New Algorithms for Modeling and Rendering Architecture from Photographs

by

Georgi Dobrinov Borshukov

B.S. (University of Rochester) 1995

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Master of Science

in

Electrical Engineering and Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Jitendra Malik, Chair
Professor David Forsyth

Spring 1997

The dissertation of Georgi Dobrinov Borshukov is approved:

Chair Date

Date

Date

University of California at Berkeley

Spring 1997

New Algorithms for Modeling and Rendering Architecture from Photographs

by

Georgi Dobrinov Borshukov

Abstract

New Algorithms for Modeling and Rendering Architecture from Photographs

by

Georgi Dobrinov Borshukov

Master of Science in Electrical Engineering and Computer Science

University of California at Berkeley

Professor Jitendra Malik, Chair

Façade is a system developed by Paul Debevec, C. J. Taylor, and Prof. Jitendra Malik for modeling and rendering 3-D architectural models from photographs. Until recently, the reconstruction capabilities of Façade were limited to polyhedral structures such as boxes, frustums, octagons, etc. The rendering algorithms were computationally expensive. Façade needed tools for recovery of some non-polyhedral structures appearing in architecture. The first part of this work describes a new set of tools for recovery of surfaces of revolution and arches which made a project such as the reconstruction of the majestic Taj Mahal from a single uncalibrated photograph possible. The second part, describes the significant advances made in image-based rendering. New robust ways of performing view-dependent texture mapping are introduced. The novel rendering algorithms allowed us to produce completely photorealistic renderings of flying around the Berkeley tower, the Campanile, and the surrounding campus at speeds close to real-time.

Professor Jitendra Malik
Dissertation Committee Chair

To my family,

My Father Dobrin, My Mother Sneji, and My Sister Margi

Borshukovi

Contents

List of Figures

Acknowledgements

I want to thank Paul Debevec, Yizhou Yu, C.J. Taylor, Carl Korobkin, Jitendra Malik, Jason Luros, Vivian Jiang, Chris Wright, and Sami Khoury for making the Campanile movie project happen. I also want to thank Ismail Eldumiati, Randy Chung, and Rockwell Semiconductor Systems for their invaluable help and support.

Chapter 1

Reconstruction

Originally, the reconstruction capabilities of Façade presented in [2, 3] which were based on the work in [14] were limited to polyhedral structures such as boxes, frustums, octagons, etc. Façade definately needed tools for recovery of some non-polyhedral structures commonly appearing in architecture. One class of such structures are surfaces of revolution (SORs) demonstrated in columns, domes, minarets, etc. Another common class are arches. In general, the problem of reconstructing such objects could be extremely complex, and there is a significant amount of computer vision literature addressing the issue [4, 5, 8, 11, 13, 15]. However, in the context of architectural scenes where the camera positions and the proportions of polyhedral structures could be recovered robustly through the minimization algorithm from [14] inside Façade, the problem of arch and surface of revolution recovery is considerably simplified. After looking at different approaches that did not seem to fit our particular goals well, a simple and elegant solution was found. Façade now has features that made a project such as the reconstruction of the Taj Mahal from a single photograph

found on the Internet possible (see Fig. 1.5). Our camera model is described in section 1.1. Details about the recovery of arches can be found in section 1.2. Details about the recovery of surfaces of revolution can be found in section 1.3.

1.1 Camera Model and Coordinate System Transformations

The reconstruction methods described in the following sections are based on using the camera model from Fig. 1.1. We assume that the image has been properly undistorted and the camera calibrated using the pre-existing algorithms of Façade. Therefore, the focal length f and the center of the image plane (u_0, v_0) in pixels are known. Also the camera position in the world coordinate system defined by the rotation matrix $\mathbf{R^C}$ and the translation vector $\mathbf{T^C} = [\ T_x^{\mathbf{C}} \quad T_y^{\mathbf{C}} \quad T_z^{\mathbf{C}}\ ]^T$ have been previously reconstructed by the minimization algorithm. With these quantities, we can easily obtain the camera coordinates $\mathbf{p}$ of a point with world coordinates $\mathbf{p^W} = [\ x^W \quad y^W \quad z^W\ ]^T$ with the transformation

$$\mathbf{p} = \mathbf{R^C}(\mathbf{p^W} - \mathbf{T^C}) \tag{1.1}$$

and vice versa

$$\mathbf{p^W} = [\mathbf{R^C}]^{-1}\mathbf{p} + \mathbf{T^C} \tag{1.2}$$

With the above information we can also convert an image measurement $\mathbf{p_i^I} = (x_i^{\mathbf{I}}, y_i^{\mathbf{I}})$ to a point in the camera coordinate system $\mathbf{p}_i = [\ x_i \quad y_i \quad -1\ ]^T$ with the equations:

$$\begin{aligned} x_i &= (x_i^{\mathbf{I}} - u_0)\frac{1}{f} \\ y_i &= (y_i^{\mathbf{I}} - v_0)\frac{1}{f} \end{aligned} \tag{1.3}$$

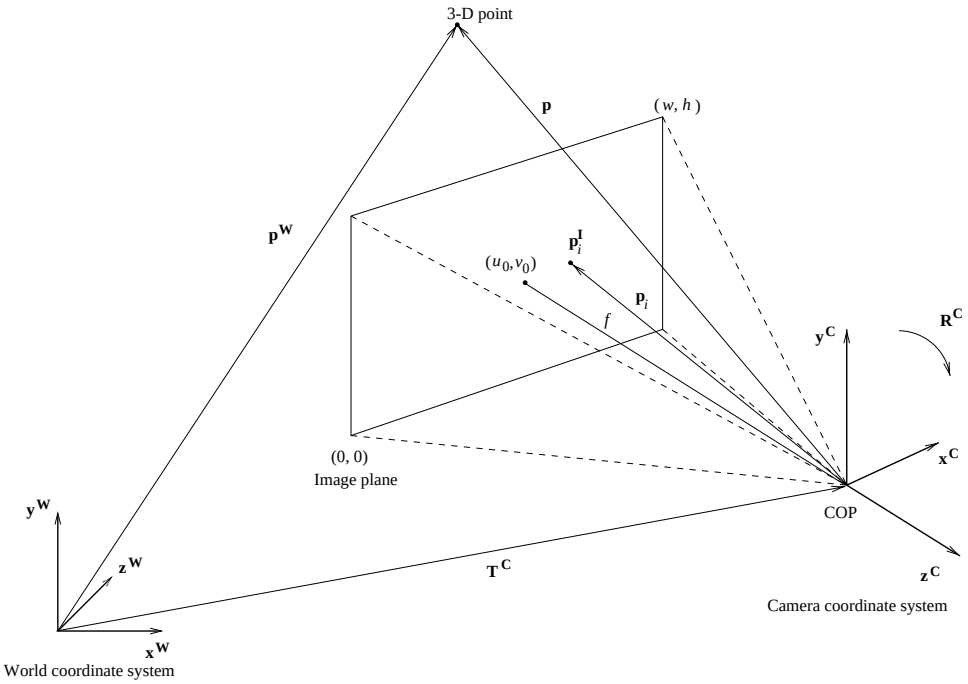


Figure 1.1: Camera model

1.2 Arches

1.2.1 Derivation

The arch is initially created as a rectangular *arch box* block for which the parent and relation are specified. Then, its width, depth, and height are reconstructed by the minimization algorithm. We know image points like $\mathbf{p}_i^{\mathbf{I}} = (x_i^{\mathbf{I}}, y_i^{\mathbf{I}})$ marked by the user that lie on the arch contour in the image plane. These points are expressed in camera coordinates according to equation 1.3.

Now $\mu_i \mathbf{p}_i$ are rays from the camera's COP passing through the marked points in the image (see Fig. 1.3). These rays intersect the face of the *arch box* where the arch begins at points $\mu_{i0}\mathbf{p}_i$. To find these intersections, i.e. the values of μ_{i0}, we use the face normal

$\mathbf{n^W}$ and a point $\mathbf{P}_c$ on the arch face (the middle point of the bottom edge) with world coordinates $\mathbf{p}_c^{\mathbf{W}}$ and camera coordinates $\mathbf{p}_c$ obtained by equation 1.1. The points $\mu_{i0}\mathbf{p}_i$ lie on the face, therefore, their distances from the face are:

$$[\mu_{i0}\mathbf{p}_i - \mathbf{p}_c]^T(\mathbf{R^C n^W}) = 0 \tag{1.4}$$

which gives:

$$\mu_{i0} = \frac{\mathbf{p}_c^T(\mathbf{R^C n^W})}{\mathbf{p}_i^T(\mathbf{R^C n^W})} \tag{1.5}$$

We need to rotate the vectors $(\mu_{i0}\mathbf{p}_i - \mathbf{p}_c)$ back into world coordinates to obtain the desired vectors $\mathbf{r}_i^{\mathbf{W}}$:

$$\mathbf{r}_i^{\mathbf{W}} = [\mathbf{R^C}]^{-1}(\mu_{i0}\mathbf{p}_i - \mathbf{p}_c) \tag{1.6}$$

The algorithm uses the projections r_i and h_i of these vectors onto the bottom edge and the middle axis of the arch face to automatically generate the arch surface.

1.2.2 Reconstruction Steps

The actual procedure for reconstructing an arch within Façade goes as follows:

1. First, the user selects *Arch Box* from the *Add Block* submenu of the *World Viewer*. The program includes an *arch box* block (a box with 2 extra edges marking the arch faces).

2. The *arch box* position and dimensions are then reconstructed with Façade's existing algorithms after specifying the necessary line correspondences.

3. The user selects an image and then marks the points $p_i^{\mathbf{I}}$ on the occluding contour of the arch in this image with Façade's *point marquee* tool.

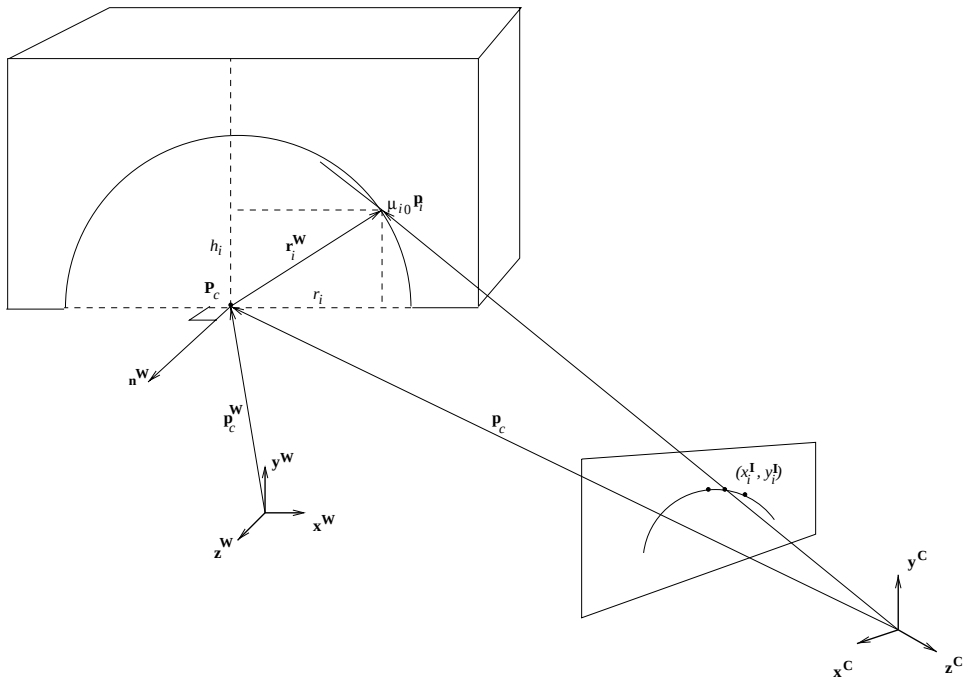


Figure 1.2: Geometry of the arch reconstruction method.

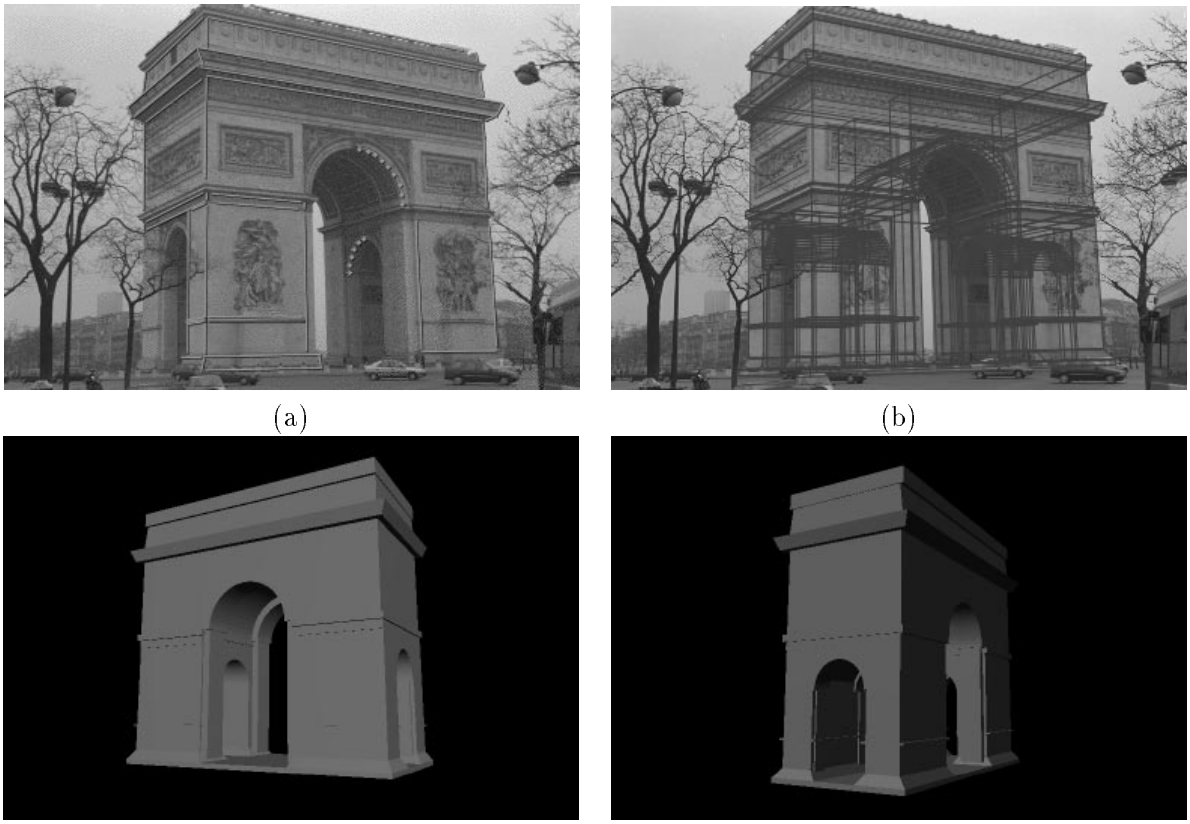
4. In order to prevent the algorithm from using points marked on another contour, the user surrounds the points to be used in the reconstruction with Façade's *quad* tool.

5. Then the *Arch Box* option from the *Reconstruct* submenu is selected. This invokes the algorithm described in the previous section. For each marked point $p_i^{\mathbf{I}}$ within the *quad* it finds the values r_i and h_i.

6. These values are used for procedural generation of an arch block following the block file syntax. The block file is saved on disk and then automatically included in the *World Viewer* replacing the old *arch box*.

1.2.3 Results

Fig. 1.3 shows the results of reconstructing a 3-D model of the Arc de Triomphe using the new arch recovery tools.



(a) (b)

(c) (d)

Figure 1.3: Model of the Arc de Triomphe demonstrating the new arch recovery capabilities of Façade. **(a)** One of three photographs used to reconstruct the Arc de Triomphe, with marked features indicated. **(b)** Reconstructed model edges projected into the original photograph. **(c)** Recovered model of the Arc de Triomphe. **(d)** Another view of the recovered 3-D model.

1.3 Surfaces of Revolution

1.3.1 Derivation

The method that we use for reconstructing a surface of revolution assumes that its central axis is known. This means that we need a point on the axis $\mathbf{p}_b^{\mathbf{W}}$ (usually the base point). In architecture, for example, this is often known when the SOR lies centered on top of the underlying parent block. Otherwise, we could get such a point from two images (see section 1.4 about reconstructing point positions). We also require knowledge of the axis direction, which for most practical cases is $\mathbf{y}^{\mathbf{W}} = [\ 0 \quad 1 \quad 0\ ]^T$, the vertical axis of the world. We also know image points $\mathbf{p}_i^{\mathbf{I}} = (x_i^{\mathbf{I}}, y_i^{\mathbf{I}})$ marked by the user that lie on the occluding contour in the image plane.

First, we get the camera coordinates $\mathbf{p}_b$ of the base point using equation 1.1. Then the axis direction of the SOR is expressed in the camera coordinate system $\mathbf{m} = \mathbf{R}^{\mathbf{C}}\mathbf{y}^{\mathbf{W}}$. The contour points also get converted according to equation 1.3.

We want to find the minimum distances between the ray $\mathbf{p}_b + \lambda\mathbf{m}$ and and the set of rays $\mu_i\mathbf{p}_i$ (see Fig. 1.4). Exploiting the fact that the minimum distance vectors

$$\mathbf{d}_{i0} = (\mu_{i0}\mathbf{p}_i - \mathbf{p}_b - \lambda_{i0}\mathbf{m}) \tag{1.7}$$

must be perpendicular to the rays $\mathbf{p}_b + \lambda\mathbf{m}$ and $\mu_{i0}\mathbf{p}_i$, conveniently our task boils down to solving the following simultaneous equations with respect to λ_{i0} and μ_{i0}.

$$\begin{aligned} \mu_{i0}\mathbf{p}_i^T\mathbf{d}_{i0} &= 0 \\ \lambda_{i0}\mathbf{m}^T\mathbf{d}_{i0} &= 0 \end{aligned} \tag{1.8}$$

Excluding the trivial solutions $\lambda_{i0} = 0$ and $\mu_{i0} = 0$ and substituting equation 1.7 into

equation 1.8 we get

$$\begin{aligned} A\mu_{i0} - C\lambda_{i0} &= B \\ C\mu_{i0} - E\lambda_{i0} &= D \end{aligned} \tag{1.9}$$

where

$$\begin{aligned} A &= \mathbf{p}_i^T \mathbf{p}_i \\ B &= \mathbf{p}_i^T \mathbf{p}_b \\ C &= \mathbf{p}_i^T \mathbf{m} \\ D &= \mathbf{m}^T \mathbf{p}_b \\ E &= \mathbf{m}^T \mathbf{m} \end{aligned} \tag{1.10}$$

Further solving the system of simultaneous equations in 1.9 we obtain

$$\begin{aligned} \lambda_{i0} &= \frac{BC - AD}{AE - C^2} \\ \mu_{i0} &= \frac{B + C\lambda_{i0}}{A} \end{aligned} \tag{1.11}$$

Now knowing λ_{i0} and μ_{i0}, the radius R_i of a circular cross section offset by $H_i = \lambda_{i0}$ from $\mathbf{p}_b$ in the direction of $\mathbf{m}$ can be expressed by

$$R_i = \sqrt{\mathbf{d}_{i0}^T \mathbf{d}_{i0}} \tag{1.12}$$

The algorithm uses the quantities H_i and R_i to automatically generate the surface of revolution.

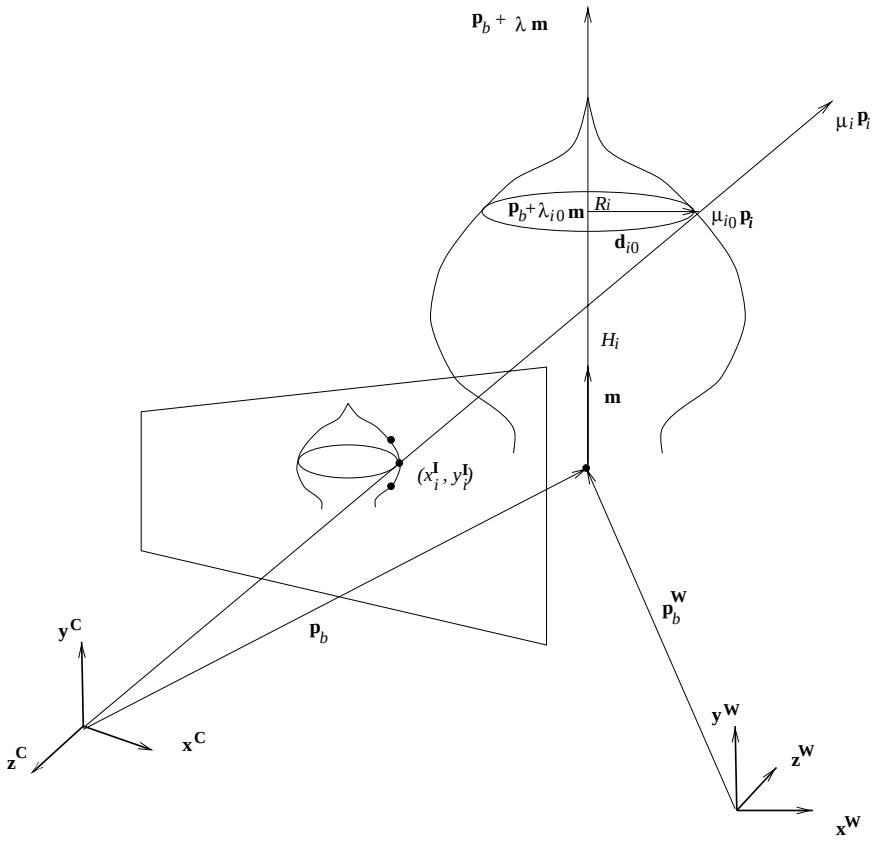


Figure 1.4: Geometry of the SOR reconstruction method.

1.3.2 Reconstruction Steps

The actual procedure for reconstructing a surface of revolution within Façade goes as follows:

1. First, the user selects *Surface of Revolution* from the *Add Block* submenu of the *World Viewer*. The program includes an *axis* block (a single vertical edge) which can be seen in *faces off* mode.

2. The *axis* position is then specified, either with relation to its parent block, or reconstructed as a position in space from two images with the *3-D point position* recovery tool.

3. The user selects an image and then marks the points $p_i^{\mathbf{I}}$ on the occluding contour of the SOR in this image with Façade's *point marquee* tool.

4. In order to prevent the algorithm from using points marked on another contour, the user surrounds the points to be used in the reconstruction with Façade's *quad* tool.

5. Then the *Surface of Revolution* option from the *Reconstruct* submenu is selected. This invokes the algorithm described in the previous section. For each marked point $p_i^{\mathbf{I}}$ within the *quad* it finds the values R_i and H_i.

6. These values are used for procedural generation of a SOR block following the block file syntax. The block file is saved on disk and then automatically included in the *World Viewer* replacing the SOR *axis* block.

1.3.3 Results

Fig. 1.5 demonstrates the successful use of the new SOR recovery algorithms for reconstructing a 3-D model of the Taj Mahal.

1.4 3-D Point Positions

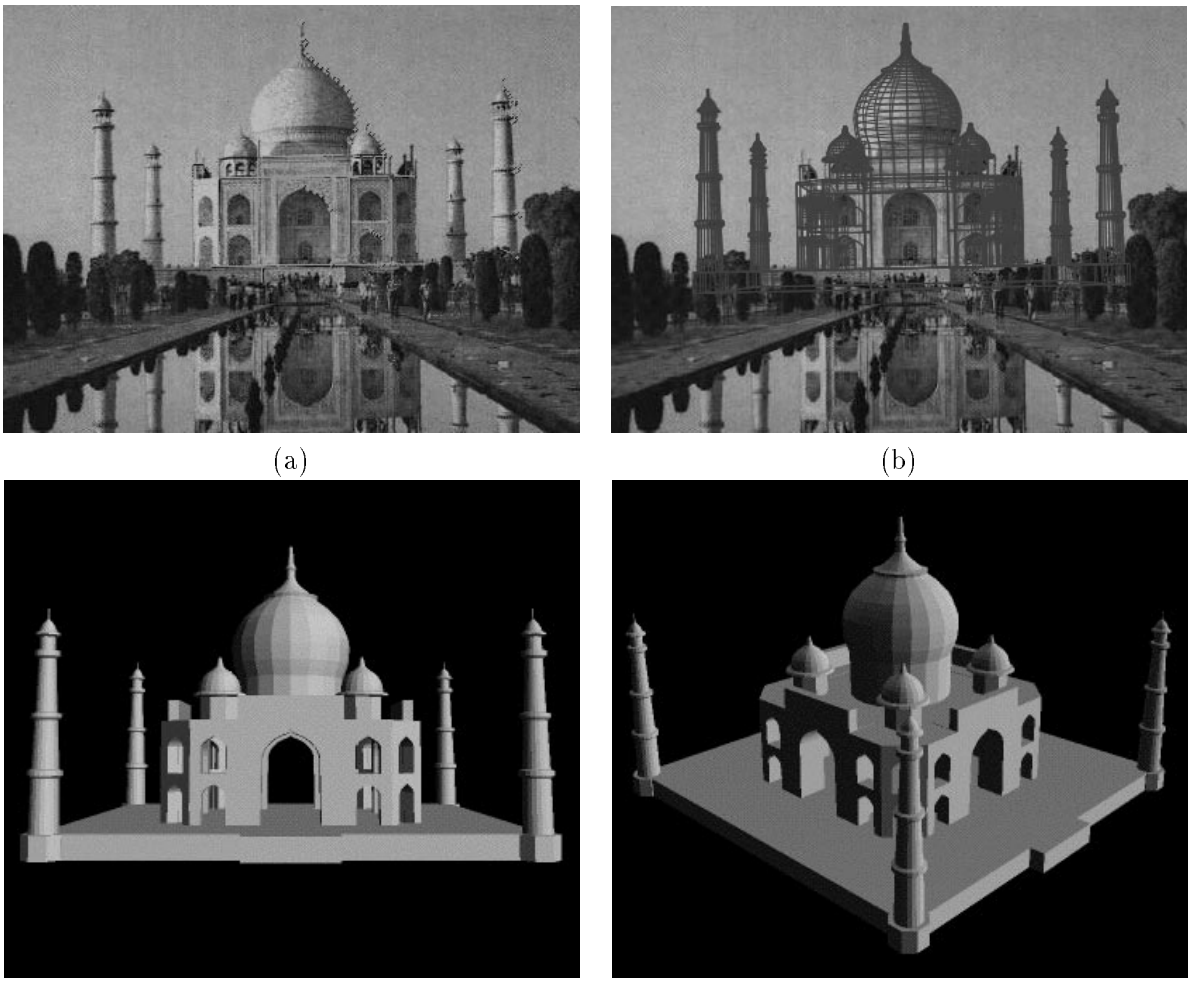
During the attempts for reconstructing a terrain mesh for the Berkeley campus environment (see section 1.5) a new tool for reconstructing single points in space through point correspondences was developed within Façade. The reconstruction steps are described below:

1. First the user selects *3-D Point Position* from the *Add Block* submenu of the *World Viewer*. The program includes a block (a single vertical edge) which can be seen in faces off mode. The bottom vertex of this edge defines the point position in space.

2. The user selects one or two images and marks with Façade's *point marquee* tool in each image the point which 3-D position needs to be recovered.

3. The user then corresponds the vertical edge of the 3-D point position block to the points marked in the images: $p^{\mathbf{I}}$ (in the case of one image) or $p_1^{\mathbf{I}}$ and $p_2^{\mathbf{I}}$ (in the two image case).

4. Then the *3-D Point Position* option from the *Reconstruct* submenu is selected. This invokes an algorithm which, in the case of two correspondences, assumes that the point position lies in the middle of the minimum distance sector between the rays from the image cameras' COPs through the marked points $p_1^{\mathbf{I}}$ and $p_2^{\mathbf{I}}$. In the case of



(a) (b)

(c) (d)

Figure 1.5: Model of the majestic Taj Mahal created with the new surface of revolution and arch reconstruction tools. **(a)** A single low-resolution photograph of the Taj Mahal obtained from the Internet, with marked features shown. **(b)** Reconstructed model edges projected onto the original photograph. **(c)** 3-D model of the Taj Mahal, complete with domes and minarets, recovered from the single photograph in less than an hour of modeling time. **(d)** Another view of the recovered 3-D model.

a single correspondence, the point position is reconstructed at a fixed distance along the ray from the image camera's COP through the point $p^{\mathbf{I}}$.

1.5 Berkeley Campus Environment Terrain

This section describes the procedures of reconstructing a 3-D model of the Berkeley campus that was later used in the photorealistic renderings of the Berkeley tower fly-by.

1.5.1 Campus Terrain

Using Façade we were able reconstruct about 47 campus buildings from aerial photographs and photographs taken from the Berkeley tower (see Fig.1.6 (a)). We had buildings floating in space, which was not enough. Somehow we needed to reconstruct the campus terrain. We noticed that we already had plenty of elevation data from the bases of the buildings. This data was extended by reconstructing extra points around the tower and the campus periphery with the new 3-D point position recovery tool. Then, the campus terrain 3-D mesh, in Fig. 1.6 (b) and (c), was generated in the format of a Façade block in the following way:

1. We extracted the bottom-most (minimum $\mathbf{y}$-coordinate) vertices of every Façade block whose parent was the ground plane. This gave us a cloud of 3-D points, which represented our elevation dataset.

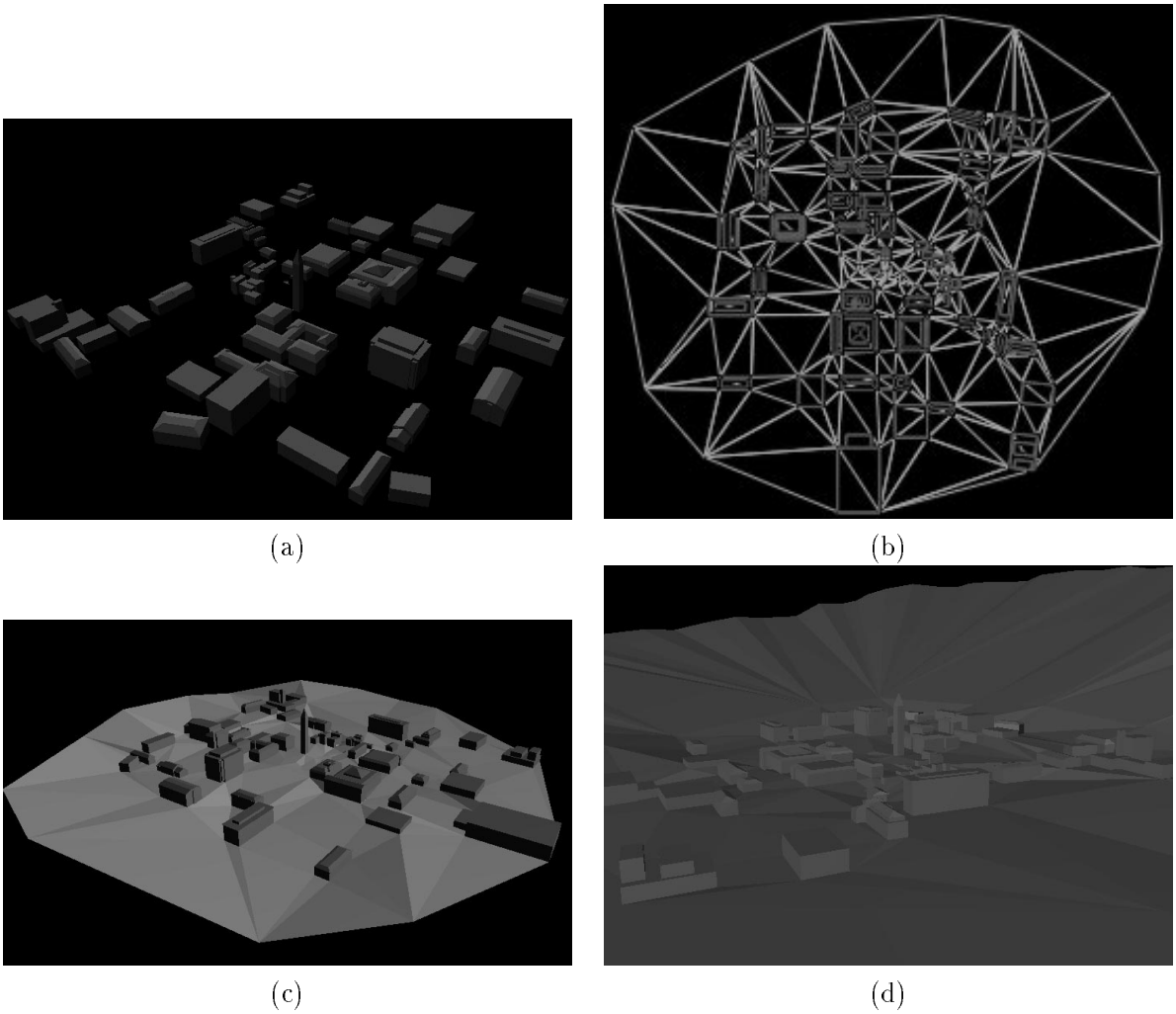
2. The elevation points were then projected onto the ground ($\mathbf{x} - \mathbf{z}$ plane).

3. Next we performed a Delaunay triangulation on this set of planar points.

4. Finally, we used the vertex connectivity obtained after the triangulation to connect the 3-D points in space and obtain a 3-D mesh approximating the campus terrain.

1.5.2 Extending the Mesh to the Environment Horizon

The next task was reconstructing a model of the environment beyond the campus periphery. For this we started by marking points on the image horizon. Then each point was corresponded to a 3-D point position block. The 3-D point positions were reconstructed at a great distance along the rays from the camera COPs through the image marks. These new points along the horizon had to somehow be connected to the campus terrain. Initially, we tried to use the technique described in the previous section, but that produced bad triangulations of the horizon points since their projections onto the ground plane did not form a convex polygon. In order to solve the problem of properly connecting the horizon points to the campus terrain points, we implemented a new triangulation algorithm. The algorithm visits each 3-D horizon point in order and finds its closest campus terrain point. Then it establishes edges and faces between those two points, the previous horizon point and its closest campus point. Similar techniques were used to model the sky.



(a) (b)

(c) (d)

Figure 1.6: Campus environment model demonstrating the new 3-D point position and 3-D mesh reconstruction algorithms. **(a)** Campus buildings reconstructed with Façade. **(b)** Delaunay triangulation of the elevation dataset. **(c)** The recovered campus 3-D terrain mesh. **(d)** Environment horizon.

Chapter 2

Rendering

This chapter presents the new rendering techniques we developed for performing both view-dependent and non view-dependent hardware projective texture mapping. The view-dependent projective texture mapping scheme described in [3] was computationally expensive. It required about 10 minutes to render a single synthetic view of a simple model. Certain performance enhancements were also suggested in [3]. These enhancements included avoiding image camera selection calculations at every pixel and pre-selecting which real images to use for rendering a given virtual view. This chapter describes the advances in exploring, extending, and implementing these suggestions.

2.1 Exploration of Projective Texture Mapping in OpenGL

This section presents the OpenGL capabilities for performing projective texture mapping. Examples are provided and the need for a novel visibility algorithm is justified.

2.1.1 Practice

Given the fundamentals and developments of texture mapping techniques covered in [6, 7, 12] the task was to find the most efficient way to do the projective texture mapping with our new ONYX *RealityEngine2* capable of performing texture mapping in hardware. For more detail about the *RealityEngine* architecture see [1]. The implementation in OpenGL involved careful examination of its automatic texture coordinate generation capabilities (see [10, 9]).

After appropriate parameter specification for the texture coordinate generation function, each texture coordinate s, t, r, and q for a vertex could be a linear combination of the object coordinates of the vertex $[\ x^{\mathbf{O}} \quad y^{\mathbf{O}} \quad z^{\mathbf{O}} \quad w^{\mathbf{O}}\ ]^T$. This could be written as

$$\begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} = \begin{bmatrix} p_{s1} & p_{s2} & p_{s3} & p_{s4} \\ p_{t1} & p_{t2} & p_{t3} & p_{t4} \\ p_{r1} & p_{r2} & p_{r3} & p_{r4} \\ p_{q1} & p_{q2} & p_{q3} & p_{q4} \end{bmatrix} \begin{bmatrix} x^{\mathbf{O}} \\ y^{\mathbf{O}} \\ z^{\mathbf{O}} \\ w^{\mathbf{O}} \end{bmatrix} \tag{2.1}$$

For projective texture mapping we need the matrix to be identity which could be ensured by supplying the values $1,0,0,0$ for $p_{s1}, p_{s2}, p_{s3}, p_{s4}$, $0,1,0,0$ for $p_{t1}, p_{t2}, p_{t3}, p_{t4}$, and so on using the appropriate OpenGL calls. For the s texture coordinate, this can be accomplished by:

```
GLfloat objectPlaneS[] = { 1.0, 0.0, 0.0, 0.0 };
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glTexGenfv(GL_S, GL_OBJECT_PLANE, objectPlaneS);
```

In OpenGL, just as model coordinates are transformed by a matrix before being rendered, texture coordinates are multiplied by a 4×4 matrix $\mathbf{M}_{texture}$ before any texture

mapping occurs.

$$\begin{bmatrix} s' \\ t' \\ r' \\ q' \end{bmatrix} = \mathbf{M}_{texture} \begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} \tag{2.2}$$

This matrix is identity by default. However, by modifying it while redrawing an object, one can make the texture slide over the surface, rotate around it, stretch and shrink, or any combination of the three. In fact, effects such as perspective, in which we are interested, can be achieved since $\mathbf{M}_{texture}$ is a completely general 4×4 matrix.

The transformed texture coordinates $[\ s' \quad t' \quad q' \quad r'\ ]^T$ are interpreted as homogeneous. In other words, the texture map is indexed by $\frac{s'}{q'}$ and $\frac{t'}{q'}$. For our purposes of projecting photographs onto geometry reconstructed from them the texture matrix has the following structure. It contains a part that performs the projection of the model vertices into the appropriate image texture plane. A second part makes a scaling and translational adjustment of the coordinates in order to account for the fact that texture coordinates always run from 0 to 1, starting at the texture's bottom left corner and texture size is always restricted to $2^m \times 2^n$ where m and n are integers.

Since we have already set $[\ s \quad t \quad r \quad q\ ]^T = [\ x^{\mathbf{O}} \quad y^{\mathbf{O}} \quad z^{\mathbf{O}} \quad w^{\mathbf{O}}\ ]^T$ and, in our case, the entire geometry is in world coordinates, i.e. the vertex object coordinates are effectively world coordinates, the first part of the texture coordinate transformation involves the familiar quantities $\mathbf{R}^{\mathbf{C}}$ and $\mathbf{T}^{\mathbf{C}}$. These quantities specify the image camera position with respect to the world and determine the transformations from world to camera coordinates and vice versa. Those matrices are determined with Façade during the 3-D model

reconstruction. Therefore, the first part of the texture matrix is **PRT** where

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -T_x^{\mathbf{C}} \\ 0 & 1 & 0 & -T_y^{\mathbf{C}} \\ 0 & 0 & 1 & -T_z^{\mathbf{C}} \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{R} = \begin{bmatrix} \mathbf{R}^{\mathbf{C}} & 0 \\ 0 & 1 \end{bmatrix} \qquad \mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & -2n \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{2.3}$$

The matrix $\mathbf{P}$ is the perspective projection matrix used in Façade and n is the near clipping plane.

The second part assures that the automatically-generated texture coordinates are going to correctly index the image texture map, which is a resized photograph. It uses information for the camera focal length f, image plane center (u_0, v_0), and image size $w \times h$. This second part of the texture matrix is

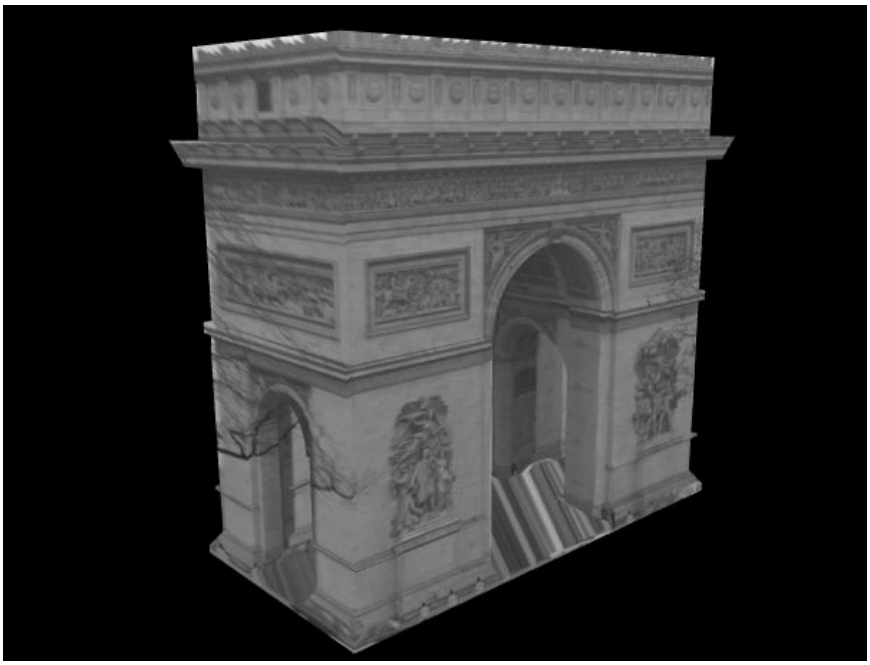
$$\mathbf{T_{cop}S} = \begin{bmatrix} 1 & 0 & 0 & \frac{u_0}{w} \\ 0 & 1 & 0 & \frac{v_0}{h} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{f}{w} & 0 & 0 & 0 \\ 0 & \frac{f}{h} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

So, finally for the texture matrix we have

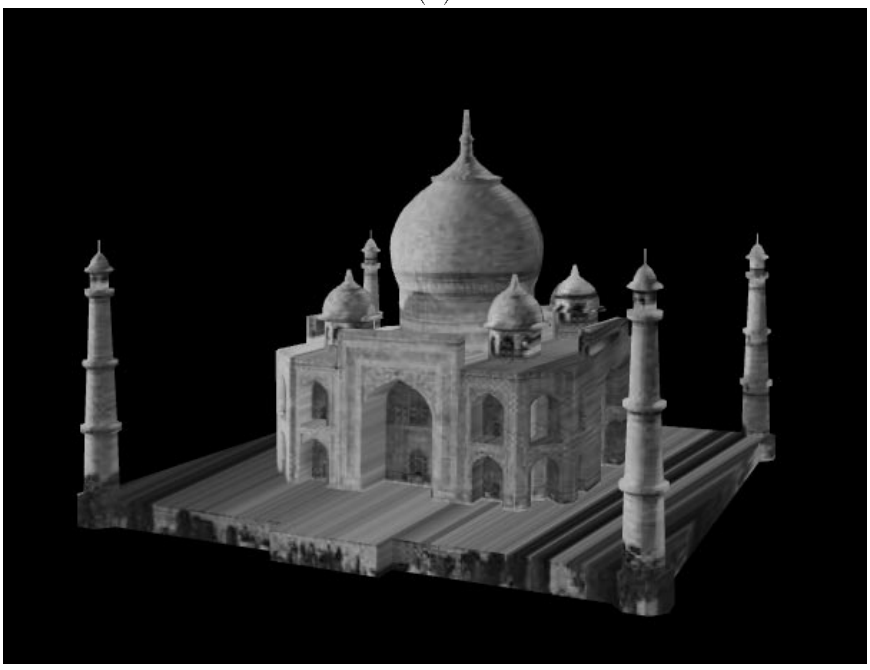
$$\mathbf{M}_{texture} = \mathbf{T_{cop}SPRT} \tag{2.5}$$

2.1.2 Results

The procedure for performing projective texture mapping, which was presented in the previous section, was used to generate the results in Fig. 2.1 (a), 2.1 (b), and 2.2 (b). The synthetic images are snapshots of an application that lets the user manipulate the 3-D model interactively in real-time.



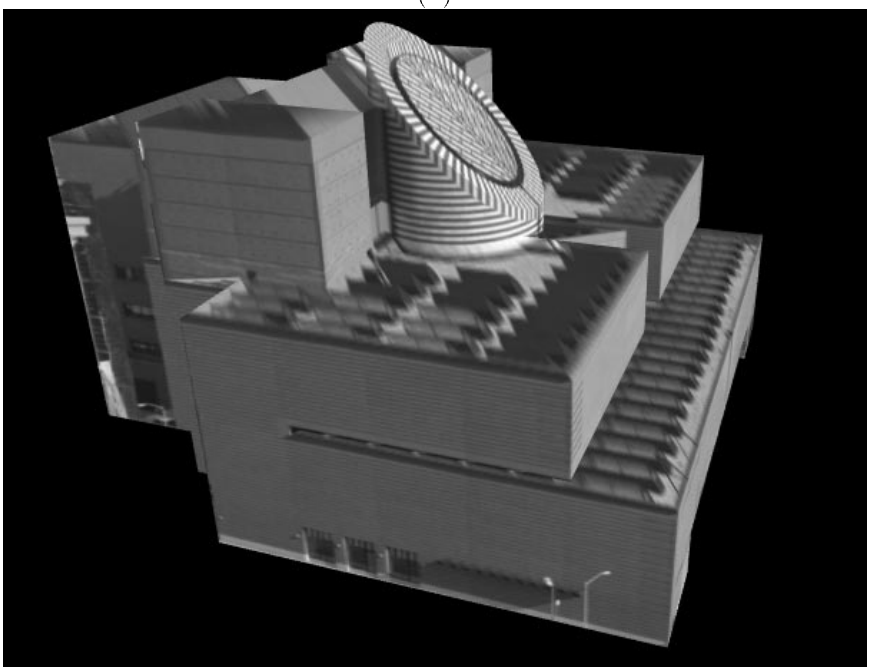
(a)



(b)

Figure 2.1: The images above are snapshots of a real-time projective texture mapping application written in OpenGL that projects a single photograph at a time onto a Façade 3-D model. **(a)** The photograph from 1.3 (a) projected onto the Arc de Triomphe model. **(b)** The photograph from 1.5 (a) projected onto the Taj Mahal model.

(a)



(b)

Figure 2.2: Another example of real-time projective texture mapping. **(a)** One of two photographs used in the reconstruction of a 3-D model of the San Francisco Museum of Modern Art by Yizhou Yu. **(b)** The photograph from (a) projected onto the SFMOMA model.

The application takes as input the original geometry generated with Façade and exported in a *save.blocks* file, as well as, the images used in the reconstruction. The user can switch between the images that are projected onto the model. The application produces pleasing effects, as long as the synthetic view is not too far from the position from which the photograph was taken.

2.1.3 Need for a Visibility Algorithm

When parts of the geometry not visible in the original image are exposed, we are able to notice an interesting but unpleasant artifact of the projective texture mapping scheme. The algorithm lacks the notion of visibility, so parts of the geometry that are occluded in the original image still receive legible texture coordinates and are incorrectly texture mapped instead of remaining in shadow. The effect is easily observed in Fig. 2.3. This result clearly shows the need for a pre-processing step which would determine the visibility of the different parts of the model in the images used in the texture projection. An algorithm developed and implemented by Yizhou Yu, with some help from me and Paul Debevec, uses a hybrid (object-image space) approach. First the original model geometry is clipped to each camera's viewing frustum, then appropriately triangulated and subdivided. The resulting triangles are then rendered with a unique color ID from the viewpoint of each image camera. The resulting images are used to determine the visibility of triangles. Triangles partially visible in a certain image camera are clipped (in object space) to neighboring (in image space) triangles. For completely invisible triangles, vertex colors are derived from neighboring visible triangles. At the end, the program outputs a file (see section 2.4.1) with the following information:

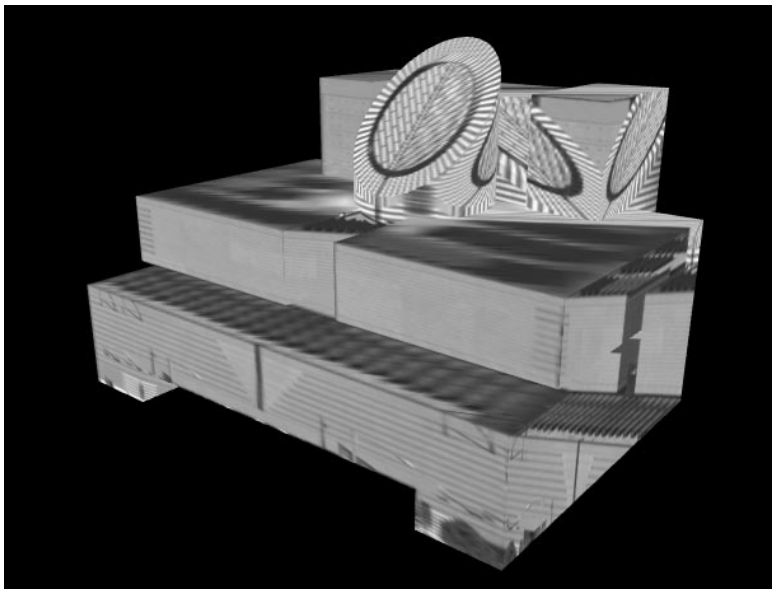


Figure 2.3: Viewing the model from a viewpoint far from the original produces artifacts unless proper visibility pre-processing is performed.

1. For each visible triangle: a list of image cameras in which the triangle is visible.

2. For each invisible triangle: vertex colors derived from neighboring visible triangles.

2.2 Selecting the Image Cameras for Texture Mapping

After the visibility problem was resolved, we faced another interesting problem. For each triangle in the model geometry, we could now have more than one image camera in which the triangle was visible. This posed a couple of questions. How many of the visible image cameras per triangle should we use in the texture mapping? How should we select them? After the selection, in what proportions should we blend them?

The following sections describe first a scheme in which a single image camera per triangle is used in the texture mapping. Then, the development of a new view-dependent texture mapping scheme is presented. In this scheme, three image cameras per triangle are

selected based on the current viewpoint, and their textures blended in appropriate ratios.

2.2.1 Non View-Dependent Texture Mapping

This section describes a simple method for selecting a single image camera per triangle which would always be used to texture map this triangle independent of the current viewpoint.

The selection criterion is based on computing the dot product between the triangle normal $\mathbf{n}$ and the viewing direction $\mathbf{v}_i = [\mathbf{R^C}]^{-1}[\ 0 \quad 0 \quad -1\ ]^T$ of an image camera i in which the triangle is visible (see Fig. 2.5). Therefore, for each individual triangle the best image camera is determined by

$$\text{best camera} = \arg\left[\min_i(\mathbf{n}^T\mathbf{v}_i)\right] \tag{2.6}$$

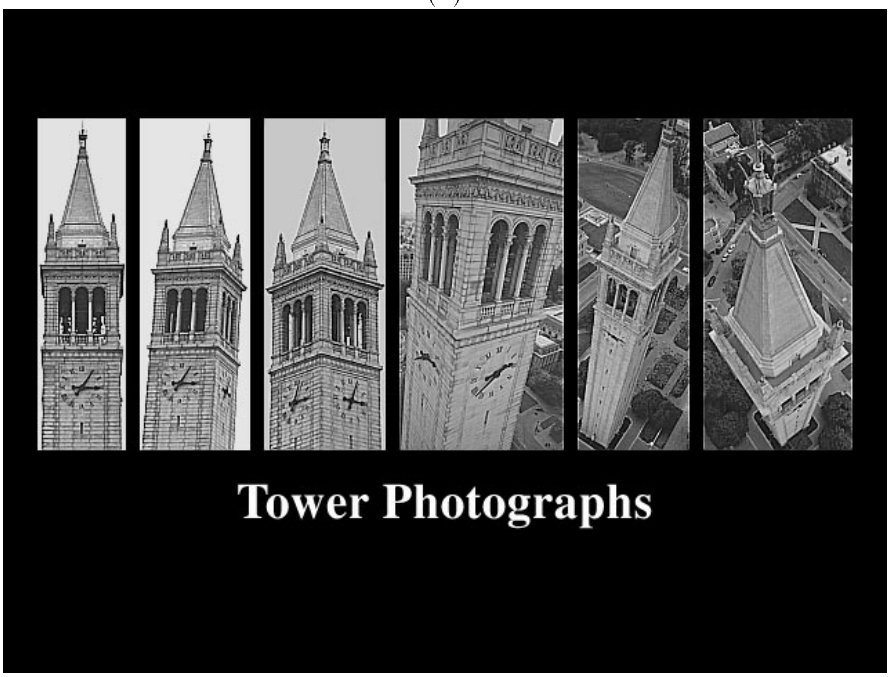
This scheme produces satisfactory results (see Fig. 2.6). Its biggest advantage is that it eliminates any additional processing during the rendering display loop and thus achieves faster renderings.

2.2.2 View-Dependent Texture Mapping

Early Versions: Minimum Distance and Triangulation

This section describes some early attempts to select the 3 image cameras and their weights to be used in the texture mapping of a single triangle from the model geometry. This selection has to be made for every triangle and for every new view that is being rendered. All approaches use a 2-D mapping of the image cameras and the current viewpoint on the surface of each triangle from the model geometry. The local coordinate system, in which

(a)



(b)

Figure 2.4: The images above were used to texture map the model from Fig.1.6 to create the photorealistic renderings of flying around the Berkeley tower. **(a)** Environment photographs taken from the top of the Berkeley tower (courtesy of Paul Debevec). **(b)** Tower photographs taken from ground level and a kite rig (courtesy of Paul Debevec and Prof. Charles Benton).

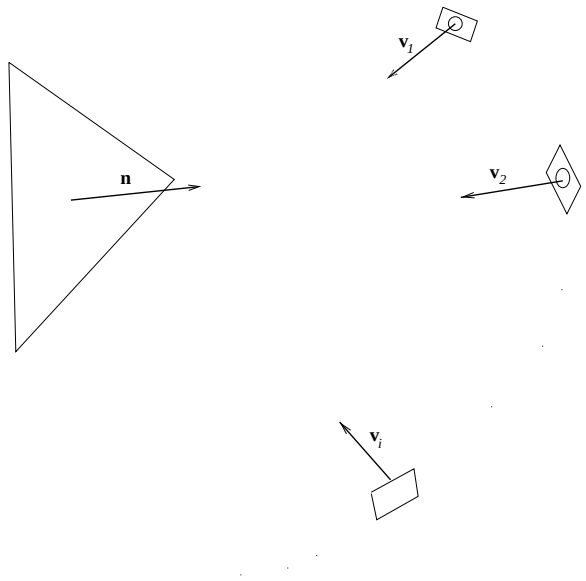


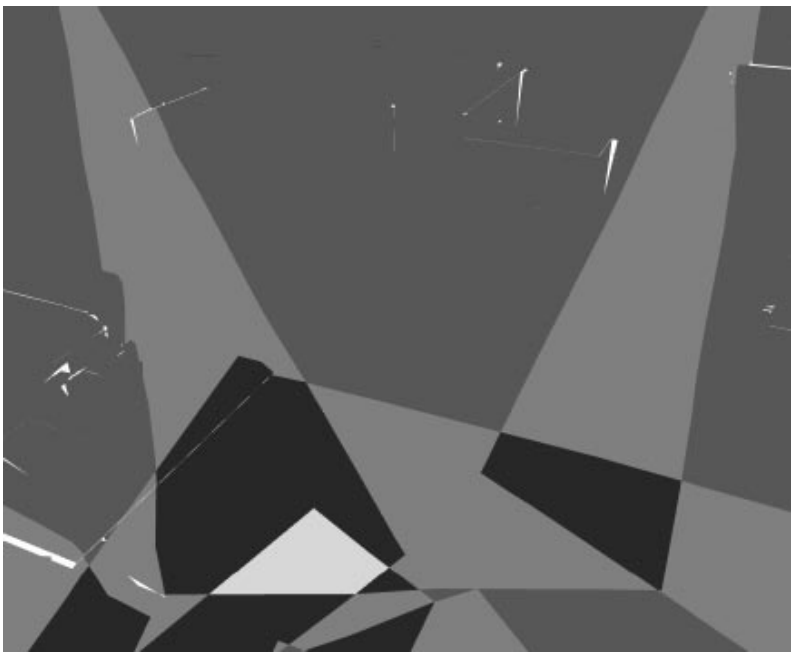
Figure 2.5: For each triangle the best camera is the selected based on the angle between the triangle normal $\mathbf{n}$ and the viewing directions $\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_i, ...$ of the cameras in which the triangle is visible.

the 2-D mapping coordinates are expressed, was originally constructed as shown in Fig 2.7 according to equation 2.7. Later, we are going to demonstrate that this construction was not the best one and had to be changed.

$$\begin{aligned} \mathbf{x} &= \frac{\mathbf{v}_0 - \mathbf{c}}{|\mathbf{v}_0 - \mathbf{c}|} \\ \mathbf{y} &= \mathbf{n} \times \mathbf{x} \end{aligned} \tag{2.7}$$

where $\mathbf{n}$ is the triangle unit normal, $\mathbf{c}$ is the triangle centroid and $\mathbf{v}_0$ one of the triangle vertices.

The camera position mapping we decided to use is approximately metric preserving. It is constructed as seen in Fig. 2.8 in the following way. We first obtain $\mathbf{v} = \frac{\mathbf{T}^C - \mathbf{c}}{|\mathbf{T}^C - \mathbf{c}|}$, the unit vector in the direction from the triangle centroid to the image camera's COP location. We then rotate this vector into the $\mathbf{x} - \mathbf{y}$ plane of the local coordinate system for



(a)

(b)

Figure 2.6: Texture mapping using a single texture per triangle: **(a)** Visibility results for the campus environment model with the images from Fig. 2.4 (a): triangles seen in 1 image camera are in red, in 2 image cameras are in green, in three image cameras are in blue, etc. Invisible in white. **(b)** Rendering results: one texture was selected for each triangle based on equation 2.6 and then the scene was rendered as described in section 2.3.1.

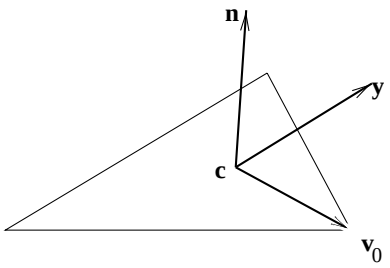


Figure 2.7: Original local coordinate system for 2-D image camera position mapping.

the triangle.

$$\mathbf{v}_r = (\mathbf{n} \times \mathbf{v}) \times \mathbf{n} \tag{2.8}$$

This vector is then scaled by the arc length $l = \cos^{-1}(\mathbf{n}^T\mathbf{v})$ and projected onto the $\mathbf{x}$ and $\mathbf{y}$ axes giving the desired 2-D mapping of the image camera position:

$$\begin{aligned} x &= (l\mathbf{v}_r)^T\mathbf{x} \\ y &= (l\mathbf{v}_r)^T\mathbf{y} \end{aligned} \tag{2.9}$$

We pre-compute and store for each triangle of the model the 2-D mapping coordinates $\mathbf{p}_i = (x_i, y_i)$ for each image camera i in which the triangle is visible. The selection of three image cameras per triangle for a virtual viewpoint always involves obtaining the the 2-D mapping coordinates $\mathbf{p}_{virtual} = (x_{virtual}, y_{virtual})$ of that viewpoint.

The first scheme used to determine the three best image cameras, based on the current viewpoint, was simplistic. We picked the three nearest neighbors of the virtual viewpoint based on the distance metrics (see Fig. 2.9)

$$d_i^2 = (x_i - x_{virtual})^2 + (y_i - y_{virtual})^2. \tag{2.10}$$

Then the weights to be used in the blending (in this example of textures from image cameras

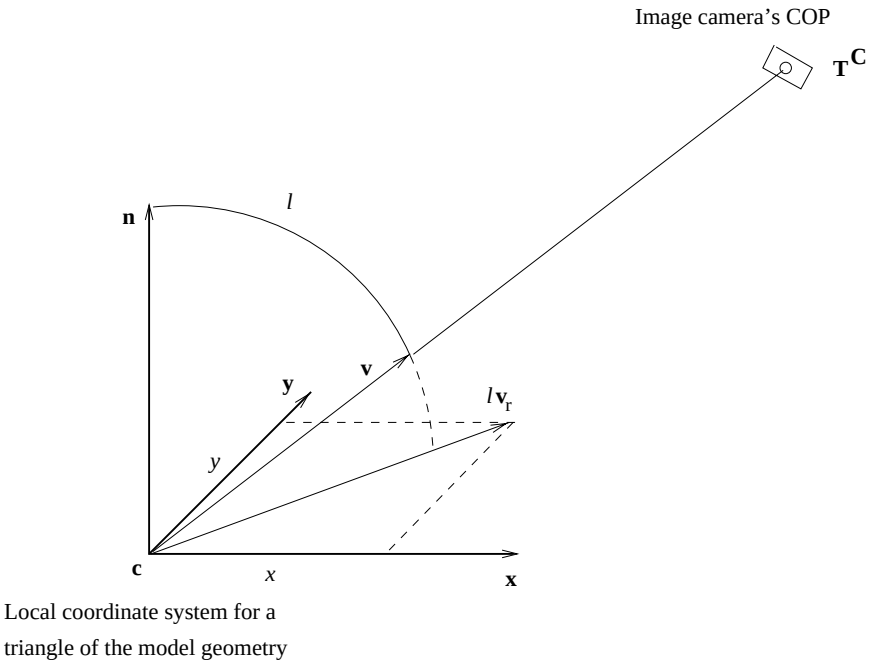


Figure 2.8: Image camera position 2-D mapping.

4, 5, and 6) were calculated as follows:

$$\begin{aligned} \alpha_4 &= \frac{d_4^{-2}}{d_4^{-2} + d_5^{-2} + d_6^{-2}} \\ \alpha_5 &= \frac{d_5^{-2}}{d_4^{-2} + d_5^{-2} + d_6^{-2}} \\ \alpha_6 &= 1 - \alpha_4 - \alpha_6 \end{aligned} \tag{2.11}$$

This scheme obviously did not guarantee gradual transitions from one image camera to another.

The next method theoretically was supposed to fix that problem. The idea was to

1. Triangulate the planar set of image camera mappings $\mathbf{p}_i = (x_i, y_i)$ using a Delaunay triangulation algorithm.

2. Find the triangle or region in which the current virtual viewpoint maps (see Fig.

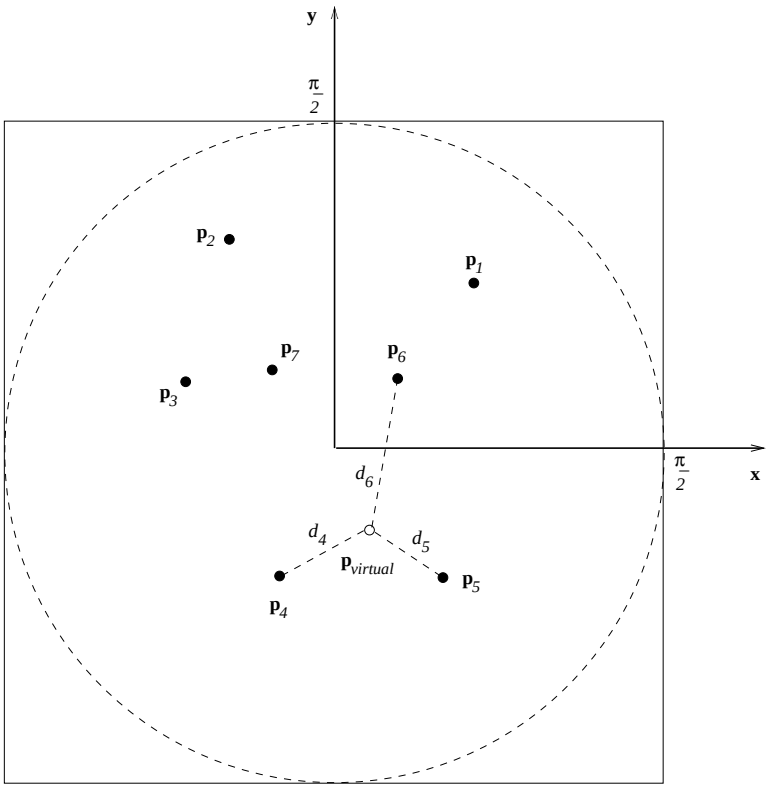


Figure 2.9: Nearest-neighbors method for camera selection.

2.10).

3. If the viewpoint mapped into a region outside the convex-hull of the set of image camera mappings $\mathbf{p}_i$, e. g. $\mathbf{p}'_{virtual}$, use the two image cameras (in this example 3 and 4) associated with this region in the blending. The weights are then calculated according to equation 2.12. If one the weights becomes negative, it is reset to 0 and the other weight is 1.

$$\begin{aligned} \alpha_4 &= \frac{(\mathbf{p}_4 - \mathbf{p}_3)^T(\mathbf{p}'_{virtual} - \mathbf{p}_3)}{|\mathbf{p}_4 - \mathbf{p}_3|^2} \\ \alpha_3 &= 1 - \alpha_4 \end{aligned} \tag{2.12}$$

4. If the viewpoint mapped inside a triangle, e.g. $\mathbf{p}''_{virtual}$, use the three image cameras represented by its vertices in the blending (4, 5, and 6 in this example). The weights are the barycentric coordinates of $\mathbf{p}''_{virtual}$ in the triangle in which it lies:

$$\begin{aligned} \alpha_4 &= \frac{\Delta_{\mathbf{p}_5\mathbf{p}_6\mathbf{p}''_{virtual}}}{\Delta_{\mathbf{p}_5\mathbf{p}_6\mathbf{p}_4}} \\ \alpha_5 &= \frac{\Delta_{\mathbf{p}_6\mathbf{p}_4\mathbf{p}''_{virtual}}}{\Delta_{\mathbf{p}_5\mathbf{p}_6\mathbf{p}_4}} \\ \alpha_6 &= 1 - \alpha_4 - \alpha_5 \end{aligned} \tag{2.13}$$

where Δ is the area of the triangle formed by the points in the subscript.

This technique, unfortunately, did not produce the expected results. Generic sets of image camera positions (see Fig. 2.11 (a)) produced degenerate planar mappings and triangulation configurations, like the one shown in Fig. 2.11 (b). The presence of long and skinny triangles meant that the virtual viewpoint would rarely map inside them. If it did that would happen only for a frame or two, and then map outside again. That ultimately

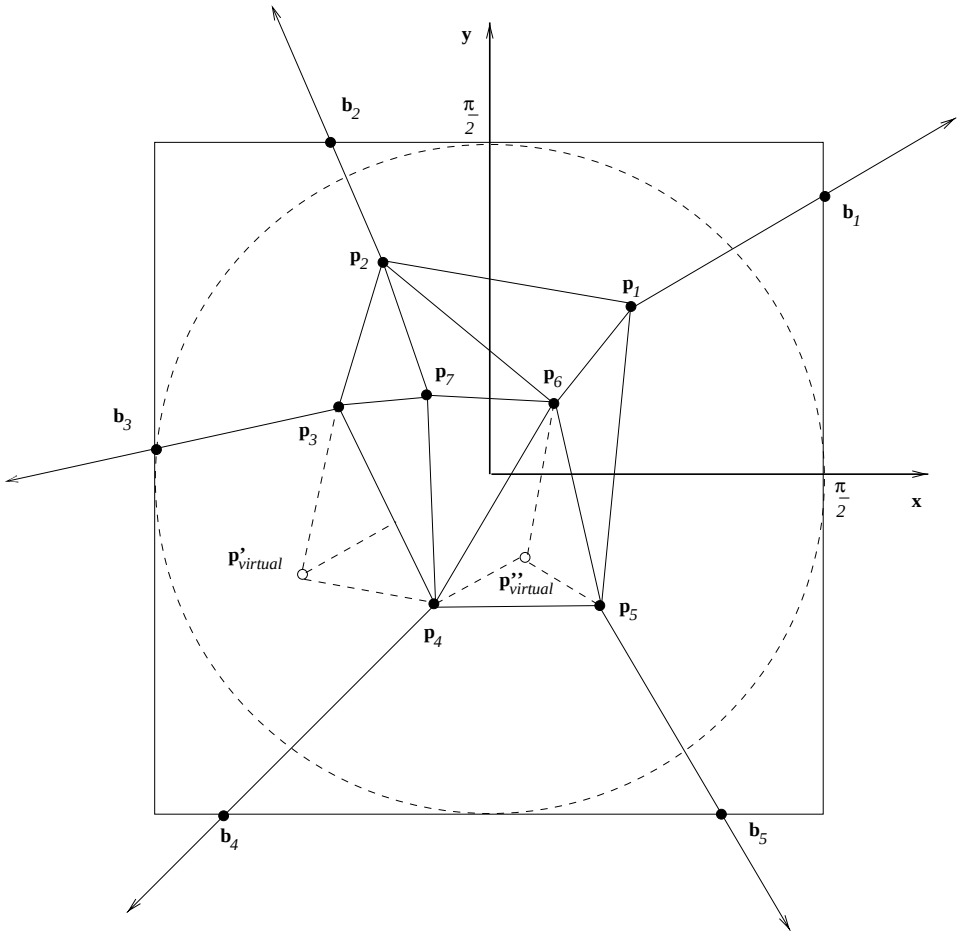
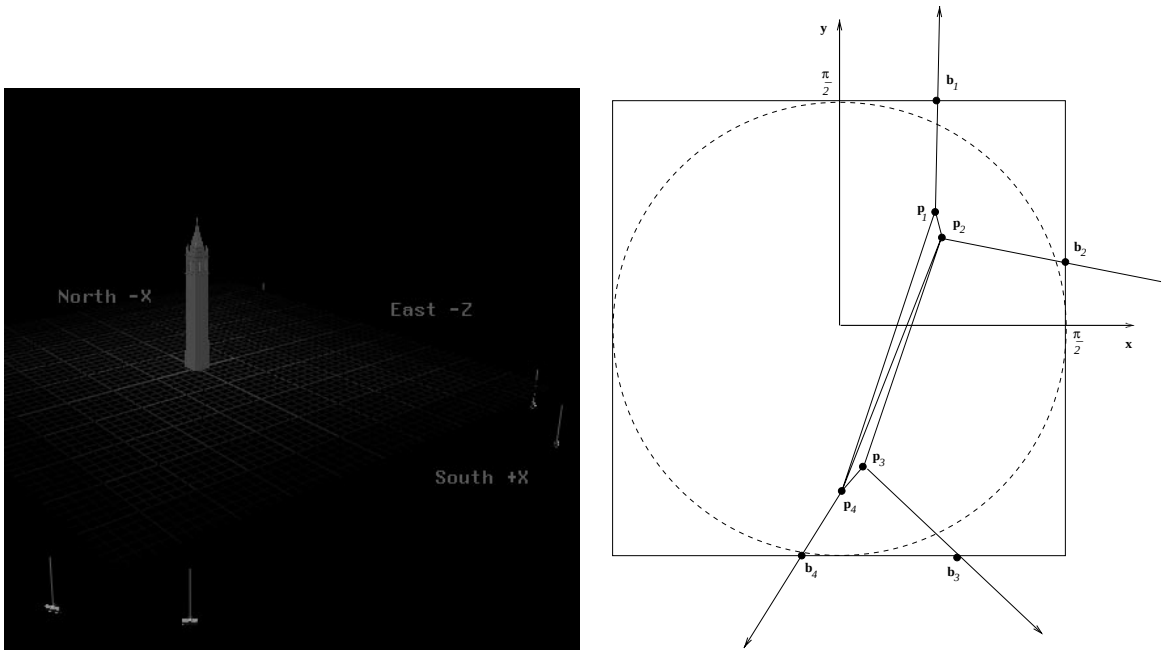


Figure 2.10: Delaunay triangulation method for camera selection. The points $\mathbf{p}_i$ are the planar mappings of the image cameras in which the triangle is visible. The points $\mathbf{b}_i$ are the intersections of the bisectors at each point of the convex-hull with the square region.

resulted in more noticeable changes in the texturing of the model geometry with the change of viewpoint. In order for this scheme to work, we had to guarantee picture sets that were taken at very different elevations (certainly, a quite impractical requirement). Clearly, we needed to come up with a more robust solution.



(a) (b)

Figure 2.11: An example where the Delaunay method fails. **(a)** A common image camera configuration. **(b)** The corresponding 2-D mapping triangulation for a triangle from the model geometry contains long and skinny triangles. This results in undesired sudden changes of the set of three image cameras used for texturing when moving the virtual viewpoint.

A Solution: Adjustment to a Regular Sampling Grid

The solution to the problem was to adjust the image camera 2-D mappings to a regular sampling grid instead of using a configuration like the one from Fig. 2.11 (b). In order to achieve this we developed the following algorithm (see Fig. 2.12).

1. Construct a regular grid for the image camera position 2-D mapping space of each triangle from the model geometry.

2. Obtain values for the vertices of this grid by assigning the closest mapped image camera.

3. During the rendering, find the triangle in which the current viewpoint maps.

4. Use the three image cameras represented by the vertices of this triangle in the blending (4, 5, and 7 in the example from 2.12). The weights are the barycentric coordinates of $\mathbf{p}_{virtual}$ in the triangle in which it lies.

After examining the results of the algorithm described above, we noticed a flaw in the way we were constructing the local coordinate system **xyn** (see eqn. 2.7) for a triangle of the model geometry. Triangles with the same normal $\mathbf{n}$ could have different $\mathbf{x}$ and $\mathbf{y}$ axes. This caused some discontinuities between the texturing of adjacent triangles with the same normal. We had to redefine the coordinate system axes (see eqn. 2.14) and assure that $\mathbf{x}$ and $\mathbf{y}$ are the same for such triangles.

$$\begin{aligned} \mathbf{x} &= \begin{cases} \mathbf{y}^{\mathbf{W}} \times \mathbf{n} & \text{, if } \mathbf{y}^{\mathbf{W}} \text{ and } \mathbf{n} \text{ are not collinear,} \\ \mathbf{x}^{\mathbf{W}} & \text{otherwise} \end{cases} \\ \mathbf{y} &= \mathbf{n} \times \mathbf{x} \end{aligned} \tag{2.14}$$

where $\mathbf{x}^{\mathbf{W}}$ and $\mathbf{y}^{\mathbf{W}}$ are world coordinate system axes.

Results of applying this technique can be seen in Fig. 2.13 and details on the implementation can be found in section 2.3.2.

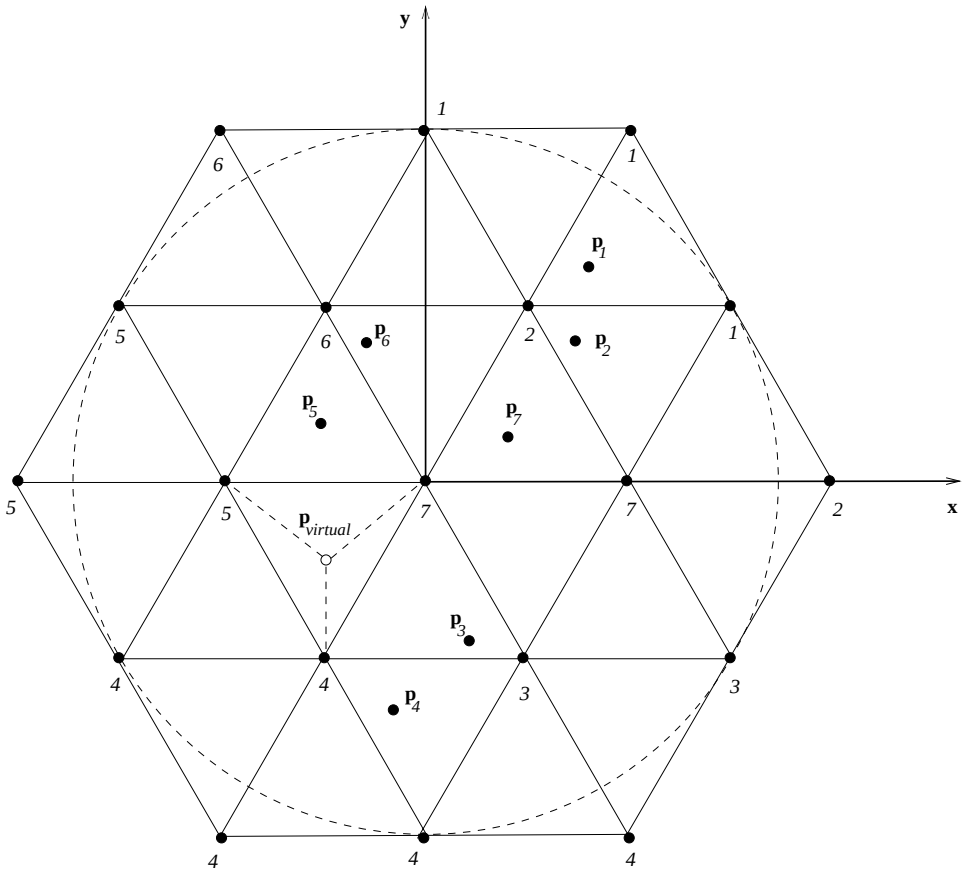
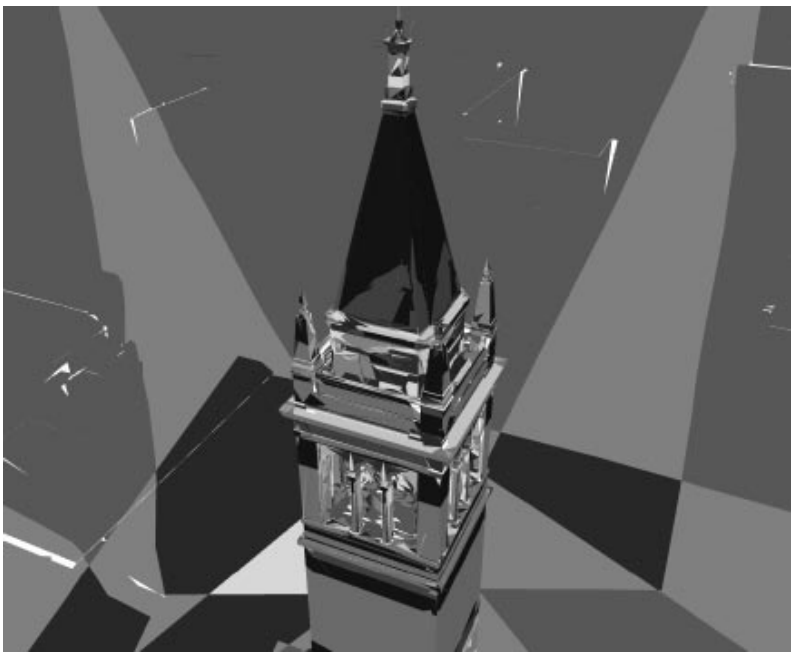


Figure 2.12: The final scheme which adjusts the 2-D mappings to a regular grid guarantees smooth changes of the set of three image cameras used for texture mapping when moving the virtual viewpoint.



(a)

(b)

Figure 2.13: View-dependent texture mapping using three textures per triangle: **(a)** Visibility results for the tower model (courtesy of Jason Luros and Vivian Jiang) with the images from Fig. 2.4 (b). **(b)** Rendering results: three textures and their weights were selected for each triangle based on the method described in section 2.2.2 and then the scene was rendered as described in section 2.3.2.

2.3 Implementation of the Multi-Pass Renderer

This section explains some details of the actual implementation of the new rendering algorithm and describes certain specific features.

2.3.1 Non View-Dependent Texture Mapping

First, for each triangle of the model geometry, the best image camera is selected using the criterion from section 2.2.1. Then the triangles are split into N separate lists (N being the total number of textures/image cameras), where each new list contains only triangles with the same best image camera. Those lists are then compiled into OpenGL display lists. Thus, each display list contains only triangles which are going to be textured by projecting the same image. The rendering proceeds as shown in Fig. 2.15 where the triangles are sent for display with color (1.0, 1.0, 1.0, 1.0). This is how the campus environment is rendered in the Berkeley campus fly-by (see Fig. 2.14 (a)).

2.3.2 View-Dependent Texture Mapping

For each triangle we pre-compute and store the centroid $\mathbf{c}$, the normal $\mathbf{n}$, the axes directions $\mathbf{x}$ and $\mathbf{y}$ computed according to equations 2.14, the image camera IDs i and planar mappings $\mathbf{p}_i$ computed as described in section 2.2.2, the regular sampling grid and the image camera IDs assigned to its vertices. Then before a frame is rendered for each triangle we find the planar mapping of the current viewpoint $\mathbf{p}_{virtual}$ and do a quick lookup to determine inside which triangle of the grid it lies. As explained in 2.2.2 this gives the three best image cameras/textures and their weights $\alpha_1, \alpha_2, \alpha_3$.

After these are known, the rendering is performed in three passes. Texture mapping is enabled in modulate mode, where the new fragment color C is obtained by multiplying the existing fragment color C_f and the texture color C_t. The Z-buffer test is set to *less than or equal* (`GL_LEQUAL`) instead of the default *less than* (`GL_LESS`). The first pass proceeds by selecting an image camera, binding the corresponding texture, loading the corresponding texture matrix transformation $\mathbf{M}_{texture}$ in the texture matrix stack and sending for display the part of the model geometry for which the first best camera is the selected one with colors $(\alpha_1, \alpha_1, \alpha_1)$. These steps are repeated for all image cameras. The results of this pass can seen on the tower in Fig. 2.14 (b). Before proceeding with the second pass we enable blending in the frame buffer, i.e. instead of replacing the existing pixel values with incoming values, we add those values together. The second pass then selects cameras and renders triangles for which the second best camera is the selected one with colors $(\alpha_2, \alpha_2, \alpha_2)$. The results of the second pass can seen on the tower in Fig. 2.14 (c). The third pass proceeds similarly rendering triangles for which the third best camera is the currently selected one with colors $(\alpha_3, \alpha_3, \alpha_3)$. The results of this last pass can seen on the tower in Fig. 2.14 (d).

2.3.3 Invisible Geometry

The triangles that are not visible in any image cameras are compiled in a separate OpenGL display list and their vertex colors are specified according to the results of the hole-filling algorithm developed by Yizhou Yu. Those triangles are rendered in another pass with Gouraud shading after the texture mapping is disabled.

(a) (b)

(c) (d)

Figure 2.14: The different rendering passes in producing a frame from the photorealistic renderings of the Berkeley campus virtual fly-by. **(a)** The campus buildings and terrain after non view-dependent texture mapping. **(b)** The Berkeley tower after the first pass of the view-dependent texture mapping scheme. **(c)** The Berkeley tower after the second pass of the view-dependent texture mapping scheme. **(d)** The complete rendering of the scene.

2.3.4 Display Loop

The block diagram in Fig. 2.15 summarizes the display loop steps.

2.4 Features of the Multi-Pass Renderer

2.4.1 Options and File Formats

Options

The renderer described in the previous section takes the following command line arguments:

```
usage: vdtmstagehhl [-size w h] [-path pathfilename [-calib W H U0 V0 F]
[-skip n] [-output filename [-interlaced] [-ppm|-tiff]]] [-nvd]
[triangles1-filename] <triangles2-filename>
```

Most of the arguments are self-explanatory. The file `pathfilename` contains the camera position information of the virtual path. It is in the format of a Façade file without any geometry. The `-calib` argument specifies the calibration values of the virtual camera. The `-skip` argument specifies that only every `n`-th frame should be rendered. This is how the renderings in Fig. 2.16 were produced. The `-nvd` argument specifies the use of non view-dependent texture mapping only.

Input File Format

The files `triangles1-filename` and `triangles2-filename` contain the model geometry and the visibility pre-processing data for the environment and the Campanile, respectively. The file format is the following:

Comment

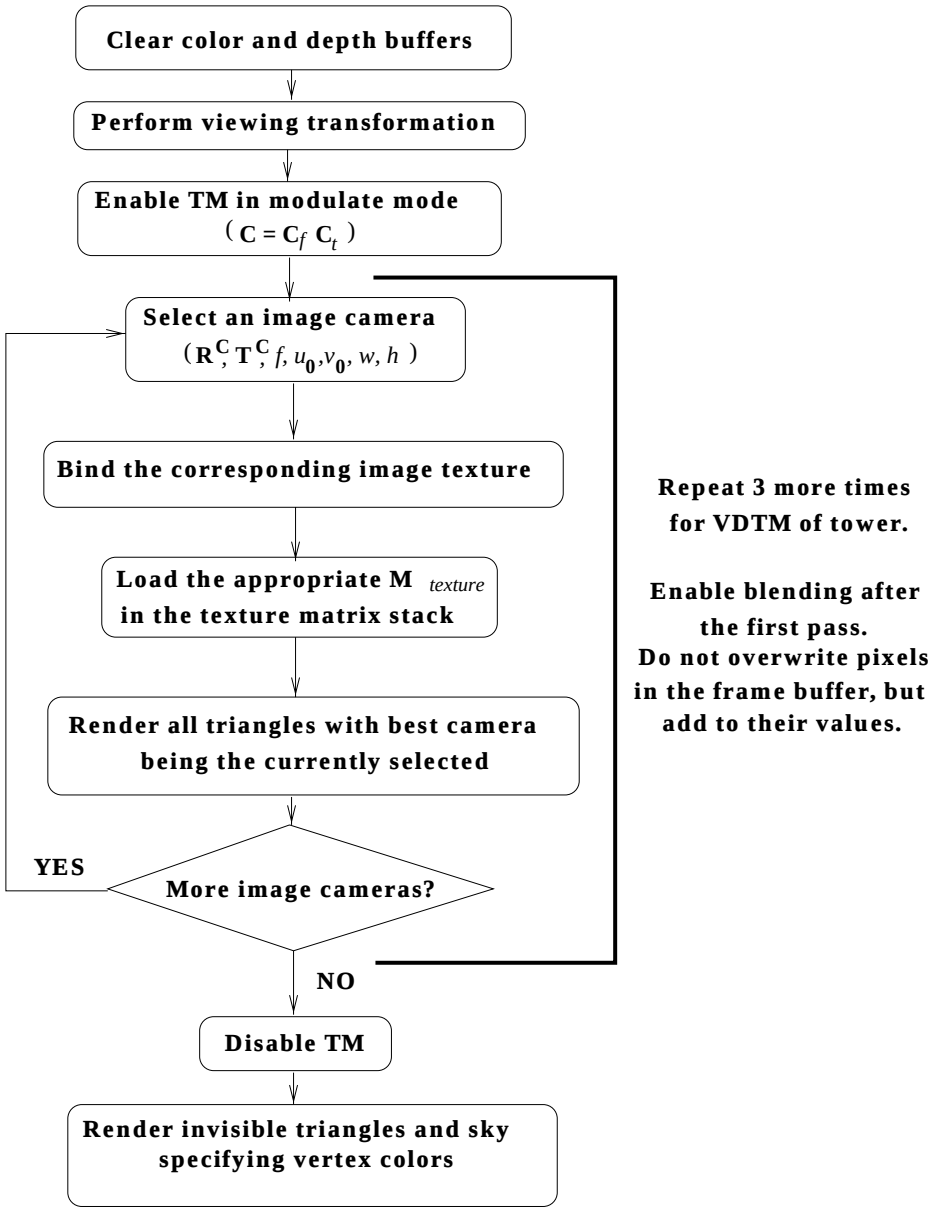


Figure 2.15: Multi-pass rendering display loop.

N - total number of image cameras/textures

texture-filename - image texture with width and height powers of 2

$w\ h$ - original width and height of the image

$T_x^{\mathbf{C}}\ T_y^{\mathbf{C}}\ T_y^{\mathbf{C}}$ - camera translation

$\alpha\ \beta\ \gamma$ - Euler angles of the camera rotation

$u_0\ v_0\ f\ a$ - image plane center, focal length, aspect ratio (not used)

.

.

.

Comment

$N_{triangles}\ N_{visible}$ - total number of triangles and number of visible triangles

$v_{1x}\ v_{1y}\ v_{1z}$ - vertex coordinates of the visible triangles

$v_{2x}\ v_{2y}\ v_{2z}$

$v_{3x}\ v_{3y}\ v_{3z}$

.

.

.

$v_{1x}\ v_{1y}\ v_{1z}$ - vertex coordinates of the invisible triangles

$v_{2x}\ v_{2y}\ v_{2z}$

$v_{3x}\ v_{3y}\ v_{3z}$

$R_1\ G_1\ B_1$ - vertex colors of the invisible triangles

$R_2\ G_2\ B_2$

$R_3\ G_3\ B_3$

.

.

.

Comment

n - number of image cameras in which the triangles is visible

i x_i y_i - image camera ID and planar mapping coordinates

.

.

.

2.4.2 Modes

After the renderer has been started the user can select the following modes. In interactive mode the user can navigate through the environment with the help of the mouse, **'w'** enables the wireframe, **'l'** enables the visibility labeling (see Fig. 2.6 and Fig. 2.13), **'t'** enables the texture mapping mode, **'v'** toggles between view-dependent and non view-dependent texture mapping on the Campanile (second geometry file), **'i'** toggles between a regular and a full-screen rendering without window borders. In flight-path mode, **'f'** renders the next frame in the flight-path, **'b'** renders the previous frame in the flight-path, and **'r'** renders the entire flight-path from the beginning.

Figure 2.16: Selected frames from the photorealistic renderings of the Berkeley campus fly-by produced with the described algorithms at 6 frames/second.

Bibliography

[1] Kurt Akeley. RealityEngine graphics. In *SIGGRAPH '93*, pages 109–116, 1993.

[2] Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. In *SIGGRAPH '96*, pages 11–20, August 1996.

[3] Paul E. Debevec. Modeling and rendering architecture from photographs. Doctor of Philosophy Thesis. Computer Science Division (EECS), University of California at Berkeley, Berkeley, CA, December 1996.

[4] Olivier Faugeras. *Three-Dimensional Computer Vision.* MIT Press, 1993.

[5] A. Gross and T. Boult. Recovery of Generalized Cylinders from a Single Intensity View. In *Proceedings of the Image Understanding Workshop*, pages 319-330, 1990.

[6] Paul S. Heckbert. Survey of texture mapping. *IEEE Computer Graphics and Applications*, 6(11):56–67, November 1986.

[7] Paul S. Heckbert. Fundamentals of Texture Mapping and Image Warping. Masters Thesis UCB/CSD 89/516, Computer Science Division (EECS), University of California at Berkeley, Berkeley, CA, June 1989.

[8] Eugene S. Lin. Recovery of 3-D Shape of Curved Objects from Multiple Views. Masters Thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1996.

[9] Tom McReynolds. Programming with OpenGL: Advanced Rendering. SIGGRAPH'96 Course Notes, New Orleans, LA, August 1996.

[10] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide*. Addison-Wesley Publishing Company, New York, NY, June 1993.

[11] M. Richetin, M. Dhome, J. T. Lapreste, and G. Rives. Inverse Perspective Transform Using Zero-Curvature Contour Points: Application to the Localization of Some Generalized Cylinders from a Single View. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(2):185-192, February 1991.

[12] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadows and lighting effects using texture mapping. In *SIGGRAPH '92*, pages 249–252, July 1992.

[13] Steve Sullivan, Lorraine Sandford, and Jean Ponce. Using Geometric Distance Fits for 3-D Object Modeling and Recognition. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(12):1183-1196, December 1994.

[14] 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.

[15] Mourad Zerroug and Ramakant Nevatia. Segmentation and Recovery of SHGCs from a Real Intensity Image. In *European Conference on Computer Vision*, pages 319-330, 1994.