLocation>
        </Point>
      - <Point>
          <PointName>P6</PointName>
          <PointLocation>[.5*width,height,-
            .5*length]</PointLocation>
        </Point>
      - <Point>
          <PointName>P7</PointName>
```

```
      <PointLocation>[.5*width,0,-.5*length]</PointLocation>
    </Point>
  </Points>
- <Extents>
    <MinExtent>[-.5*width,0,-.5*length]</MinExtent>
    <MaxExtent>[.5*width,height,.5*length]</MaxExtent>
  </Extents>
  <NumLines>12</NumLines>
- <Lines>
  - <Line>
      <LineName>L00</LineName>
      <LineLocation>[0,1]</LineLocation>
    </Line>
  - <Line>
      <LineName>L01</LineName>
      <LineLocation>[1 ,2]</LineLocation>
    </Line>
  - <Line>
      <LineName>L02</LineName>
      <LineLocation>[2, 3]</LineLocation>
    </Line>
  - <Line>
      <LineName>L03</LineName>
      <LineLocation>[3, 0]</LineLocation>
    </Line>
  - <Line>
      <LineName>L04</LineName>
      <LineLocation>[4 ,5]</LineLocation>
    </Line>
  - <Line>
      <LineName>L05</LineName>
      <LineLocation>[5 ,6]</LineLocation>
    </Line>
  - <Line>
      <LineName>L06</LineName>
      <LineLocation>[6 ,7]</LineLocation>
    </Line>
  - <Line>
      <LineName>L07</LineName>
      <LineLocation>[7 ,4]</LineLocation>
    </Line>
  - <Line>
      <LineName>L08</LineName>
      <LineLocation>[0 ,7]</LineLocation>
    </Line>
  - <Line>
      <LineName>L09</LineName>
      <LineLocation>[1 ,6]</LineLocation>
    </Line>
  - <Line>
      <LineName>L10</LineName>
      <LineLocation>[2,5]</LineLocation>
    </Line>
  - <Line>
      <LineName>L11</LineName>
```

```
            <LineLocation>[3 ,4]</LineLocation>
        </Line>
    </Lines>
    <NumFaces>6</NumFaces>
  - <Faces>
    - <Face>
        <FaceName>Front</FaceName>
        <FaceLocation>[0,1,2,3]</FaceLocation>
      </Face>
    - <Face>
        <FaceName>Back</FaceName>
        <FaceLocation>[4,5,6,7]</FaceLocation>
      </Face>
    - <Face>
        <FaceName>Right</FaceName>
        <FaceLocation>[7,6,1,0]</FaceLocation>
      </Face>
    - <Face>
        <FaceName>Left</FaceName>
        <FaceLocation>[3,2,5,4]</FaceLocation>
      </Face>
    - <Face>
        <FaceName>Top</FaceName>
        <FaceLocation>[1 ,6 ,5, 2]</FaceLocation>
      </Face>
    - <Face>
        <FaceName>Bottom</FaceName>
        <FaceLocation>[3,4,7,0]</FaceLocation>
      </Face>
    </Faces>
  </BlockType>
</BlockTypes>
```