

Interactive Editing Tools for Image-Based Rendering Systems

by

Sudeep Rangaswamy

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

February 4, 1998

Copyright 1998 Sudeep Rangaswamy. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and distribute publicly paper and electronic copies of this thesis and to grant others the right to do so.

Author _____
Department of Electrical Engineering and Computer Science
January 29, 1998

Certified by _____
Prof. Julie Dorsey
Thesis Supervisor

Accepted by _____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUL 14 1998

ENG.

Interactive Editing Tools for Image-Based Rendering Systems
by
Sudeep Rangaswamy

Submitted to the
Department of Electrical Engineering and Computer Science

February 4, 1998

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

We will describe a toolkit for editing 3D scenes in an image-based rendering system. The toolkit makes use of color, camera and depth information from a set of images to generate arbitrary view-points of a given scene. Our system allows the user to both navigate through a scene and change its structure, appearance, and illumination. These changes are achieved indirectly by applying editing operations to the component images. The goal of this work is to provide a scheme for editing virtual environments within the framework of an image-based rendering system.

Thesis Supervisor: Julie Dorsey

Title: Associate Professor, MIT Department of Architecture

Acknowledgments

This document is the product of over one year of research. It also marks the end of my MIT experience. I have many people to thank for getting me this far. Special thanks to Julie Dorsey, whose advice and encouragement guided me throughout this project. Thanks also to Steven Gortler, Leonard McMillan, and Steve Seitz for their help in developing and documenting the system presented here. Finally, I'd like to thank my family and friends, who put up with the trials and tribulations of my thesis. Here's what all that fuss was about.

Table of Contents

1.0	Image-Based Editing.....	11
1.1	Control over Imagery	11
1.2	Image-Based Rendering: Flexibility and Photorealism	13
1.3	Objectives of our System	15
1.4	Potential Applications.....	16
1.5	Outline.....	17
2.0	Related Work	18
2.1	Research in Image-Based Rendering	18
2.1.2	Movie-Maps.....	18
2.1.3	View Interpolation	19
2.1.4	QuickTime VR.....	19
2.1.5	View Morphing.....	20
2.1.6	Plenoptic Modeling.....	21
2.1.7	Light Field Rendering & The Lumigraph.....	21
2.1.8	Shape as a Perturbation of Projective Mapping.....	22
2.2	Research in Interactive Editing	23
2.2.1	Adobe Photoshop.....	23
2.2.2	WYSIWYG Painting	24
2.2.3	Painting with Light	24
2.3	Research in Image-Based Editing.....	25
2.3.1	3D Modeling and Effects on Still Images.....	25
2.3.2	Plenoptic Image Editing.....	26
2.4	Summary	27
3.0	An Overview of the Editing Toolkit.....	28
3.1	Interface	28
3.2	File Menu.....	29
3.3	Edit Menu.....	30
3.3.1	Copying, Cutting, & Pasting.....	30
3.3.2	Erasing Paint	31
3.4	Component Menu.....	31
3.5	Layer Menu.....	32
3.6	Render Menu.....	33
3.7	Navigation Tools	34
3.8	Selection Tools.....	34

3.9	Painting Tools	35
3.10	Magnify Tool	36
3.11	Re-illumination Tools	36
3.12	Scaling Tool	38
3.13	Skin Removal Tools	38
3.14	Summary	39
4.0	The Core of the System.....	40
4.1	Inputs to the System.....	40
4.2	Acquiring Depth & Camera Information.....	41
4.3	Overall Framework	42
4.4	Data Structures.....	43
4.4.1	The World Class	44
4.4.2	The toolchest Class	44
4.4.3	The vwarp Class.....	45
4.5	Image Warping & the Generalized Rendering Algorithm.....	45
4.6	Triangle Meshes.....	49
4.7	Skin Removal with the Threshold Lever	50
4.8	Summary	52
5.0	Adding & Removing Elements in an Image-Based Scene	53
5.1	Object Selection.....	53
5.1.1	Selection Events.....	53
5.1.2	Intelligent Scissors.....	54
5.2	Layers & Views.....	56
5.3	Cutting Objects	57
5.3.1	Selection.....	57
5.3.2	The clipboard	58
5.3.3	Finding Pixels that Warp to the Selected Region	58
5.3.4	Flipping the bits in the layerlist & clipboard	59
5.3.5	The Revised Rendering Algorithm	60
5.4	Copying Objects.....	62
5.5	The Cut More & Copy More Operations.....	62
5.6	Skin Removal with the Scalpel Tool.....	62
5.7	Pasting Objects.....	63
5.8	Combining Layers.....	64
5.9	Deleting Layers.....	66
5.10	Importing Objects	66
5.11	Saving & Reloading Edited Scenes	68
5.12	Summary	69

6.0	Modifying Elements in an Image-Based Scene	70
6.1	Transforming Objects	70
6.2	Painting on Object Surfaces	71
6.2.1	Drawing Events	71
6.2.2	Committing a Paint Operation	73
6.2.3	Rendering the Painted Scene	74
6.2.4	The Erase Paint Operation	75
6.3	Re-illuminating Object Surfaces	76
6.3.1	Finding Normals	76
6.3.2	Painting Light	78
6.4	Summary	79
7.0	Results	80
7.1	Generating Inputs	80
7.2	Example Applications	82
7.3	Performance	84
7.4	Summary	84
8.0	Conclusions & Future Work	85
8.1	Meeting Our Objectives	85
8.2	Improvements to the System	86
8.2.1	Rendering Speed	86
8.2.2	Skin Removal	87
8.2.3	Image Warping	87
8.3	Extensions to our Toolkit	87
8.3.1	Painting Tools	88
8.3.2	Re-illumination Tools	88
8.3.3	Sketching & Modeling Tools	89
8.4	Our System's Contributions	90
	Appendix A Color Plates	91
	Bibliography	95

Table of Figures

1.1	The trial-and-error cycle of traditional 3D editing systems.....	12
1.2	The image-based rendering process.....	14
3.1	The interface	28
3.2	Importing an image-based object.....	29
3.3	Cutting, copying, and pasting.	30
3.4	Manipulating the component images.....	32
3.5	Render modes.....	33
3.6	Painting on the surface of a teapot.....	36
3.7	Re-illumination	37
3.8	Scaling objects in the scene	38
3.9	Skin removal	39
4.1	A pipeline for controlling an IBR system.....	42
4.2	Basic data types in the system	43
4.3	Structures within the tools class.....	44
4.4	Image warping	48
4.5	Filling in holes	49
4.6	Adjusting the Threshold lever to remove skins.....	51
5.1	Intelligent Scissors	55
5.2	Initial configurations for the layerlist and viewlist data types	56
5.3	A layer object.....	56
5.4	Selecting an object with the Lasso tool.....	57
5.5	Initial configurations of the layerlist and the clipboard	58
5.6	Final state of the masks after a cut operation has been completed	60
5.7	The scene no longer contains the selected plate	61
5.8	Changes to the layerlist during a paste operation	63
5.9	Changes to the viewlist during a paste operation.....	63
5.10	A pasted object can be independently moved.....	64
5.11	Combining layers in the layerlist	65
5.12	Combining layers in the viewlist	65
5.13	Examining a combined layer from a different viewpoint	66
5.14	Final state of the masks after an import operation has been completed	67
5.15	An imported object can be manipulated in the scene	68
5.16	Using multiple views to position an object during a scaling operation	71
6.1	Painting with a thick brush	72
6.2	Paint in the scene propagates to the reference images, via masks	74
6.3	Removing paint with the Erase Paint feature.....	75
6.4	Phong shading.....	78
6.5	Diffuse light added to the teapot.....	79
7.1	Laser-scanned inputs.....	80

7.2	Rayshade inputs	81
7.3	An input generated from Alias Wavefront Studio.....	81
7.4	Adding specularities to a scene.....	82
7.5	Compositing.....	83
7.6	Cutting & Compositing.....	83
7.7	The modified statue is saved and imported into an office scene	84
8.1	Specifying depth in a sketching tool.....	89

CHAPTER 1 *Image-Based Editing*

1.1 Control over Imagery

Much research in computer graphics has been devoted to producing realistic images from synthetic data, through an intricate process called *rendering*. Classical rendering involves creating a geometric model of the world we would like to view, specifying the illumination and material properties of the objects in this world, and using a projective mapping to transform our 3D data onto a 2D image composed of pixels. But why should we go through such a complicated and lengthy process to produce realistic imagery, when photographs, which have superior realism, are easier to obtain? Often, we want to generate scenes that would be too difficult or costly to build in real life. In addition, we want the flexibility to easily change aspects of our scene, such as the placement of objects or the lighting conditions. In general, the usefulness of *computer generated imagery* (CGI) is in the control it gives to the user. While real life conditions can only be constrained to a certain degree, a user is free to modify any of the basic parameters in a virtual environment.

Though the user gains a great deal of control using CGI, he or she may find it difficult to obtain photorealistic results. In order to successfully re-create the complexities of the real world, geometry, lighting, and shading models must be defined with the utmost precision. Of course, this

kind of precision is nearly impossible to obtain, especially when we are attempting to model phenomena that are not well understood. As a result, approximations are made by the user or the renderer to simulate real world behavior. For example, users will frequently make use of *texture maps* in their geometric models to add surface detail. A texture map is an image that is applied to the surface of 3D object, like a type of wallpaper. Thus, to create a wooden block, we can apply a wood-like texture to a cube instead of modeling the intricacies of the wood directly. Through approximations such as these, classical graphics renderers have been able to produce fairly realistic images.

Yet, despite these efforts, problems remain in bringing photorealism to CGI. Converting a desired scene to 3D geometry usually involves a long and complex modeling process. The resulting geometric model is often composed of millions of tiny polygons, making it unwieldy for interactive editing. Consequently, most software rendering packages constrain users to a cycle of trial-and-error, as shown in Figure 1.1 below:

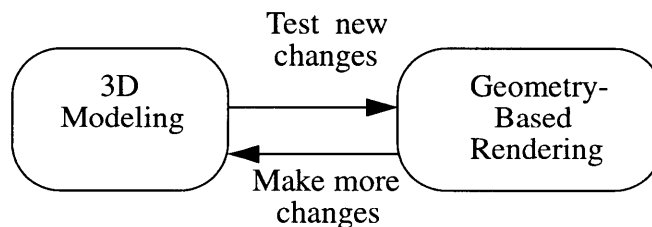


FIGURE 1.1 The trial-and-error cycle of traditional 3D editing systems.

The user makes changes to the scene in a 3D modeling program and must then render it separately to determine whether the changes were appropriate. If more changes are necessary, the scene must be re-edited and the cycle continues. Because massive polygonal data is costly to render, most commercial modeling programs, such as Alias|Wavefront Studio, limit users to edit

wireframe versions of their 3D models [Alias96]. Thus, as a model becomes more detailed and realistic, it also becomes less interactive. Finally, there is a fundamental problem in making the approximations we described above. The estimates made during classical rendering cannot actually describe the physical world, and so the resulting images cannot be truly accurate. For these reasons, we must explore other approaches to photorealism in computer graphics.

There exist some alternatives to the traditional modeling and rendering process. Recently, laser scanning technology has allowed users to acquire geometric models of real objects. A laser scanner works by casting a stripe of lasers on an object, which is simultaneously viewed with a video camera to determine the object's contour. From this information, we can construct a 3D coordinate mesh to represent the object. Laser scanners, however, will often generate a huge amount of data for any given object, again making interactive changes difficult. On the other hand, one could abandon rendering altogether and simply use image processing techniques. Programs such as Adobe Photoshop [Adobe97] allow users to manipulate images, both real and synthetic, to produce a variety of effects. But since image processing software only contains knowledge about the pixels in the image itself, it is difficult to make accurate changes with such programs. For instance, rotating an object in an image or looking at a scene from an arbitrary viewpoint are nearly impossible using pure image processing techniques. Clearly, we would like a system that maintains the flexibility of a 3D modeler but achieves photorealism through less costly means.

1.2 Image-Based Rendering: Flexibility and Photorealism

Image-based rendering (IBR) is a rapidly growing alternative to the classical computer graphics process. Systems that make use of image-based rendering use photometric observations

to generate imagery. In other words, these systems utilize a set of sample images and some information about how these images correspond in order to produce arbitrary views of a scene. Figure 2 illustrates the image-based rendering procedure. Here, a gargoyle statue can be seen from a new perspective by using pixel data from three reference images taken at different viewpoints. Various techniques have been developed to perform this mapping, as we will discuss in Chapter 2.

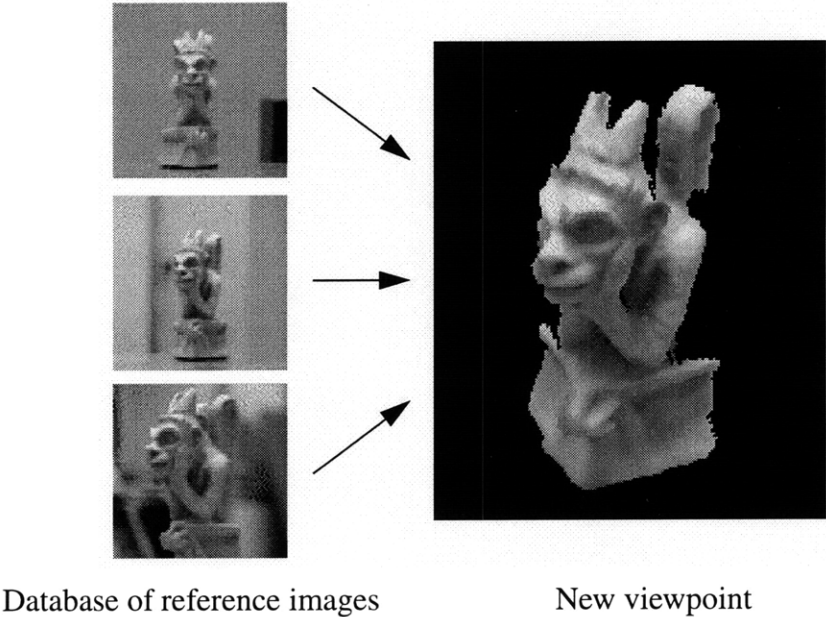


FIGURE 1.2 The image-based rendering process.

This approach contrasts with the one used by classical rendering, which relies on geometry, surface, and lighting models to yield the desired results.

IBR systems have many key advantages over the traditional methods. First, the cost of viewing a scene is independent of its complexity. Even when a scene’s geometric and surface data are difficult to obtain or model, an IBR system can produce convincing results, since it manipulates sample images of a scene and not a 3D representation of it. The use of sample images also means that an image-based renderer can use either photographic or synthetic input. As a result,

we can manipulate rich, photorealistic scenes without any explicit 3D modeling. Finally, image-based rendering techniques generally require less computational power than their geometry-based counterparts, making them accessible to a wide variety of applications.

While the image-based approach holds promise, the problems of manipulating and modifying these representations have not been addressed. Standard editing operations, such as composing a scene from a set of objects, changing the material properties of an element in the scene, or changing the lighting conditions, have not been supported in existing IBR systems. These shortcomings may be due to the fact that scene elements, which would have been distinct in a geometry-based system, are represented indirectly, if at all, in an image-based system. Furthermore, modifications to a scene's structure are difficult to make without reverting to a geometric description. Thus, there exists a need for powerful editing tools in an IBR system that enable the user to manipulate and change the attributes of a scene.

But how would one edit scenes in such a system? Certainly, making individual modifications to all of the reference images in our database can be time-consuming and produce inconsistencies. Instead, we would like to have operations that automatically propagate changes back to the reference images. In this manner, an IBR system would maintain its advantages in speed and realism while still allowing the user to modify a given scene's contents. The focus of our work is to make this form of image-based editing a reality.

1.3 Objectives of our System

A useful editing system must be able to change aspects of an image-based virtual environment to suit our purposes. To this end, we have designed the IBR editing toolkit to meet the following objectives:

- The system will allow interactive editing and navigation of scenes.
- It will be flexible enough so that both photographic and synthetic images can be used as input.
- It will allow the user to manipulate objects in a given scene (i.e. We can cut and paste objects, move them around independently, paint on their surfaces, etc.).
- The system will allow the user to change the illumination in the scene.
- Changes to a scene can be saved and restored.
- All changes are made via modifications to the input images and not through the specification of new geometry.

A system that meets these objectives will make image-based rendering a viable alternative or supplement to geometry-based systems. Image-based editing brings a fundamental set of controls to the user, where none existed before. We are providing the user with a necessary suite of tools for manipulating imagery.

1.4 Potential Applications

There are many possibilities for image-based editing technology. An architect might use this system to try out different building configurations or composite a scale model into a real life scene. An interior decorator could use the system to move lights, furniture, or change the appearance of a room given only a set of images. An IBR editing toolkit could act as a virtual movie or theater set, where lights and camera angles can be adjusted in the pre-production stage. In general, many operations that would be difficult to achieve in real life could be done with relative ease through image-based editing. One could position a car at the edge of a cliff or move a fire hydrant from one side of the street to another while still maintaining photorealism. Furthermore, any

touch-up work one might have performed on several individual images can now be done at once with our editor. In this way, we can extend the photo-editing capabilities of programs such as Adobe Photoshop to deal with multiple images in a consistent manner.

The ease in creating virtual environments for our system might also give rise to image-based clip art. Just as repositories for 3D geometric models and 2D images exist today, sample images corresponding to particular objects and scenes may develop in the future, making the editing process even easier. Rather than acquiring new images for every scene, one could merely reuse image-based objects from an existing archive.

1.5 Outline

In this discussion, we have examined our goals and motivations in building an editing toolkit for IBR systems. The remainder of this document is structured as follows. Chapter 2 surveys related work in image-based rendering and editing systems. Chapter 3 presents an overview of our system, including the various operations available to the user. The implementation details are examined in the following three chapters. Chapter 4 introduces the inputs, data structures, and the image-based rendering algorithms used by our system. Selecting, adding, and removing objects are detailed in Chapter 5. Chapter 6 describes how we modify scene elements through the scaling, re-positioning, painting, and re-illuminating operations. Our results are reported in Chapter 7, and finally, Chapter 8 discusses our conclusions and future work.

CHAPTER 2 *Related Work*

In recent years, many researchers in the field of computer graphics have focused on image-based rendering. We will analyze their various approaches to find a suitable basis for our editing system. We will also examine previous interactive and image-based editing efforts. From this related work, we can gain insight into the problems involved in manipulating image-based scenes.

2.1 Research in Image-Based Rendering

Since image-based rendering is at the core of our system, we will start by investigating the research of others in this area. Note that much of this research focuses on navigating a scene and not on editing its appearance. While the ability to tour a scene is an essential feature, our system must be designed for more than passive interaction. The background information we obtain from this previous work will help us to design the rendering engine behind our editing toolkit.

2.1.1 Movie-Maps

Lippman's *Movie-Map* system [Lippman80] was perhaps the first step toward image-based rendering. The *Movie-Map* made use of pre-acquired video to generate a walkthrough in a particular environment. Its speed was therefore independent of the complexity of the scene, since the system dealt only with video clips. The *Movie-Map* system also allowed for random access to

the video clips, so that users could change their viewpoint as they traversed the scene. However, Movie-Maps required a great deal of space to store the database of video footage. In addition, the system would only allow navigation in areas where footage had been acquired.

2.1.2 View Interpolation

In another approach to image-based rendering, Chen and Williams developed a technique to interpolate between sample images to generate intermediate images [Chen93]. These intermediate images were made by re-mapping pixels from the samples' viewpoints. *View interpolation*, as it was called, assumed that the user could provide camera transformation and range data for each of the sample images. Range data is simply the depth associated with each pixel in an image. Using these depth values, Chen and Williams were able to find a pairwise, pixel-to-pixel correspondence between two sample images. They called this correspondence a *morph map*, and used linear interpolation to generate new images from this map. Though the principles of view interpolation could be applied to real scenes, the system was only demonstrated on synthetic environments. Moreover, this method only found correspondences between a pair of images at a time, limiting its accuracy. Nonetheless, this system developed the concept of using depth to determine a pixel's location in 3D space. This idea has been incorporated into later work, including our editing system.

2.1.3 QuickTime VR

Apple Computer's QuickTime VR application [Chen95] allowed navigation through an image-based virtual environment. The system used cylindrical panoramic projections of its input images. These images, which could be either real or synthetic, together comprised 360-degree

panoramas. Each image was projected onto a cylinder, called an *environment map*. Environment maps let the user vary his or her orientation while the viewing location remains fixed. Thus, QuickTime VR allowed the camera to pan in space. One of the key advantages of the system was its ability to construct a panoramic image from sample images, which could be photographs. The sample images had to contain overlapping regions, which the user identified. QuickTime VR then applied a “stitching” algorithm to merge the component images. The result was one complete 360-degree image. Another type of QuickTime VR movie allowed the user to rotate an object about its axis. Here, sample images taken at regular increments around the perimeter of the object were used to simulate rotation.

2.1.4 View Morphing

A technique known as *view morphing* was suggested by Seitz and Dyer [Seitz96] to handle 3D projective camera and scene transformations in images. View morphing generates an in-between view from a pair of reference images by using projective geometry to re-map pixels. That is, given that we know the viewpoints of the reference images, we can compute an in-between viewpoint by interpolation. First, the user specifies corresponding points on a pair of reference images. He or she also provides projection matrices that correspond to these images. A projection matrix can be found for any image if we know the viewpoint from which the image was captured as well as some characteristics of the camera that captured it. The camera, like the images themselves, can be real or synthetic. We can then derive camera matrices based on the reference images’ projection matrices. Next, we apply the inverse of these camera matrices to the images. Finally, we linearly interpolate the positions and colors of corresponding points in the two images and apply a new camera matrix. This last matrix maps pixels to the desired view-

point. No knowledge of 3D shape was needed to perform this morph, so photographic input could be used. View morphing is also limited only to pairwise mapping, but its projective warping techniques are useful for our renderer as well.

2.1.5 Plenoptic Modeling

In recent years, IBR systems have incorporated methods derived from computer vision research. We can describe mathematically everything that is visible from a given point using a procedure called the *plenoptic function* [Adelson91]. McMillan and Bishop developed an image-based rendering system [McMillan95a] that sampled and reconstructed the plenoptic function for a given scene. The user specifies a pair of corresponding points in two input images. These reference images are projected onto cylinders. Then, an epipolar geometry for each of the cylinders is determined using the user's correspondences. Epipolar geometry is a collection of curves that make up the image flow field. This means that if a point falls on a curve in the first image, it is constrained to fall on the corresponding curve in the second image. Using this result, McMillan and Bishop were able to reconstruct the plenoptic function for a scene and render new viewpoints arbitrarily.

2.1.6 Light Field Rendering & The Lumigraph

Levoy and Hanrahan's *light field rendering* system [Levoy96] also generated views from arbitrary camera positions using a set of reference images. This system, however, made use of a large number of input images that were resampled and recombined to produce new views. The reference images represented 2D slices of a four-dimensional function called the light field. Light fields are the radiance at points in a given direction, and can be generated from an array of images.

This array takes the form of a light slab, representing a light beam entering one quadrilateral and exiting another. By placing our camera's center of projection at the sample locations on one these quadrilaterals, we can generate the light slab. The system then resamples a 2D slice of lines from the slab to produce new images. A similar process of resampling a 4D function called the *Lumigraph* was used by Gortler, Grzeszczuk, Szeliski, and Cohen [Gortler96]. Both of these systems assume a large database of either real or computer-generated images, but do not require depth information. In addition, light field and Lumigraph representations do not need to compute correspondences between pixels. While these rendering algorithms are computationally inexpensive, they do not contain any explicit notion of the scene that is being displayed. As a result, it is difficult to integrate editing operations into this scheme. One would need to determine how light rays are affected by scene modifications and how these rays should be redefined to preserve changes made by the user.

2.1.7 Shape as a Perturbation of Projective Mapping

The rendering approach we use most closely resembles that described by McMillan and Bishop in *Shape as a Perturbation of Projective Mapping* [McMillan95b]. Here, a 2D image-warping function is derived to represent shape in a three-dimensional scene. The warping algorithm determines the location of a reference image pixel in 3D space by using the pixel's disparity value and a perspective mapping. Disparity is a form of depth information, which must be supplied to the system. A perspective mapping is a two-dimensional transform that relates projections of a 3D surface. Such a mapping can be found from the projection matrix associated with a reference image. Thus, if the projection matrix and disparity values associated with an image can

be specified, the warping algorithm can map each pixel in the image to a 3D coordinate. The collection of coordinates from a set of reference images comprises the overall shape of the scene.

Because this method uses an explicit and reversible function to map pixels from image space to world space, it is a good candidate for our system. We would like a relatively simple procedure to determine which pixels in the reference images need to be modified in order to preserve a user's editing operations. McMillan and Bishop's two-dimensional image warp serves this purpose.

2.2 Research in Interactive Editing

There exists a long history of research into making applications that allow interactive editing in a scene. We will introduce a few examples that are particularly relevant for image-based editing: Adobe Photoshop [Adobe97], Hanrahan and Haeberli's WYSIWYG painting [Hanrahan90], and Schoeneman et al's painting with light [Schoeneman93]. These efforts serve as models for the kind of interaction we would like to achieve in our system.

2.2.1 Adobe Photoshop

Photoshop [Adobe97], by Adobe Systems Incorporated, has become one of the standard tools for manipulating images in photo design and production. The program enables users to touch-up, composite, write text, cut, paste, paint, and add special effects to either photographic or computer-generated images. Photoshop lets users select regions of an image with lasso, box, and snake tools. These regions can then be isolated into separate *layers*. Users can make modifications to a layer without changing the contents of any other part of the image. When all the changes have been made, the layers are recombined into a single image. The user may also paint

on the image with brushes of different sizes and opacity. Because the Photoshop interface is relatively intuitive, our system will make use of similar constructs. In image-based editing, users should be able to make changes to a scene just as they would edit an image in Photoshop, without any direct knowledge of how the actual operations work.

2.2.2 WYSIWYG Painting

The traditional trial-and-error approach to 3D editing led Hanrahan and Haerberli to develop a system for interactive painting [Hanrahan90]. The goal of this work was to develop a WYSIWYG (What You See Is What You Get) editor. The program allowed users to apply a pigment to the surface of an object. The pigment's color, specularity, roughness, and thickness could be controlled by the user to achieve different effects. Once the user was done editing, the system outputted the painted regions as texture maps on the surface of the object. While this system was not image-based, we would like to incorporate a comparable form of interactivity in our toolkit. Particularly for painting operations, the user should be able to modify objects in the scene and not have to wait for a long period of time to determine whether or not to make more changes.

2.2.3 Painting with Light

Schoeneman, Dorsey, Smits, Arvo, and Greenberg developed an interactive system [Schoeneman93] for lighting design that also empowered the user with direct control over virtual environments. In this system, the user paints illumination on objects in the scene, and the program finds the light intensities and colors that best match the painted regions. Thus, the system solves the inverse problem of finding a light setting that fits the description of what we would like to observe in the scene. The *painting with light* approach has a great deal of relevance to image-

based editing. Other methods of re-illumination usually involve specifying a geometric notion of the scene's lights. Since it is difficult to preserve such information in an IBR system, we instead will choose to re-light a scene by painting light onto the surface of objects. Moreover, this procedure allows the user to immediately view his or her lighting changes.

2.3 Research in Image-Based Editing

Our final two examples, Zakai and Rappoport's technique for producing effects on still images [Zakai96] and Seitz and Kutulakos's plenoptic editing system [Seitz98] are direct antecedents of our work. Both systems develop 3D editing operations for real scenes and establish an interactive form of IBR. While the structure of our toolkit will be very different, some of the basic functionality of our system was inspired by these concepts.

2.3.1 3D Modeling and Effects on Still Images

Zakai and Rappoport attempted to bring three-dimensionality to image processing [Zakai96]. Their system let users define and manipulate 3D objects in a still image, such as a photograph. First, the user adjusts the appearance of a simple object, a cube, for example, so that it is correctly projects on a photograph. This adjustment process defines a projection matrix for the image, which contains the camera parameters of the scene. Next, the cube is removed and the user creates a 3D polyhedron around an object that he or she would like to edit. Now the system has a simple geometry for the object, which can be used for cutting and pasting, motion blur effects, and deformations. Essentially, the system has provided a means for texture-mapping parts of the image onto polygonal objects. These objects roughly represent actual elements in the image. Lighting effects can be achieved by modifying the textures associated with an object.

Zakai and Rappoport's technique works well with individual images, but was not designed as an IBR system. Its rendering engine is geometry-based, and does not work with multiple, corresponding images. Furthermore, the system adds 3D geometry to the image, making the cost of editing dependent on the complexity of the scene. While we can achieve photorealism with this approach, we also want the flexibility that an IBR system can provide.

2.3.2 Plenoptic Image Editing

Seitz and Kutulakos's work in plenoptic image editing [Seitz98] perhaps most closely matches our own goals. In this system, changes that are made to one reference image get propagated to the other reference images that comprise a scene. In a typical interaction, the user makes a change to one of the sample images. This change is reinterpreted as a variation in the plenoptic function that governs the scene. The system then decomposes the plenoptic function into its shape and radiance components. These components are modified to reflect the change, and a new version of the sample images is displayed. Hence, if the images were subsequently used as inputs to an image-based renderer, the edit can be observed as a consistent effect on the scene. Using this idea, Seitz and Kutulakos implemented a set of operations that one could use to modify a scene's plenoptic function. The operations included: painting, cutting and pasting, and morphing. Our system differs in that editing will be done primarily in the scene and not in the separate images that make up the scene. Also, there are several operations that will influence the plenoptic function in ways that are difficult to discern. In order to incorporate re-illumination and compositing from multiple scenes into this type of editor, we would need to use a more complex process of plenoptic decomposition. As a result, our IBR system will simply encode more of a scene's structure, so that it can be used during the editing process.

2.4 Summary

From this survey of related work, we have gained the necessary background to pursue the editing problem in image-based rendering systems. Much research has been dedicated to different aspects of the problem, but few have attempted to integrate these features into a cohesive system. What makes the goal of editing in an IBR system so compelling is the unique and useful nature of the work itself.

This chapter will introduce our toolkit for editing image-based rendering objects. We will briefly describe the system as a whole and the different operations available to the user. This section will serve mainly as an overview of the toolkit from the perspective of the user. The following chapters will discuss implementation details.

3.1 Interface

Figure 3.1 shows the interface to our system. The interface is also shown in Color Plate 1.

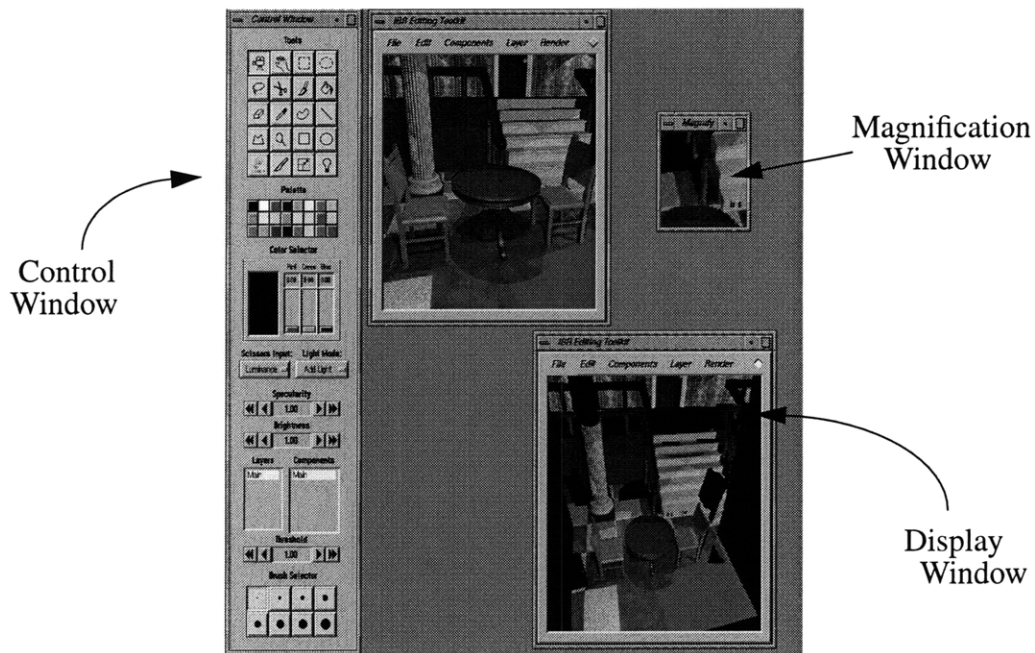


FIGURE 3.1 The interface.

Our program allows users to look at the scene in multiple Display Windows from different points of view. A Magnification Window is provided so that we can examine the fine details of a particular region in the scene. Most of the interaction with the system is done through pop-up menus and the Control Window to the left. We will discuss each of these functions in turn.

3.2 File Menu

The File menu can be used to access to new inputs, save the existing scene, close the current window, or exit the program. Most of these features are self-explanatory, but the *Import* option deserves further discussion. With this function, we can select a new file for compositing. In Figure 3.2a, we see an office, which represents our current scene. We can then import into the scene another image-based scene or object, like the gargoyle statue shown in Figure 3.2b. After an import operation, the gargoyle appears in the same environment as the office, and the user is free to position the statue in the scene. Figure 3.2c shows the result of placing the gargoyle on top of the computer.

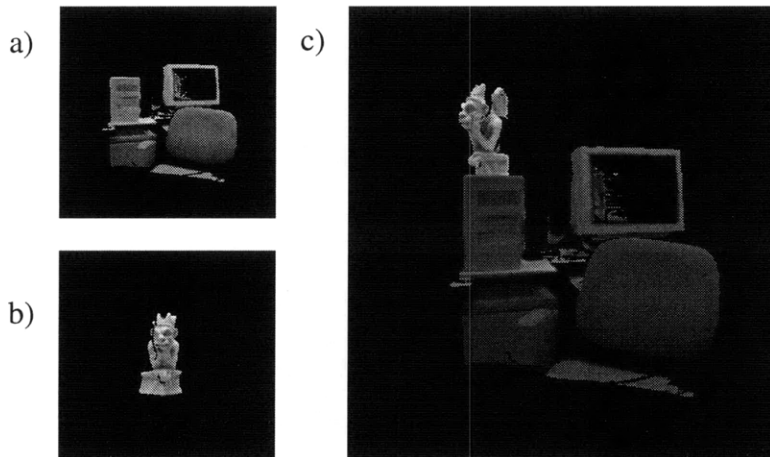


FIGURE 3.2 Importing an image-based object. a) The original scene. b) The object to be imported. c) The resulting scene.

3.3 Edit Menu

As one might expect, all of the menu-driven editing operations are called with the Edit Menu. The options here allow the user to copy, cut, and paste objects in the scene, and erase paint that has been applied to an object. This menu also lets the user reset the camera to its initial position. We will not explore this operation in detail, since its purpose is straightforward.

3.3.1 Copying, Cutting, & Pasting

One of the major features of our system is its capability to duplicate or remove regions in the virtual environment. Figure 3.3 provides an example of how one might use these operations. Here, the dining room scene is modified in Figure 3.3b when one of the plates is removed. In Figure 3.3c, a plate has been copied, pasted back into the scene, and moved into position on the table. The copying and cutting operations work only on the visible pixels in the scene. Thus, if we want to capture something that is behind another object, we must first remove the closer object. The target then becomes visible and can be selected. Of course, the object that we just removed can itself be pasted back into the scene.

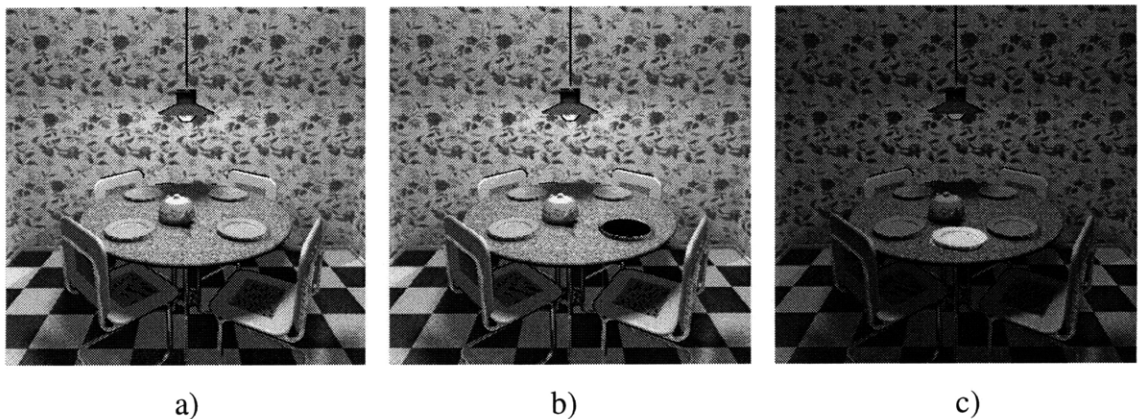


FIGURE 3.3 Cutting, copying, and pasting. a) The original scene. b) The lower right plate is cut out of the table. c) One of the plates is copied and pasted back into the scene. The new plate is then repositioned on the table.

Because a full object is often not completely visible from a given viewpoint, one could also imagine that multiple cuts would be needed to fully remove something. For instance, we may need to apply the cut operation once to get the front side of a teapot and then cut again to remove the back side. To facilitate this type of “group cutting,” we include the options to *Cut More* or *Copy More* from the scene. When we *Cut/Copy More*, we are telling the system to group the currently selected region with the one that already exists in the clipboard. Returning to our hypothetical scenario, the user could use *Cut* to remove the front of the teapot. He or she could then move the camera to a point where the remaining part of the teapot was visible, and use *Cut More* to grab this piece. Finally, when the paste operation is applied, the complete teapot object will appear, and the user is free to move it about the scene.

3.3.2 Erasing Paint

The system will always store the original colors in the scene, even after a painting operation has been committed. At any point, the user can select a previously painted region in the scene and call the *Erase Paint* function to undo the changes. This operation will restore the original colors to this region. The painting tools are discussed in more detail in Section 3.9.

3.4 Component Menu

The Component menu gives access to the reference images that make up a scene. Each of these reference images is referred to as a *component*. When the user executes the *Split Components* command, the system lists all of the scene components in a text box within the Control Window. The user can then toggle the different components on and off to control how many images are being used as input. He or she can also call the *Shade Components* function to re-color the

scene based on the number of reference images. Here, the parts of the scene that are generated from a reference image are assigned some particular hue. Figure 3.4a shows a dining room scene with one of the three reference images toggled off. Notice that much of the wall and the floor underneath the table is missing. In Figure 3.4b, the full dining room is rendered from three reference images, but the *Shade Components* option has been activated.

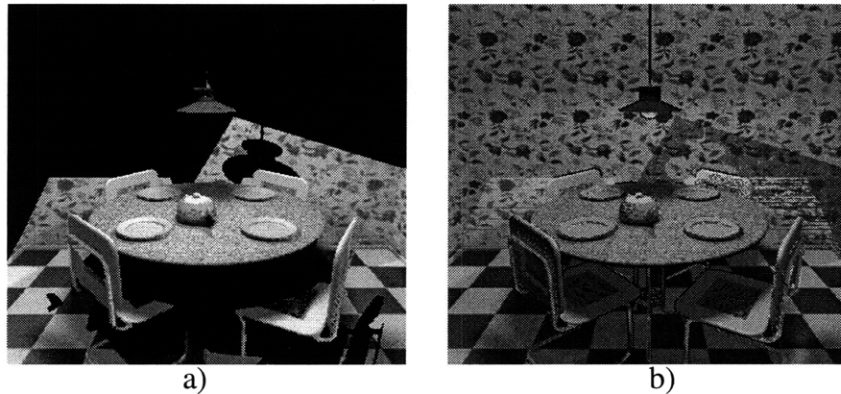


FIGURE 3.4 Manipulating the component images. a) A dining room scene rendered using only two reference images. b) The same dining room scene rendered from three images, with different hues identifying the components.

3.5 Layer Menu

As in Adobe Photoshop, we can apply editing operations to different *layers* which make up a scene. In our system, we begin with an initial layer. A new layer is created whenever we paste or import an object into the current scene. If more than one layer exists in the system, the one that is currently active is shown in full color, while the others are dimmed, as we saw in Figure 3.3c. Multiple layers allow the user to move objects independently. In a previous example, we copied and pasted a plate into the dining room scene. This plate became a new layer in the system so that it could be moved separately from the rest of the dining room. If we were to use the painting or re-illumination tools while the dinner plate layer was active, only this layer would be modified by the system. Thus, layers give us more fine-tuned control over objects in the scene.

The Layer menu manages the layers in our system. Through this menu, the user can toggle the display so that it only renders the current layer and not the full scene. In addition, users can delete layers or combine any subset of them into one layer. Once layers have been combined, the objects they contain are locked into place and behave as if they are grouped together. A text box within the Control Window maintains a list of the layers in the system, and allows the user to switch between them.

3.6 Render Menu

To improve performance we have included three different display modes, accessible through the Render menu. In *Point* mode, only the pixels from the reference images are rendered. In *Triangle* mode, the reference image pixels serve as vertices for a triangular mesh, which is shaded based on the color of the vertices. The reference image pixels are also vertices in *Line* mode. However, this mode draws a wireframe through the vertices instead actually shading triangles between them. Point mode allows for the fastest traversal of a scene, while Triangle mode is the slowest. On the other hand, Triangle mode is best for filling in holes in the scene and making objects appear solid. All three modes are demonstrated in Figure 3.5.

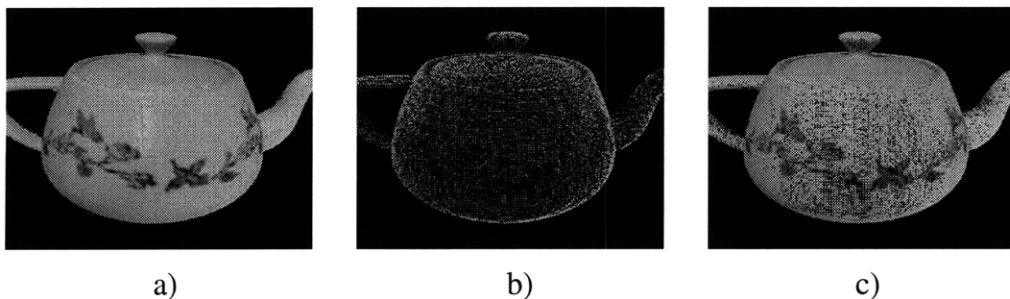


FIGURE 3.5 Render modes. a) A teapot rendered in Triangle mode. b) The same teapot rendered in Point mode. c) Another display, in Line mode.

3.7 Navigation Tools

Our system includes two means of navigation: the *Camera* tool and the *Hand* tool. To move all of the objects in the scene, regardless of which layers they belong to, we use the Camera tool. Essentially, we are moving a global camera to a new viewpoint in the world. If more than one display window is open, the Camera tool can be used to position the scene in any one of the windows. In this manner, we can maintain separate views of a scene in different windows.

The Hand tool only moves objects in the layer that is currently active. This operation can be seen as manipulating a particular object (as opposed to the camera) in 3D space. If more than one display window is open, the Hand tool moves a layer consistently in all views. For example, suppose we are editing a street scene and we have two windows open. Window 1 displays the front side of some buildings while Window 2 displays their back side. Now, suppose we cut out one of these buildings and paste it into a new layer. If we use the Hand tool to move the selected building toward the right in Window 1, the building will correctly move left in Window 2. Hence, the views do not become incongruous while an item is being moved.

Both of the navigation tools allow rotation, translation, and twisting motions. With the Camera tool, the user can pan left or right, move forward or backward, spin on one axis, and rotate about the scene. The Hand tool applies these same transformations to a specified layer.

3.8 Selection Tools

The *Circle*, *Box*, *Lasso*, and *Scissors* tools provide us with the means for selecting regions of the scene on which to operate. The Circle Selection tool lets the user specify an elliptical area for modifications. Similarly, the Box Selection tool creates a rectangular region for editing. The

lasso is a freehand tool that can be used to draw an outline around the desired region. Finally, the Scissors tool automatically finds contours in the scene when the user selects a few points along a given contour. To use the scissors, we would click our mouse at some starting point for the region. As we move the mouse within the Display Window, a selection wire hugs the boundary with the greatest change in pixel intensities. This usually corresponds to the outline of some object in the scene. We can continue clicking on points until the selection wire completely surrounds the desired object. The Scissors tool is particularly useful for selecting irregularly shaped regions that are difficult to isolate by hand.

3.9 Painting Tools

We have included a suite of tools for painting on the surface of objects. To execute a painting operation, the user selects one of the drawing tools, chooses a color and brush size in the Control Window, and applies paint to the scene. When the user has finished painting, he or she pushes the *Commit* button in the Control Window to preserve the color modifications. Now scene navigation is again possible. A Color Selector with red, green, and blue sliders has been included in the system to assist the user in choosing colors. In addition, a palette containing several common color values exists in the Control Window. The user may also use the *Eyedrop* tool to grab the color value from a particular region in the Display Window.

There are several tools to assist in the painting process. The *Box*, *Circle*, and *Line* tools let the user draw these respective shapes on the screen. The *Polygon* tool can be used to form an arbitrary n -sided polygon. The user can select the *Shape* tool to generate any freehand, closed shape. The *Paintbrush* allows general freehand sketching, while the *Paintbucket* can flood fill a closed

region with a particular color. Any of these tools can be adjusted using the Color and Brush Selectors to produce different effects.

Before the Commit button has been activated, the user can always erase any paint they may have added to the scene with the *Eraser* tool. The Brush Selector can also be used with this tool to vary the thickness of the eraser. However, once modifications have been preserved with the Commit button, the eraser can no longer be used on these changes. Instead, the user must select a region where he or she would like to remove paint, and choose Erase Paint from the Edit menu, as described in Section 3.3.2.

Figure 3.6 shows an example usage of the painting tools in our system:

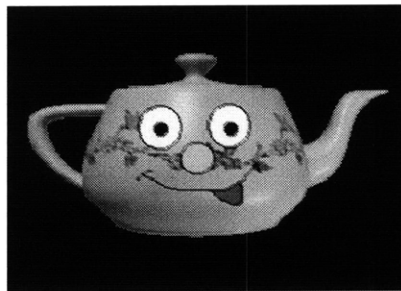


FIGURE 3.6 Painting on the surface of a teapot.

3.10 Magnify Tool

We can change the region currently being displayed in the Magnification Window using the *Magnify* tool. The user simply clicks on the desired region, and the Magnification Window updates. Automatic updating of the Magnification Window occurs during the painting and selection. In these cases, the area surrounding the most recent mouse click is magnified.

3.11 Re-illumination Tools

Lighting effects can be added to a scene using the *Re-illumination* tool. After activating this tool, the user specifies a direction for the light source by clicking at a position in the scene.

Next, he or she uses the Control Window to choose a color, specularity, and brightness level for the light source. If the user selects black as the color of the light source, the system performs diffuse lighting on a pixel-per-pixel basis from the color values already in the scene. Otherwise, the Phong shading model is used. Finally, the user decides to either add or subtract light by setting the Light Mode indicator. The system is now ready for re-lighting. We use a “painting with light” scheme that lets the user paint over the objects he or she wishes to re-illuminate [Schoeneman93]. The paintbrush’s thickness can again be adjusted with the Brush Selector in the Control Window. Lighting calculations are performed interactively, so the changes can be seen immediately after the paint is applied. When the user paints over an object more than once, the desired lighting effect is amplified. Just as with the painting tools, re-illumination effects are preserved when the user pushes the Commit button. They can similarly be erased with the eraser or Erase Paint functions.

Figure 3.7 demonstrates our system’s re-lighting capabilities. In Figure 3.7a, we see a teapot. The teapot has been given a specular highlight in Figure 3.7b, and light from the right side of the object has been removed in Figure 3.7c.

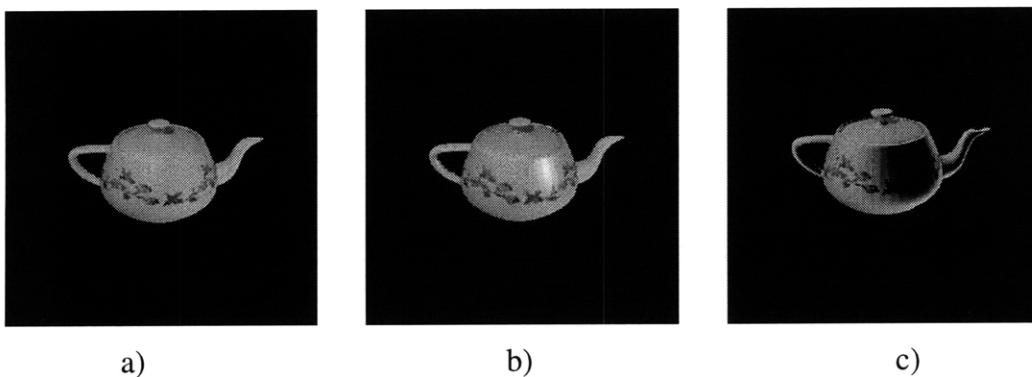


FIGURE 3.7 Re-illumination. a) A teapot. b) The same teapot with a specular effect added. c) The teapot with lighting removed from the right.

3.12 Scaling Tool

Any layer in our scene can be scaled so that its contents appear larger or smaller with respect to the rest of the scene. The *Scaling* tool changes the size of all items in the currently active layer. The tool also performs a translation on these items, pushing them away if they are being made larger and moving them forward if they are being shrunk. Thus, when the user applies a scaling operation, it will appear as if no change has occurred in the current Display Window. However, if other Display Windows are open with different views of the scene, the user will see the re-sizing taking place. Of course, if the scaling operation pushes an object behind something else in the scene, the object will no longer be visible. In Figure 3.8, a gargoyle statue has been imported into an office scene and scaled to different sizes.

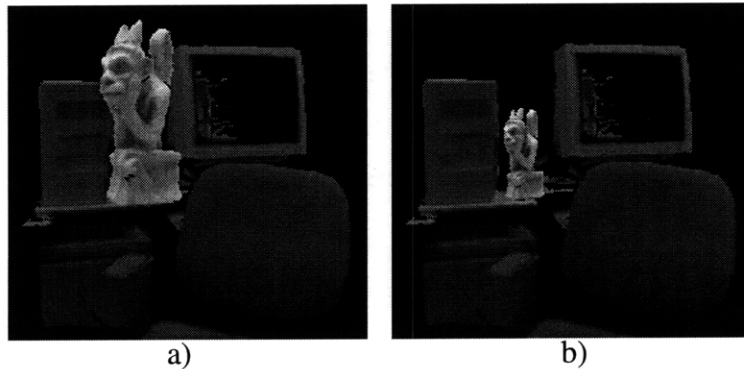


FIGURE 3.8 Scaling objects in the scene. a) A gargoyle statue is made large with respect to the office. b) The gargoyle can be shrunk as well.

3.13 Skin Removal Tools

Because of its image-based nature, our system has no conception of the distinct objects in a scene, aside from those identified as layers. Consequently, when rendering in Triangle mode, the system assumes that each layer is composed of one continuous surface. This assumption will cause some parts of objects to appear connected to other parts, as shown on the teapot handle in

Figure 3.9a. We provide two tools to remove the *skins* that connect distinct objects. The first is a *Scalpel* tool, which lets the user draw a line through the region to be cut. In essence, the user's line will slice off the skin at an object boundary. The second tool is a *Threshold* lever, which can be used to automatically remove skins from the entire scene. Adjusting the lever lets the user control how much material is removed. Figure 3.9b shows a teapot handle after the Threshold lever has been adjusted. Note the skins have now been eliminated.

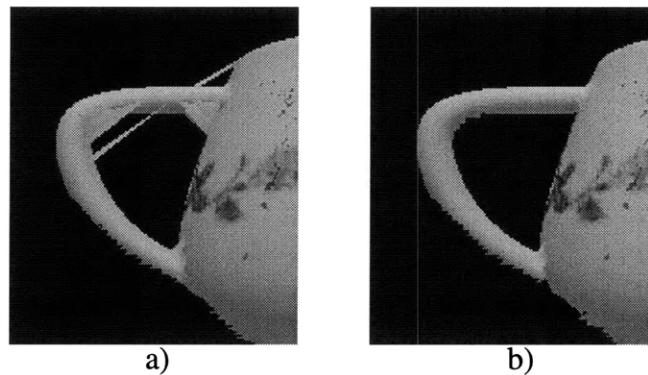


FIGURE 3.9 Skin removal. a) Skins connecting parts of a teapot handle. b) The same teapot handle after the Threshold lever has been adjusted.

3.14 Summary

We have described all of the basic operations available to the user in this chapter. In a typical session, the user would load up a scene, use any of the tools described above to modify its contents, and save the results to a new file. The resulting file can also be loaded into our system for further editing. With this general understanding of the system's functionality, we are ready to explore the details of our implementation.

CHAPTER 4 *The Core of the System*

We begin analysis of our design choices and the technical aspects of our system in this chapter. Specifically, we will describe the system's inputs and how they relate to the overall framework for the toolkit. We will also introduce the data structures and our rendering process. These elements are at the core of our system, and make the editing operations possible.

4.1 Inputs to the System

We have chosen to use reference images, camera, and depth information as inputs to our system. This data will provide us with the structure of a given scene, which we can then modify during the editing process. Our input will be any number of file pairs. One file in this pair will be the reference image itself. The other file will contain a form of depth called range data, the position of the camera in 3D space, and a projection matrix. This projection matrix can be found from the parameters of the camera which captured the image. For example, suppose we have an image of width w and height h captured by a camera at point $eyep$ in space. The camera is looking down the vector \overrightarrow{lookat} , and its orientation is determined by an \overrightarrow{up} vector. The camera also has a field of view given by fov . With these variables, we can determine the following quantities:

$$\overrightarrow{du} = \frac{\overrightarrow{lookat} \times \overrightarrow{up}}{|\overrightarrow{lookat} \times \overrightarrow{up}|} \quad \overrightarrow{dv} = \frac{\overrightarrow{lookat} \times \overrightarrow{du}}{|\overrightarrow{lookat} \times \overrightarrow{du}|} \quad \hat{n} = \overrightarrow{du} \times \overrightarrow{dv}$$

$$k = \frac{w}{2 \tan((fov)/2)}$$

$$\vec{vp} = k \cdot \vec{n} - \left(\frac{w}{2} \cdot \vec{du}\right) - \left(\frac{h}{2} \cdot \vec{dv}\right)$$

Now we compute a scale factor for the projection matrix:

$$scale = \sqrt[3]{\begin{vmatrix} du_x & dv_x & vp_x \\ du_y & dv_y & vp_y \\ du_z & dv_z & vp_z \end{vmatrix}}$$

Finally, our projection matrix P is:

$$P = scale \cdot \begin{bmatrix} du_x & dv_x & vp_x \\ du_y & dv_y & vp_y \\ du_z & dv_z & vp_z \end{bmatrix}$$

Hence, our depth/camera file contains *eyep*, the matrix P , and the associated range data. For convenience, our program actually reads in a text file containing any number of filenames. These filenames correspond to the file pairs described here. The program then parses the individual files and loads the data into the system.

4.2 Acquiring Depth & Camera Information

How do we obtain depth and camera information for an image, given that we require this data for our system to operate? For computer-generated images, the process is relatively straightforward. Depth at each pixel in a synthetic image is computed during the classical rendering procedure. That is, while an image is being produced from 3D geometry, a depth value for each pixel is stored in an entity called the *Z-buffer*. The values in the *Z-buffer* need only be outputted by the renderer after a reference image has been generated. To obtain camera parameters for a synthetic

image, we can look at the geometric data used by the renderer. A modeling program, for instance, will include camera parameters that we can simply record for our own use.

The problem of determining depth and camera data from real images is more complex, however. A user is generally involved to specify correspondences between the reference images. These correspondences can then be used to build up a pseudo-geometry with the depth and camera information we require. The process is similar to McMillan and Bishop's plenoptic modeling [McMillan95a].

We can use a laser scanning device to assist us in obtaining data for real images. Such a device instantly generates range data from a real scene. Thus, we would have a representation that we could use to find depth and camera parameters. A laser scanner makes a real scene become synthetic, for our purposes.

4.3 Overall Framework

From the above discussion, we see that our toolkit is actually part of a larger framework for interacting within an IBR system. Figure 4.1 illustrates this point:

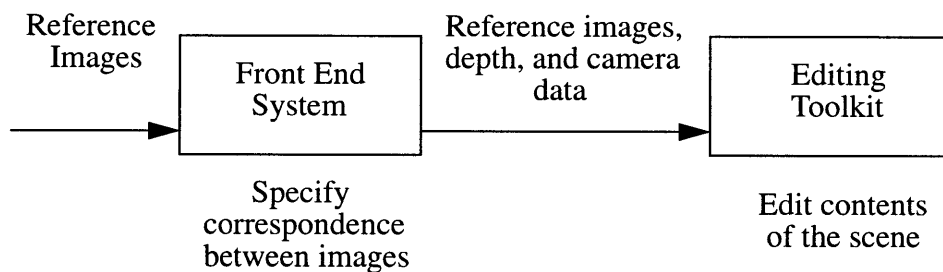


FIGURE 4.1 A pipeline for controlling an IBR system.

Here, reference images are fed into a pipeline. These images are sent through a front-end, which generates depth and camera information. Then, all of this data is fed into our toolkit, where it is

used to make changes to the scene. The front-end system will be treated as a black box. For real images, it can involve sophisticated correspondence techniques. For computer-generated scenes, it could employ more direct means for yielding the required parameters.

We propose this scheme as a plan to effectively control image-based data. The pipeline creates a standard method for manipulating a set of related images. With the inclusion of our toolkit, this scheme establishes an elegant way for acquiring, associating, and editing images, both real and synthetic.

4.4 Data Structures

We will now delve into the data types maintained by the toolkit. The following schematic shows how the various structures in our system interact:

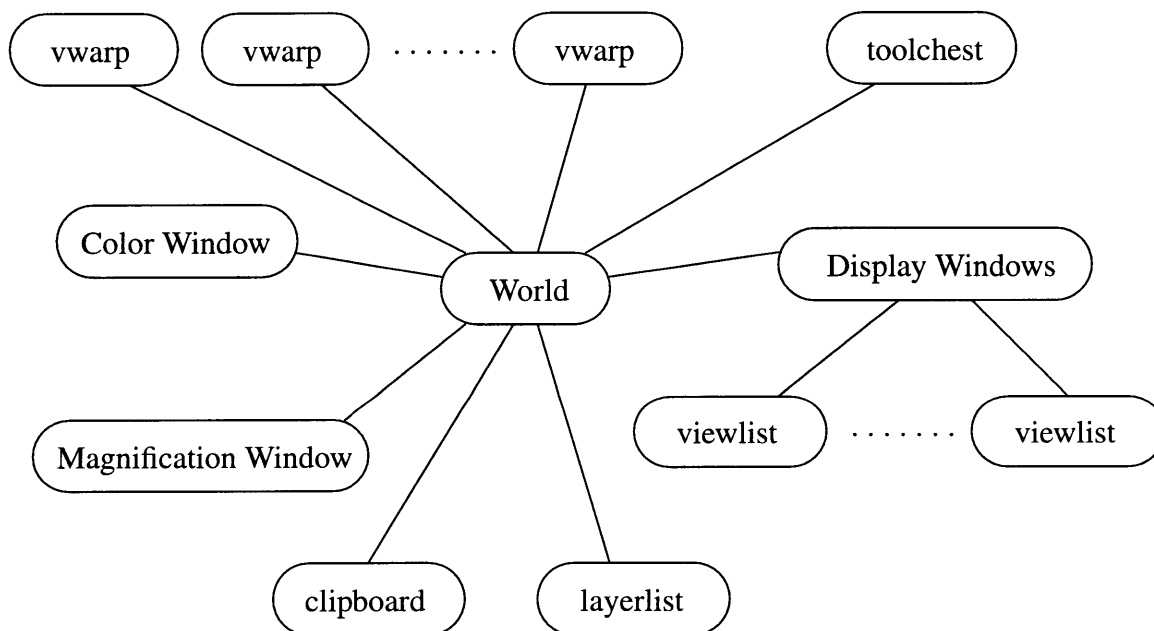


FIGURE 4.2 Basic data types in the system.

4.4.1 The World Class

The *World* class acts as the overall manager. Mouse events and callbacks from the menus and Control Window all get routed through World, which makes changes to the other data types. This class also has access to all of the windows in the system. Note that the Display Windows contain a structure called a *viewlist*. The viewlist holds all of the camera transformations for a particular window. The number of viewlists is therefore equal to the number of open Display Windows. The World class accesses a *clipboard* for cutting, copying, and pasting operations. It also makes use of a *layerlist* to handle the set of layers in our system.

All modifications to the scene are made by World through the layerlist and viewlist. We will describe how these structures change in response to editing operations in Chapters 5 and 6.

4.4.2 The toolchest Class

The *toolchest* class can be broken down into the following units:

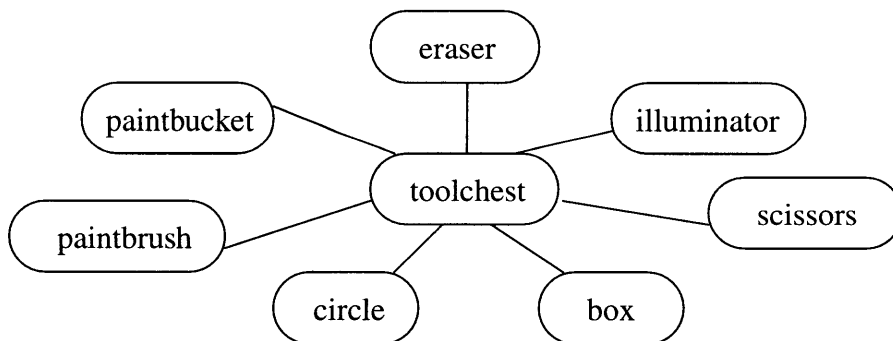


FIGURE 4.3 Structures within the tools class.

These units hold the necessary algorithms for screen updates during object selection, painting, and re-illumination. In other words, we use the various pieces of the toolchest class to draw selection wires, paint, erase, or do re-lighting .

4.4.3 The *vwrap* Class

When the system reads in a file pair, it creates a *vwrap* class. Thus, a *vwrap* holds the information obtained from a reference image and its associated depth/camera data file. We can formally describe a *vwrap* as containing:

- The width w and height h of the reference image
- A buffer called *colors* containing the color values of the reference image at each pixel
- A buffer called *depths* containing the depth values at each pixel
- A view V_0

A view is defined as a camera position *eyep* and a projection matrix P . So V_0 contains all of the required parameters of the camera that captured this image.

If we are using n reference images, the system initializes n *vwraps*. This number may grow depending on whether the user decides to import other image-based objects into the current scene. We use information from the *vwraps* to do scene rendering and editing.

4.5 Image Warping & the Generalized Rendering Algorithm

As stated in Section 2.1.7, we can use McMillan and Bishop's technique to map a reference image pixel into 3D space if we have the pixel's disparity value and the reference image's projection matrix [McMillan95b]. We can readily compute the disparity of a pixel (i, j) from the data stored in its *vwrap*, as shown below:

$$\hat{x} = \begin{bmatrix} i \\ j \\ 1 \end{bmatrix} \quad \text{disparity}[i, j] = \frac{|P \cdot \hat{x}|}{\text{depths}[i, j]}$$

Now, to place pixel (i, j) in 3D space, we apply the following transformation:

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = C_0^{-1} \cdot \begin{bmatrix} i \\ j \\ 1 \\ \frac{disparity[i, j]}{1} \end{bmatrix}$$

where C_0^{-1} is a 4 x 4 matrix defined as:

$$C_0^{-1} = \begin{bmatrix} P_{00} & P_{01} & eyep_x & P_{02} \\ P_{10} & P_{11} & eyep_y & P_{12} \\ P_{20} & P_{21} & eyep_z & P_{22} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Here, we use the term C_0^{-1} because we are applying an inverse camera matrix to the given pixel.

The point $(x/w, y/w, z/w)$ is the desired projection of (i, j) . The pixel coordinates have now been transformed into “world space” coordinates.

The above process projects pixels with respect to the view V_0 . Now suppose we change the viewpoint of the scene. We will have a new view V_1 to represent this viewpoint with a different camera position $eyep'$ and projection matrix P' . How do we reproject the pixels in the reference images so that they appear from the perspective of V_1 ? First, we form a camera matrix C_1

from the new view:

$$C_1 = \begin{bmatrix} P'_{00} & P'_{01} & eyep'_x & P'_{02} \\ P'_{10} & P'_{11} & eyep'_y & P'_{12} \\ P'_{20} & P'_{21} & eyep'_z & P'_{22} \\ 0 & 0 & 1 & 0 \end{bmatrix}^{-1}$$

Then we can project pixel (i, j) as follows:

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = C_1 \cdot C_0^{-1} \cdot \begin{bmatrix} i \\ j \\ 1 \\ \frac{disparity[i, j]}{1} \end{bmatrix}$$

Again, $(x/w, y/w, z/w)$ is the desired projection. We will use the term *image warping* to describe the above transformation from pixel coordinates to 3D coordinates based on a particular view. Observe that when C_I is the identity matrix, we are simply warping pixels into “world space.”

Now we have all the elements necessary to draw the scene. Below is our generalized rendering algorithm for displaying a scene from the perspective of view V_I :

```
for each vwrap vw in the system {
  for each pixel (i,j) {
    use the image warping procedure to produce (x/w,y/w,z/w)
    place (x/w,y/w,z/w) in the scene & color it with vw.colors[i,j]
  }
}
```

In our actual implementation, we take advantage of the fact that we are warping many contiguous pixels at once. Therefore, we can compute the projection of the next pixel incrementally from values stored in the previous warp. This method saves us from performing unnecessary and costly matrix multiplications. Our system also makes use of layers in rendering, as we will detail in Chapter 5.

A few issues remain. How do we determine visibility for all of the coordinates that we place in the scene? Our rendering engine uses a Z-buffer, where depth values for each pixel in the Display Window are written. Thus, while we are computing, if we come across two coordinates that will map to the same pixel in the Display Window, the one closest to the camera gets written in the Z-buffer. When the rendering is complete, the Z-buffer holds values associated with those coordinates that are currently visible.

Finally, how do we insure that coordinates from different reference images line up correctly in our scene? For example, suppose we are rendering a teapot from two images. Can we guarantee that our algorithm will warp both images' pixels from the teapot's spout to the same region in space? If we assume that our depth values are consistent across related images, the answer is yes. Of course, this also means that there may be redundant warping, where two pixels map to approximately the same 3D coordinate. While somewhat inefficient, this redundancy does not significantly hinder the capabilities of the system.

Figure 4.4 shows a scene rendered from three images. Note that our rendering algorithm is indeed image-based. The computation time is based on the number of reference images and the number of pixels in each of these images. Consequently, the scene can be arbitrarily complex and we can still achieve interactive and photorealistic results.

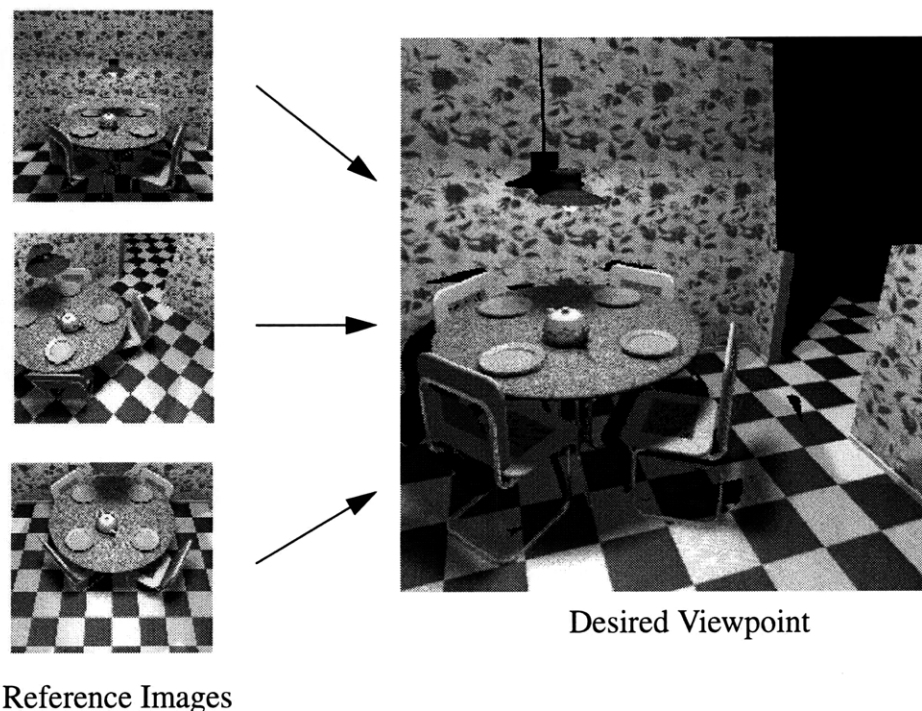


FIGURE 4.4 Image warping. We apply the image warping procedure separately to each reference image. This procedure maps pixels to coordinates based on the desired viewpoint.

4.6 Triangle Meshes

Using the above algorithm and our image warping procedure, we can project pixels from the reference images to a cloud of points in 3D space. We can perform this projection for any arbitrary viewpoint as well. But since we are dealing simply with points, holes may appear when the sampling rates of the reference images and the current viewpoint differ. For instance, if we move the camera very close to a scene, we will see gaps between coordinates as they spread apart.

Figure 4.5 illustrates this point:

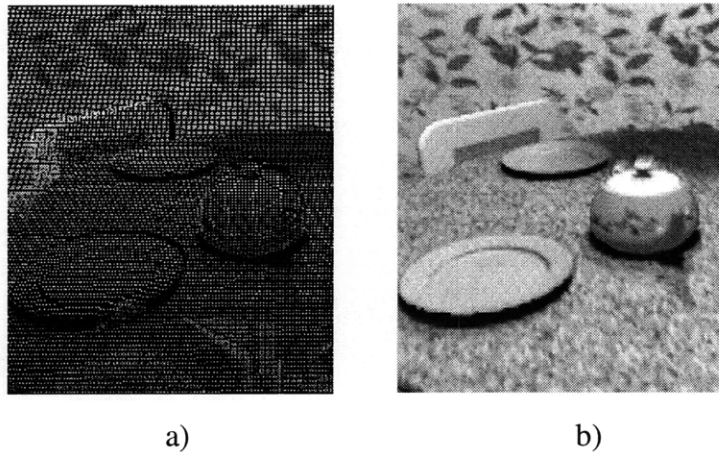


FIGURE 4.5 Filling in holes. a) A scene rendered in Point mode. b) The same scene rendered in Triangle mode.

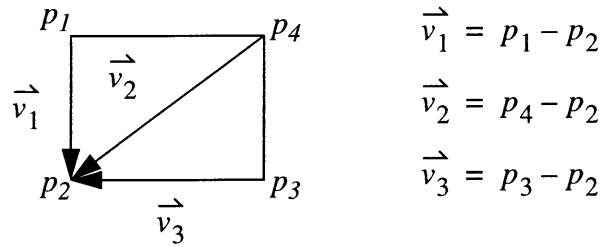
In Figure 4.5a, the system is in Point mode, where only the coordinates themselves are rendered. A similar situation occurs in Line mode, where wireframes become more obvious as their vertices spread apart. To resolve this problem, our system also includes a Triangle mode that generates a triangular mesh out of the coordinates in space. Each triangle in the mesh is shaded by interpolating the colors of its vertices. We use the OpenGL graphics interface language to create and shade the triangles [Woo97]. In Figure 4.5b, our holes have been filled by the mesh, resulting in a more cohesive scene. Because the various modes have different rendering speeds, we allow the user to

switch between them. For a large environment, the user may navigate in Point mode initially to quickly move to a desired position. He or she could then switch to Triangle mode in order to examine a specific part of the scene.

4.7 Skin Removal with the Threshold Lever

As mentioned in Section 3.13, Triangle mode will sometimes create skins between objects that should not be connected in the scene. These skins can be removed automatically with the Threshold lever, located in the Control Window. We will now explain how the Threshold lever performs this operation.

We consider a pair of triangles at a time. Each pair in our mesh is composed of four coordinates p_1, p_2, p_3 , and p_4 . We can define three corresponding vectors \vec{v}_1, \vec{v}_2 , and \vec{v}_3 :



Now suppose p_3 came from projecting pixel (i, j) into space. We can define a ray \hat{r} based on the reference image's projection matrix P . \hat{r} will pass through p_3 :

$$\hat{r} = P \cdot \begin{bmatrix} i \\ j \\ 1 \end{bmatrix}$$

Next, we normalize the ray and the vectors we computed:

$$\hat{r} = \frac{\hat{r}}{\sqrt{\hat{r} \cdot \hat{r}}} \quad \hat{v}_1 = \frac{\vec{v}_1}{\sqrt{\vec{v}_1 \cdot \vec{v}_1}} \quad \hat{v}_2 = \frac{\vec{v}_2}{\sqrt{\vec{v}_2 \cdot \vec{v}_2}} \quad \hat{v}_3 = \frac{\vec{v}_3}{\sqrt{\vec{v}_3 \cdot \vec{v}_3}}$$

Finally, we take the dot product of the ray with each of the vectors:

$$l_1 = \hat{r} \cdot \hat{v}_1$$

$$l_2 = \hat{r} \cdot \hat{v}_2$$

$$l_3 = \hat{r} \cdot \hat{v}_3$$

Here, l_1 , l_2 , and l_3 represent the length of their respective vectors along the direction of the ray.

We compare these values to a threshold, which can be adjusted by the user with the Threshold lever. If l_1 , l_2 , or l_3 is greater than the threshold, we remove the triangle pair from the mesh. In other words, we have decided that the triangle pair is stretched out too far, and could therefore be an unnecessary skin. This process is repeated for each pixel in all of the reference images. With occasional help from the user in finding a suitable threshold, the system can successfully account for discontinuities in a scene, as shown in Figure 4.6.

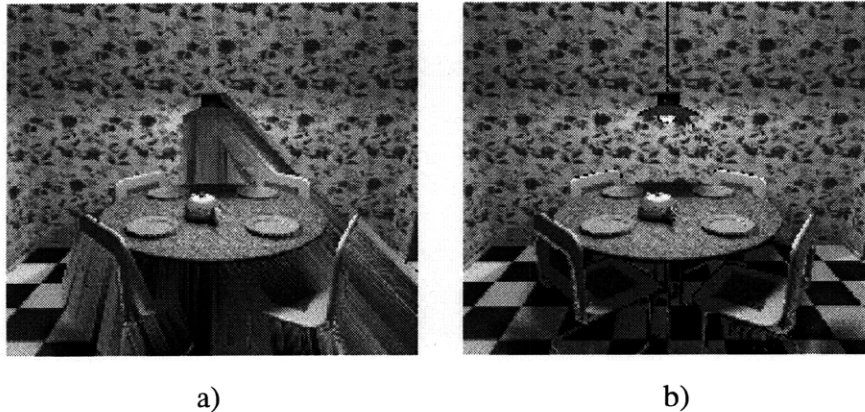


FIGURE 4.6 Adjusting the Threshold lever to remove skins. a) A dining room scene with no thresholding. Notice the skins erroneously connecting the chairs and table to the floor. Similarly, skins connect the lamp to the wall and the table. b) By using the threshold technique, all of the unnecessary skins in the scene are removed.

Before a scene is initially displayed in our system, we apply a default threshold to remove dramatic artifacts, like those seen in Figure 4.6a. Fine-tuning can also be performed with the Scalpel tool, which we will discuss in the next chapter.

4.8 Summary

We have outlined much of the theoretical basis for our system, including its organization, inputs, data structures, and rendering routines. In the process, we have established the groundwork for editing. Now we can finally address how editing operations work in our system.

CHAPTER 5 *Adding & Removing Elements in an Image-Based Scene*

One of the most important set of functions provided by our program is the ability to insert and delete scene elements. We will describe the various methods for selecting items and for copying, cutting, and pasting these items. In addition, we will discuss importing other image-based objects into the current scene and using the Scalpel tool. All of these operations will reveal how our system makes use of layers to preserve user modifications.

5.1 Object Selection

In order to add or remove an object, we must first select it. We will briefly discuss how the system reacts to a selection event. Then, we will describe our implementation of the Scissors tool, which uses Mortensen and Barrett's *intelligent scissors* algorithm [Mortensen95].

5.1.1 Selection Events

The four main tools for selecting a scene element are: the Circle, Box, Lasso, and Scissors. Each of these causes the World class to access a data structure within its toolchest. These units update the Display Window with a selection wire based on the user's mouse movement. For example, when the lasso is being used, a pair of screen coordinates are sent to the paintbrush unit. The coordinates correspond to two different mouse locations. The paintbrush unit then uses Bresenham's line-drawing algorithm to draw a line between the coordinates, which we will call a

selection wire. The user can extend the wire by continuing to move the mouse while holding down its button, which sends more coordinate pairs to the paintbrush unit. Similarly, the circle data structure uses Bresenham's ellipse-drawing algorithm to draw circles and ellipses on the screen. The box unit implements a simple rectangle-drawing procedure.

As we draw a selection wire on the screen, we also mark the pixels that make up the wire in an off-screen buffer. We will use this buffer later to copy and cut the regions inside the wire.

5.1.2 Intelligent Scissors

Often, a complex scene will contain many intricate objects that are difficult to extract with traditional selection tools. Many things we would like to manipulate in our scene are not rectangular or circular. In addition, the user may not be able to draw an accurate freehand boundary for an irregularly-shaped object with the lasso. The Scissors tool uses a unique algorithm developed by Mortensen and Barrett to facilitate precise image segmentation and composition. We will present a high-level description of the algorithm here. For more details, see *Intelligent Scissors for Image Composition* [Mortensen95].

We wish to draw a boundary around the desired object in a scene. Such a boundary can be defined as a dynamic programming problem where the goal is to find the least-cost path from the start pixel to the goal pixel. But how do we determine initial costs for each pixel? Three filtering operations must be applied to the image seen in the Display Window. First, we compute Laplacian zero-crossings for each pixel in the image. A Laplacian filter emphasizes edge features in an image, while a zero-crossing is a binary value based on this filter. We will call this value f_Z . Second, we compute each pixel's gradient magnitude. Here, we apply a filter to determine the partial

derivatives of the image in x and y . Then we calculate a value f_G based on these partials. Finally, we compute a gradient direction f_D for each pixel in the image. The gradient direction is also based on the partials of the image, but adds a smoothness constraint to our costs. Thus, the total cost for each pixel is $w_Z \cdot f_Z + w_G \cdot f_G + w_D \cdot f_D$ where w_Z , w_G , and w_D are pre-determined weights for each feature. Once a cost has been computed for every pixel in the Display Window, we are ready to define a boundary.

To utilize the Scissors, a user clicks on a pixel along the boundary of an object with the mouse. This pixel becomes a *seed point*, as shown in Figure 5.1a. We set the cost of the seed point to zero and perform a graph search on the Display Window image. In this search, new costs are computed for each pixel based on its initial cost and its distance from the seed. Furthermore, each pixel stores a pointer back to its least-cost neighbor. Now, when the user moves the mouse, we can draw a selection wire from the current mouse position back to the seed point by simply following the pointers. In Figure 5.1b, we see the selection wire clings to the boundary of an object. The user can now plant new seed points and repeat this process. Eventually, an accurate, complete boundary will form around the desired object, with relatively little effort on the part of the user.

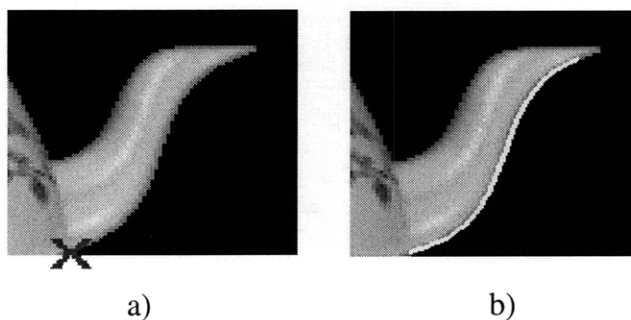


FIGURE 5.1 Intelligent Scissors. a) The “X” denotes the location of a seed point. b) As we move the mouse, a boundary curve forms, hugging the spout of the teapot.

5.2 Layers & Views

Whenever we add or remove a scene element, the World class must update its layerlist and viewlist structures. The layerlist maintains our changes to image data in the system. The viewlist, on the other hand, holds the transformations that we apply to the image data. Figure 5.2 reveals the pieces that make up a layerlist and a viewlist:

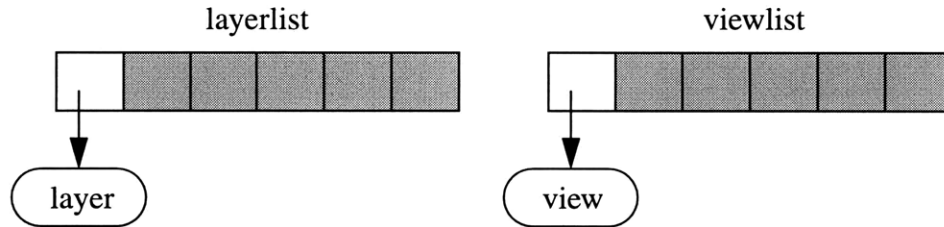


FIGURE 5.2 Initial configurations for the layerlist and viewlist data types.

Initially, a layerlist consists of a single pointer to a *layer* object. The viewlist initializes with a corresponding pointer to a *view* object. A view has the same parameters as the data type we used for V_0 in the *vwrap* class (Section 4.4.2). In fact, our initial view will be V_0 from the first reference image read into the system. Recall that we hold a separate viewlist for each Display Window. Thus, if the user decides to open a new window, a duplicate viewlist is created to manage the transformations that will be applied in this window.

A layer object is shown in Figure 5.3:

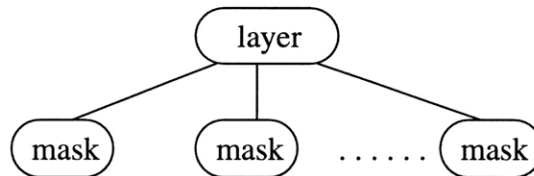


FIGURE 5.3 A layer object.

Each layer contains a set of components which we call *masks*. A mask handles data for a particular reference image. So if there are n vwarps in our system, corresponding to n reference images,

there are also n masks within each layer of the layerlist. Every mask contains an array of bits, one for each pixel in the reference image. The mask also contains another array holding color values.

A mask's bit array controls which reference image pixels will be projected into the scene. When we start the program, all of the bits are turned on, so that every pixel undergoes the warping process. As we edit the scene, however, these bits may change.

5.3 Cutting Objects

We are now ready to examine the events that take place during a cut operation. This section will walk through the stages involved in removing an object. The basic idea is that we can test whether or not to remove a reference image pixel by warping it into the scene. We prevent it from being rendered if the pixel projects into the selected region. For our example scene, we will use a dining room composed of three reference images.

5.3.1 Selection

Suppose the user wants to remove the lower-right plate from the dining room scene. First, he or she draws a wire around the plate with a selection tool. We see this in Figure 5.4 below:

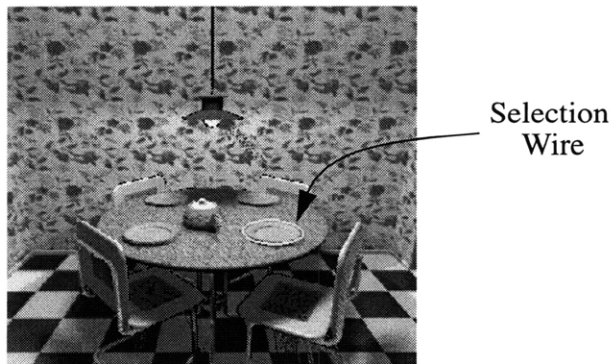


FIGURE 5.4 Selecting an object with the Lasso tool.

Next, the user chooses *Cut* from the Edit menu. At this point, the system has already marked the selection wire pixels in an off-screen buffer. After the user chooses to cut, we use a flood fill algorithm to mark the pixels inside the wire region. The marked pixels map to coordinates that the user would like to remove from the scene.

5.3.2 The clipboard

During a cut operation, the World class makes use of a clipboard, which is simply a pointer to any number of layers. The structure of the clipboard mimics that of the current element in the layerlist. In our example, the current element points only to the initial layer. Therefore, we set the clipboard to contain one layer as well. In this layer, however, we turn off all of the mask bits. Figure 5.5 shows the configuration of our data types at this stage:

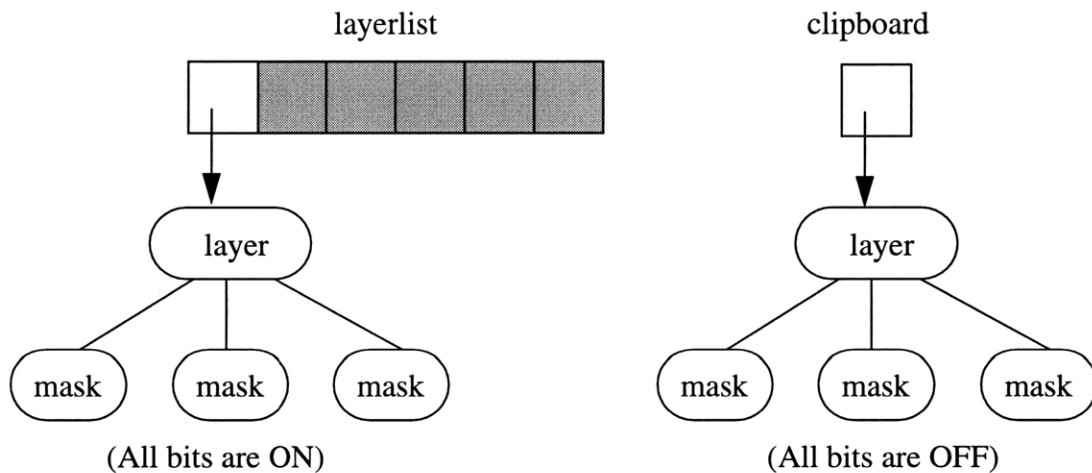


FIGURE 5.5 Initial configurations of the layerlist and the clipboard during a cut operation.

5.3.3 Finding Pixels that Warp to the Selected Region

Now, World pulls out a mask from the layerlist and a corresponding mask from the clipboard. We will call them *layer_m* and *clip_m*, respectively. The Z-buffer of the Display Window is also obtained so that we can access its depth values. All of these parameters are then passed to

the vwrap that holds the appropriate image data. We now apply the image warping procedure from Section 4.5. However, instead of projecting every pixel in the reference image, we warp based on the bits stored in the *layer_m*. If the bit corresponding to a given pixel is on, we proceed as usual, projecting the pixel into 3D space. If the bit is off, the pixel is skipped altogether. Since our dining room example uses an initial mask where all the bits are on, no pixels are skipped.

We proceed to map reference image pixels to 3D coordinates in this fashion. Instead of rendering these coordinates, we round their x and y components to integers, forming a pair (u, v) . (u, v) is a pixel location in the Display Window corresponding to a particular 3D coordinate. Next, we check our off-screen buffer to determine if location (u, v) is marked. If it is not, we discard the coordinate and move on, because it does not map to the user-selected region. If (u, v) is marked off-screen, we test if the pixel is currently visible in the scene. Here, we read location (u, v) in the Z-buffer to obtain a depth value. Then we compare the depth to the z component of our coordinate. If the two values are approximately the same, we have found a reference image pixel that maps to the selected region.

5.3.4 Flipping the bits in the layerlist & clipboard

Suppose that the reference image pixel we found is at location (i, j) . To remove the pixel's contribution to the scene, we turn off the bit (i, j) in *layer_m*. We simultaneously turn on bit (i, j) in *clip_m*. Then we check if there is a color value stored for this pixel in *layer_m*. If there is, that color value is also copied to *clip_m*. When these operations are completed, we have effectively removed the pixel from the scene. At the same time, we have stored it in the clipboard for future use. We continue this process for every pixel.

The entire procedure described here is repeated for each mask in the layer. When we are finished, all of the reference image pixels that would have projected to the selected region are ignored (i.e. The corresponding bits in the layerlist have been turned off). The analogous bits in the clipboard have been turned on. As a result, the current element in the layerlist is exactly the opposite of the clipboard's contents. Figure 5.6 shows the outcome for our example scene:

Masks

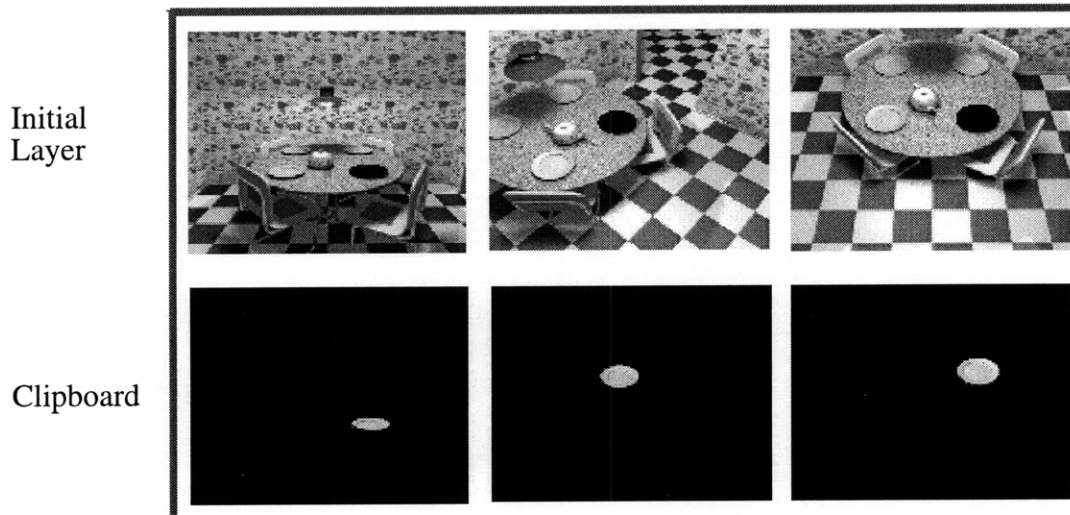


FIGURE 5.6 Final state of the masks after a cut operation has been completed.

Here, the “off” bits are colored black, while the “on” bits are colored based on their corresponding reference images. We see that the dinner plate pixels have been isolated from the rest of the scene, and transferred to the clipboard.

5.3.5 The Revised Rendering Algorithm

The generalized procedure we presented in Section 4.5 to render a scene did not take into account our concept of layers. The following algorithm revises this procedure by placing coordinates in the scene only when the appropriate bit is on. The algorithm also revises the coloring scheme, which we will discuss further in Chapter 6.

```

for each layer lyr in layerlist {
  get the matching view in viewlist
  for each vwrap in the system {
    get the corresponding mask layer_m in lyr
    for each pixel (i,j) {
      if layer_m.bit[i,j] is ON {
        use the image warping procedure to produce (x/w,y/w,z/w)
        if layer_m.colors[i,j] != NULL
          place (x/w,y/w,z/w) & color it with layer_m.colors[i,j]
        else
          place (x/w,y/w,z/w) & color it with vw.colors[i,j]
      }
    }
  }
}

```

For efficiency, we actually store extra information in each mask about where the first and last “on” bits are located. We can save several needless iterations by jumping into the algorithm at the first “on” location and jumping out after the last. We also add special flags to our procedure that allow the system to dim inactive layers or only display the current layer.

With this revised routine, we update the display. The dinner plate pixels will not be projected into the scene, since their corresponding mask bits are off. Figure 5.7 shows the new scene:



FIGURE 5.7 The scene no longer contains the selected plate after a cut operation has been performed.

5.4 Copying Objects

Our process is only slightly different if the user chooses to copy a region instead of removing it from the scene. We follow the steps described above, but we do nothing to the masks in our layerlist. In other words, *Copy* turns on the appropriate bits in the clipboard, but does not modify bits in the existing layers.

5.5 The Cut More & Copy More Operations

Our system enables the user to continue removing or copying material from the scene. He or she can select another region in the Display Window and choose *Cut More* or *Copy More* from the Edit Menu. The implementation of these functions is nearly identical to *Cut* and *Copy*, but instead of starting with a new clipboard that mimics the current layer, we add to the existing clipboard. Thus, a user can remove parts of an object that were missed during the first cutting operation. He or she can even navigate to a better viewpoint to capture these missed regions. Eventually, the complete object will be stored in the clipboard.

5.6 Skin Removal with the Scalpel Tool

Using a technique similar to cutting, the user can hand-remove erroneous skins from the scene with the Scalpel. There are two main differences between this tool and *Cut*. First, the Scalpel removes regions that are only a pixel wide. Second, the tool does not make use of the clipboard. To remove a skin connecting two distinct objects, the user draws over the boundary of one object with the Scalpel tool. The system marks the pixels that the users touches off-screen and then applies the cutting procedure described above. Since this procedure prevents marked pixels

from projecting to 3D coordinates, it also prevents the associated triangles from ever being rendered. As a result, a skin composed of these triangles will disappear from the scene.

5.7 Pasting Objects

Returning to our dining room example, say the user now wants to paste back into the scene the dinner plate he or she just removed. The system copies the information from the clipboard into a new element in the layerlist, as illustrated in Figure 5.8:

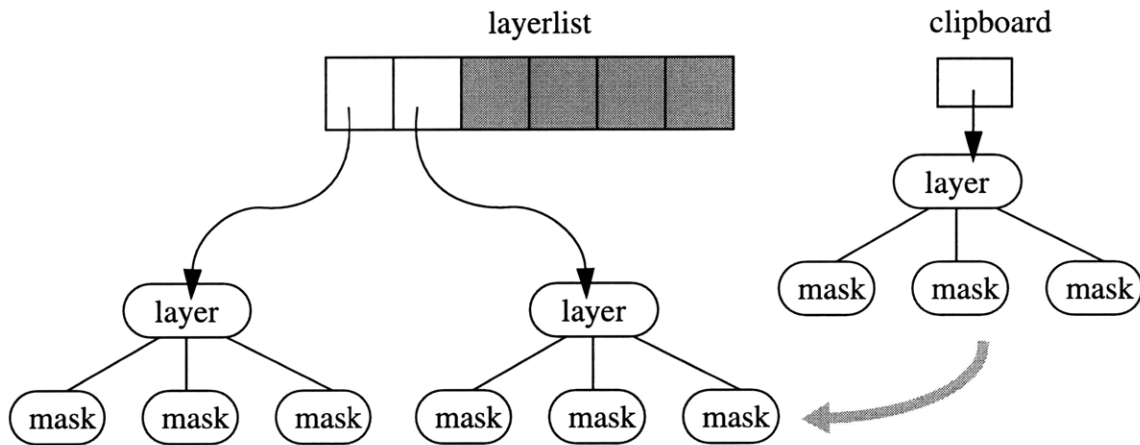


FIGURE 5.8 Changes to the layerlist during a paste operation. Here, the clipboard is copied into a new list element.

By adding this item to our layerlist, we have moved the bits that represent the dinner plate into a new layer, so that the plate can be rendered again.

We must also add an item to each viewlist in the system. The new viewlist element is simply a duplicate of the current element:

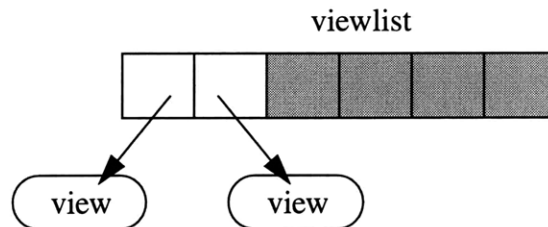


FIGURE 5.9 Changes to the viewlist during a paste operation. A duplicate entry is added to the list.

Now we have the ability to apply transformations to the dinner plate without affecting the rest of the scene.

Finally, the scene is re-rendered, using the algorithm given in Section 5.3.5. Note that each layer in the layerlist has a corresponding view V_i in the viewlist. When rendering a particular layer, we use V_i as our desired viewpoint in image warping. We render each layer successively in this manner. Thus, pixels from the same reference image can be projected to different viewpoints.

As we will discuss in the next chapter, the user is free to move items independently once they have been pasted into a layer. Below, we see the plate has been moved closer to the camera:

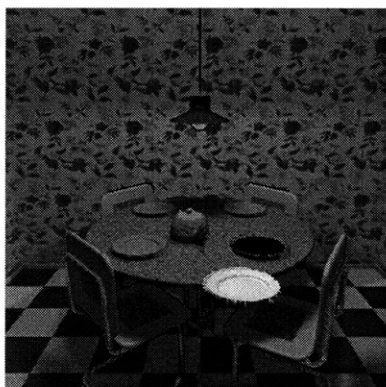


FIGURE 5.10 A pasted object can be independently moved.

5.8 Combining Layers

When the user has finished editing a particular layer, he or she can lock it back into place by selecting the *Combine Layers* option from the Layer menu. Here, the user selects the desired layers, and the system merges them together into a single item. We accomplish this by changing the first affected element in our layerlist so that it points to all of the desired layers. In our example, suppose the user chooses to merge the layer containing the floating dinner plate into the rest of the scene. Figure 5.11 demonstrates how we should modify the layerlist:

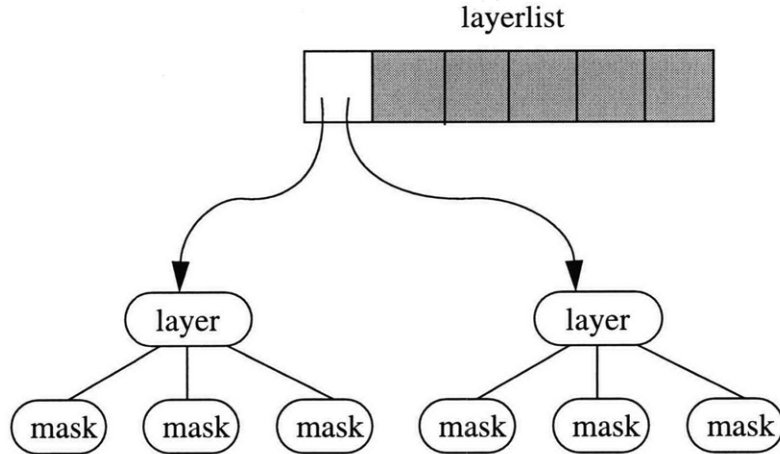


FIGURE 5.11 Combining layers in the layerlist.

Notice how the second element in the list has been removed, and the first element now points to two layers instead of one. We perform a similar modification to the viewlist:

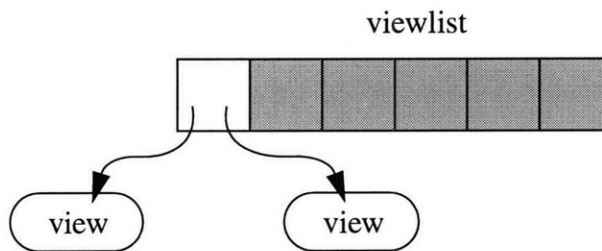


FIGURE 5.12 Combining layers in the viewlist.

Now, if the user decides to cut from the newly-combined layer, we perform the cutting procedure on both layers in our layerlist. In the same manner, if the user chooses to move the newly-combined layer, we make changes to both of views in the viewlist.

Suppose the user decided to combine the two layers in our dining room immediately after the scene from Figure 5.10. The floating plate would now be frozen in place. Figure 5.13 shows the scene from a new viewpoint, where we can observe this result.

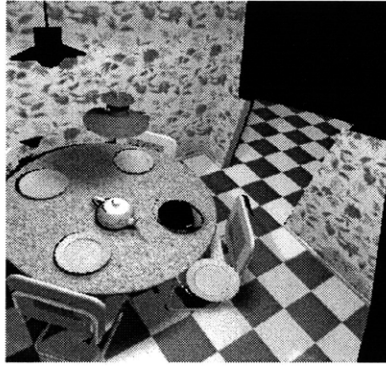


FIGURE 5.13 Examining a combined layer from a different viewpoint.

5.9 Deleting Layers

The user may also delete existing layers. In this case, the system simply removes the layer-list element that corresponds to the user's selection. The analogous viewlist element is also removed. In deleting a layer, we basically revert the system back to its state before the layer was ever pasted.

5.10 Importing Objects

The above operations allow the user to add and remove objects that already exist in the scene. But the user can also import image-based objects from other files. How do our data types adapt to allow this new form of input? Our solution has two stages. First, we enlarge the existing structures to hold the new information. Then, we add the imported object to the system as if it were being pasted into the scene.

We begin by reading the new data into the system and constructing more vwarps to represent the additional reference images. Let us return to our original, unedited version of the dining room scene for this example. Suppose the user decides to bring a teapot into the scene. The teapot consists of one reference image and its associated depth and camera data. The system must

now expand to hold four vwarps: three for the dining room (which had three reference images of its own) and another for the teapot.

Our layerlist structure must also be expanded. Currently, each layer holds only three masks, when it should actually possess four. To correct this discrepancy, we add a new mask to every layer in the layerlist. The new mask should have all of its bits turned off. We turn off these bits to insure that the imported data does not contribute to the system’s existing layers. Instead, we will create a completely new layer for the teapot.

The new layer will paste the teapot into the scene. We construct four masks for the layer. This time, however, we turn off the bits in the masks corresponding to the dining room reference images. Meanwhile, we turn on all the bits in the mask corresponding the teapot image. Thus, the dining room images do not contribute to the new layer at all. Figure 5.14 shows a graphical representation of the masks in our system:

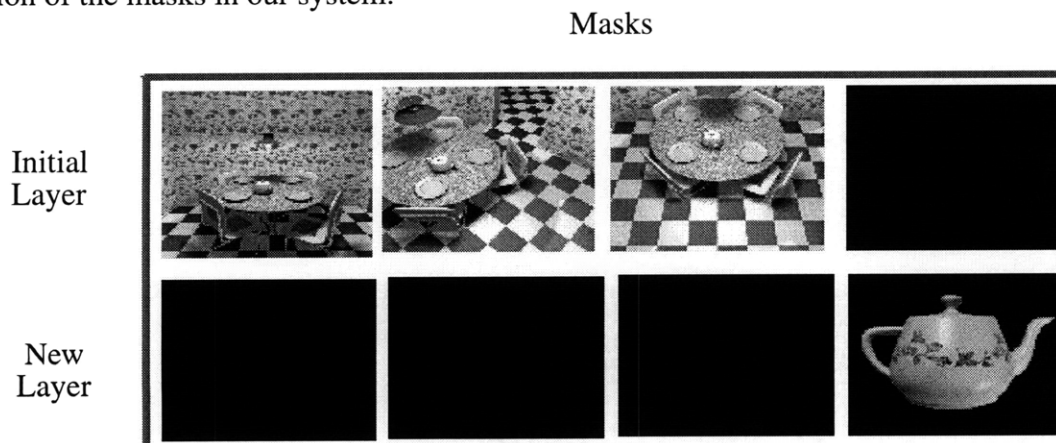


FIGURE 5.14 Final state of the masks after an import operation has been completed.

Like in Figure 5.6, the “off” bits are colored black, while the “on” bits are colored based on their corresponding reference images.

To complete the importing process, we add a new view to the viewlist. Just as we did during a regular paste, the new view is a duplicate of the current one. Now the teapot pixels will successfully project to the camera space of the dining room.

Once the scene is re-rendered, the user will be able to see the teapot and move it about. The following figure shows the teapot positioned on top of one of the dinner plates:



FIGURE 5.15 An imported object can be manipulated in the scene.

5.11 Saving & Reloading Edited Scenes

Any changes made by the user can be preserved by invoking the *Save* function. *Save* writes the layerlist and viewlist structures out to files. To reload an edited scene, we simply input these files, in addition to the original data. We store modifications in separate files so that the original image, depth, and camera information are not tampered with. The user may always return to the unedited scene by not inputting the extra files.

We follow a straightforward routine to open a previously-saved scene. Instead of starting with an initial layerlist and viewlist, we read in these structures from files. Then, we render the scene based on the saved data, and continue handling events in our usual manner.

The process of importing a previously-edited object into a scene is somewhat more complicated. We first expand the existing data structures, as we did in our original importing proce-

dure. Then, we read in the layerlist and viewlist of the object to be imported. We must append these lists to the existing lists in the system. For example, if the current layerlist has two entries, and we import an edited scene with another two elements, the resulting layerlist will have four entries. Similarly, the viewlist will have four entries as well. Note that the appended entries should also be expanded. They must contain empty masks for each of the reference images originally held in the system.

Finally, we need to apply a transformation to the view objects we just added. This is because the imported object will not always project into the camera space of the current scene, which can cause inconsistencies when we navigate. We store the first view displayed by our system as V_{init} . Let us call one of our newly-appended views V_{old} , and the scene's current view $V_{current}$. We can convert these views to camera matrices C_{init} , C_{old} , and $C_{current}$, using the technique from Section 4.5. We then compute C_{new} as follows:

$$C_{new} = C_{current} \cdot C_{init}^{-1} \cdot C_{old}$$

C_{new} can then be converted back into a view object, which will be used as a replacement for V_{old} . This procedure is applied to every view we appended. When we are finished, the new scene will correctly display the imported object in the same coordinate space as the original scene.

5.12 Summary

Through the use of layers, our system supports the addition and removal of scene elements. We can even insert image-based objects from one scene into another. Thus, our layers provide much of the same functionality for IBR worlds as the layers in Adobe Photoshop provide for single images. With the basic means to insert and delete, we can now move on to our final set of editing operations.

CHAPTER 6 *Modifying Elements in an Image-Based Scene*

How can we change the properties of objects in our scene? This chapter will explore operations that transform, paint, and re-illuminate image-based objects. With an understanding of these tools, we will have completed our tour of the system's implementation.

6.1 Transforming Objects

Once we have isolated a region by placing it in a layer, we can apply any arbitrary transformation to that region. A transformation causes the object to translate, rotate, or scale. We bring about this effect by modifying elements in the viewlist.

In the previous chapter, we demonstrated the use of multiple layers by moving a plate in a dining room scene (Section 5.7). But how was this move performed? The dinner plate has been transferred to its own layer. Thus, it has its own element in the layerlist and a corresponding element in the viewlist. Of course, there can be more than one corresponding viewlist element if there are several Display Windows open.

When the user decides to translate, rotate, or scale the plate, the system accesses the view object V pointed to by the current viewlist entry. As detailed in Section 4.5, the view can be converted to a camera matrix C . Now, to apply a transformation, we create a matrix M based on the user's mouse movement and then compute $M \cdot C$. The resulting matrix is converted back into a

view and stored in the viewlist. When the layer is re-rendered, the new viewlist parameters will transform it in relation to the rest of the scene. We repeat this routine for each existing viewlist.

M can be any standard translation, rotation, or scaling matrix. In our system, the camera position, which we called *eyep*, is the only value that changes during a translation. Rotations and scales, on the other hand, modify the projection matrix P associated with a view. Since our system only deals with image and depth data, we have no notion of an object's center of gravity. Consequently, when the user scales an object, the system also performs an inherent translation. The user can open two Display Windows, as shown in Figure 6.1, to deal with this situation. Now the object can be more easily manipulated in the scene.

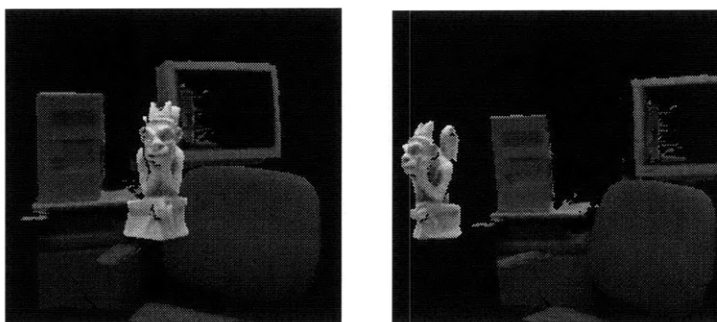


FIGURE 6.1 Using multiple views to position an object during a scaling operation.

6.2 Painting on Object Surfaces

Next, we will look at how painting tools change the properties of a given object. We begin by discussing the various methods for drawing in the Display Window. Then, we study the steps involved in committing a painting operation. Lastly, we will explain the procedure for erasing paint after it has been applied to an object.

6.2.1 Drawing Events

As with selection events, requests by the user to draw on the screen are forwarded to the relevant units within our toolchest class. First, the user sets the Color and Brush Selectors in the

Control Window. The values from these selectors are then passed along to the appropriate toolchest unit when the actual drawing takes place. For example, when the user selects the Box tool and begins drawing, the color, brush size, and endpoints of the desired rectangle are sent to the toolchest's Box unit. This unit contains a box-drawing routine which is used to generate a rectangle in the Display Window.

Similarly, the Circle tool makes use of the toolchest's Circle unit. The Polygon, Shape, Line, and Paintbrush tools, which all have lines as their basic building block, access the toolchest's Paintbrush unit. As mentioned in Chapter 5, this unit implements Bresenham's line-drawing algorithm. The paintbucket, which fills a region with color, calls its namesake in the toolchest to utilize a flood-filling routine.

We can paint shapes of varying thickness by calling the appropriate drawing algorithm multiple times in succession. For instance, suppose our brush has a diameter of four pixels, and the user wants to draw a line on the screen. We first draw the endpoints of the line as circular regions, both four pixels wide. Then, we apply the line-drawing algorithm to connect corresponding pixels in the endpoints. When we are done, a complete brush stroke, with the specified thickness, will appear on-screen. This type of painting is shown in Figure 6.2:

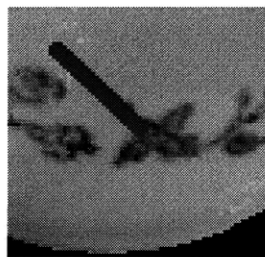


FIGURE 6.2 Painting with a thick brush.

Before the user begins to paint, we store the original color values of every pixel in the Display Window. Later, the user may choose to erase his or her painting. The Eraser works just like the Paintbrush, except that it accesses these original colors. The user is essentially painting the old colors back on to the display.

6.2.2 Committing a Paint Operation

The above routines write new display pixels, but do nothing to the underlying structures in our system. The actual modifications are made when the user pushes the Commit button. This action triggers an update of the system's layerlist, thereby preserving the new color information.

Recall that the masks in each layer contain a bit array and a color array. The color array is initially empty, when no paint has been applied to the scene. Once the user has drawn something, however, we need to fill the relevant arrays with new color values. Our approach is very similar to the technique used in Chapter 5 for cutting and copying in the scene. We warp the reference image pixels to see if they project into the painted area. If they do, we change their color values to match that of the paint.

We begin by marking the pixels off-screen that have been painted by the user. Second, we grab the Z-buffer of the current Display Window so that we can access its depth values at each pixel. Then, for each layer pointed to by the current layerlist entry, we pull out a mask. Both the Z-buffer and the mask are passed to a vwrap containing the pertinent reference image data.

In the vwrap, pixels are projected to coordinates based on the bit array of the mask. That is, we only apply the image warping procedure to a pixel if its corresponding mask bit is turned on. Once we obtain a 3D coordinate, we round its x and y components, forming (u, v) , which is its

location in the Display Window. We check if (u, v) has been marked off-screen, and if this location in the Z-buffer has approximately the same depth as our coordinate. If both of these conditions hold, we have found a reference image pixel that maps to the painted region on-screen.

Say that the reference image pixel we found is at (i, j) . We set location (i, j) in our mask's color array to the same value as pixel (u, v) in the Display Window. This process continues for each pixel and each mask in the current layer. When we are finished, the layerlist contains color values for all of the pixels that project into the painted region.

6.2.3 Rendering the Painted Scene

We now update the display, using the rendering algorithm presented in Section 5.3.5. Here, we project a pixel to a coordinate as dictated by the masks in our layerlist. Suppose pixel (i, j) projects to a point in space. We first check whether entry (i, j) is empty in the color array of the corresponding mask. If it is empty, we use the reference image's original colors, which are stored in the vwrap. Otherwise, we color the point with the value at location (i, j) in the color array.

This rendering scheme, through the use of masks, essentially replaces colors in our reference images. We can see this effect in Figure 6.3:

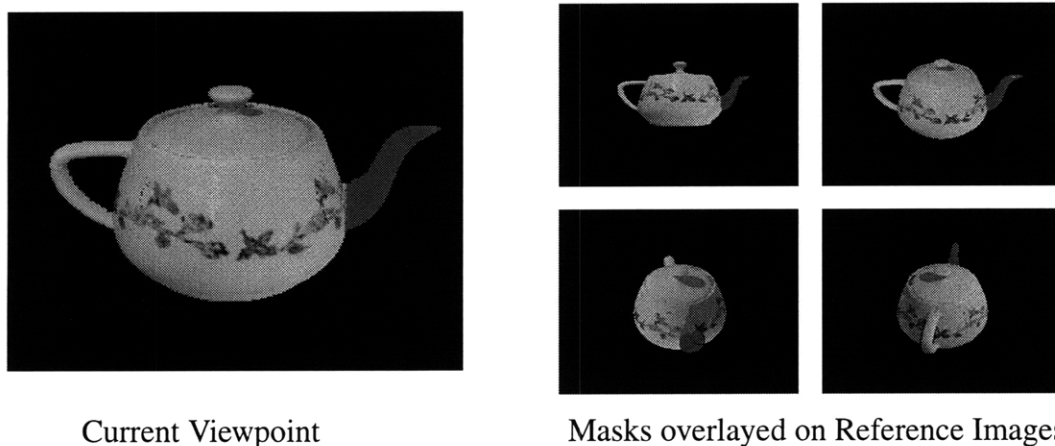


FIGURE 6.3 Paint in the scene propagates to the reference images, via masks.

In this example, the user has painted the spout of a teapot red. On the right-hand side of the figure, we see color data from the masks overlaid on top of the four reference images that comprise the scene. Here, only the nonempty values of a given mask's color array are shown. These are all of the red pixels. Because our system makes use of these colors during the rendering process, the teapot's spout will remain red even when the scene is rendered from a different viewpoint.

6.2.4 The Erase Paint Operation

Since we always hold the original colors of our reference images, the user may erase paint even after pushing the Commit button. To perform this operation, the user defines the region from which to remove paint with one of the selection tools. He or she then calls *Erase Paint* from the Edit Menu. We mark all of the pixels inside the specified region in an off-screen buffer and apply the warping process described in Section 6.2.2. This time, however, when we find a reference image pixel (i, j) that maps to the selected region, we immediately reset entry (i, j) in the color array of the corresponding mask. Since this entry is once again empty, re-rendering the scene will cause the system to use the pixel's original color. In Figure 6.3, the user has erased a section of the spout that was previously painted red:

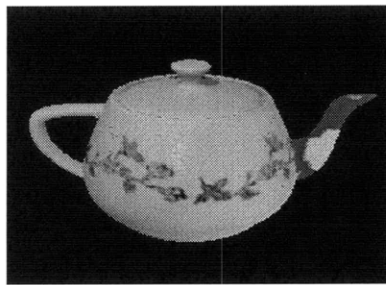


FIGURE 6.4 Removing paint with the *Erase Paint* feature.

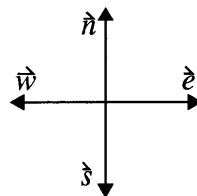
6.3 Re-illuminating Object Surfaces

Finally, we will explore re-illumination of image-based objects. This feature acts as a “smart” painting tool by modifying pixels based on approximate normal vectors. The user “paints light” directly on to the objects in the scene. Since our system encodes depth explicitly, we can compute normals and perform re-lighting in a way that is photometrically consistent. We will show how to generate these normals, and how to determine new color values based on this data.

6.3.1 Finding Normals

When the user selects the Re-illumination tool, our system produces a normal for each pixel in the Display Window. First, the window’s Z-buffer is obtained so we can access its depth values. We use this depth information and the scene’s initial viewpoint to apply the image warping procedure described in Section 4.5. Through this procedure, each Display Window pixel is projected to a 3D coordinate based on the initial view. All of the coordinates are then stored in an array, which we will call *coords*.

To determine the normal for a given pixel (i, j) , we will use the projected coordinate of this pixel and its four neighbors, $(i-1, j)$, $(i+1, j)$, $(i, j+1)$, and $(i, j-1)$. We begin by computing the following vectors:


$$\begin{aligned}\hat{n} &= \text{coords}[i, j + 1] - \text{coords}[i, j] \\ \hat{s} &= \text{coords}[i, j - 1] - \text{coords}[i, j] \\ \hat{w} &= \text{coords}[i - 1, j] - \text{coords}[i, j] \\ \hat{e} &= \text{coords}[i + 1, j] - \text{coords}[i, j]\end{aligned}$$

Next, we read the value of the Control Window’s Threshold lever and consider the pair of vectors (\hat{w}, \hat{n}) . If the length of either \hat{w} or \hat{n} is greater than the threshold value, we set a new vector, $\overrightarrow{result}_{w, n}$, equal to $(0, 0, 0)$. Otherwise, we calculate $\overrightarrow{result}_{w, n}$ as follows:

$$\overrightarrow{result}_{w,n} = \frac{\vec{w} \times \vec{n}}{\sqrt{(\vec{w} \times \vec{n}) \cdot (\vec{w} \times \vec{n})}}$$

We repeat the above routine for the pairs (\vec{e}, \vec{n}) , (\vec{e}, \vec{s}) , and (\vec{w}, \vec{s}) , yielding $\overrightarrow{result}_{e,n}$, $\overrightarrow{result}_{e,s}$, and $\overrightarrow{result}_{w,s}$, respectively. These vectors are then added together:

$$\overrightarrow{result} = \overrightarrow{result}_{w,n} + \overrightarrow{result}_{e,n} + \overrightarrow{result}_{e,s} + \overrightarrow{result}_{w,s}$$

Finally, we normalize \overrightarrow{result} :

$$\overrightarrow{norm} = \frac{\overrightarrow{result}}{\sqrt{\overrightarrow{result} \cdot \overrightarrow{result}}}$$

The vector \overrightarrow{norm} is the normal for pixel (i, j) in the Display Window. We compute the other normals in the same manner.

The process we have just described treats a pixel and two of its neighbors as a triangle in the scene. $\overrightarrow{result}_{w,n}$, $\overrightarrow{result}_{e,n}$, $\overrightarrow{result}_{e,s}$, and $\overrightarrow{result}_{w,s}$ are the normals of four such triangles. We then average these normals together to produce \overrightarrow{norm} . Note that the algorithm correctly ignores triangles that do not pass the threshold test. These triangles correspond to skins that erroneously connect distinct objects in the scene.

As one might expect, we use less coordinates in our computation if we are determining normals for the border pixels in the Display Window. For example, the system only makes use of the coordinates at $(0, 0)$, $(1, 0)$, and $(0, 1)$ when computing a normal for pixel $(0, 0)$.

6.3.2 Painting Light

With normals for each pixel, we can now perform a re-lighting operation that simulates Phong shading. In the Control Window, the user chooses a Light Mode, which indicates whether light is to be added or removed from the scene. Then, the user selects a light color l , a brightness level w , a specular coefficient e , and a preferred normal direction \vec{d} . The latter is chosen by clicking on a pixel in the scene. The normal of that pixel is used as \vec{d} . After specifying these parameters, the user picks a brush size and paints over the area that is to be re-illuminated. Suppose we are adding light to the scene. As each pixel is touched by the user, its color value c is updated to $c + w(\vec{n} \cdot \vec{d})^e \times l$, where \vec{n} is the pixel's normal. When we are subtracting light from the scene, c becomes $c - w(\vec{n} \cdot \vec{d})^e \times l$. In both cases, pixels with normals closest to the direction \vec{d} are most affected by the painting operation. In Figure 6.5, d was chosen near the upper-right of the teapot. Thus, when the user adds light with a high enough specular coefficient to the object, a highlight appears in the upper-right, as shown in Figure 6.5a. In contrast, when the user removes light, the teapot becomes darkest in the same region, as we can see in Figure 6.5b.

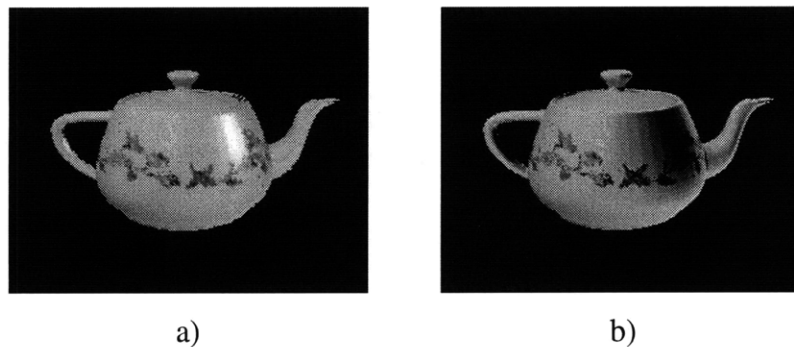


FIGURE 6.5 Phong shading. a) Adding a specular highlight to the upper-right of a teapot. b) Subtracting light from the same region.

The system also performs diffuse shading when the user sets the light color l to black. Here, we use the original pixel's color to change the illumination. Thus, our new color value is $c \pm w(\vec{n} \cdot \vec{d})^e \times c$. Diffuse effects are shown in Figure 6.6:



FIGURE 6.6 Diffuse light added to the teapot.

As with an ordinary painting operation, the re-lighting we just performed only modified the screen pixels and not the data structures within our system. When the user pushes the Commit button, we follow the routine presented in Section 6.2.2 to update the layerlist. Once this has been done, the lighting changes will be preserved.

6.4 Summary

In this chapter, we have addressed how our editing system enables users to transform objects, paint on their surfaces, and change the lighting conditions in the scene. We have now covered all of the major facets of our toolkit. In the remaining chapters, we will present our results and assess the system as a whole.

CHAPTER 7 *Results*

The image-based editing techniques we described in the previous chapters have been successfully implemented in our toolkit. We will elaborate on our methods for generating inputs and report on the performance of the system. We will also present two more examples in this section to demonstrate the editing capabilities of our toolkit.

7.1 Generating Inputs

Laser scanning real objects and rendering synthetic scenes were the two primary means of obtaining inputs for our system. We used a Minolta laser scanner to acquire range and image data for several real world sources, including a gargoyle statue, an office, and a child's head. These inputs are shown in Figure 7.1. Since the laser scanner had a consistent camera model and outputted range data, little work was needed to convert this information to our toolkit's file format. Each object was captured from several viewpoints so that we could have multiple reference images.

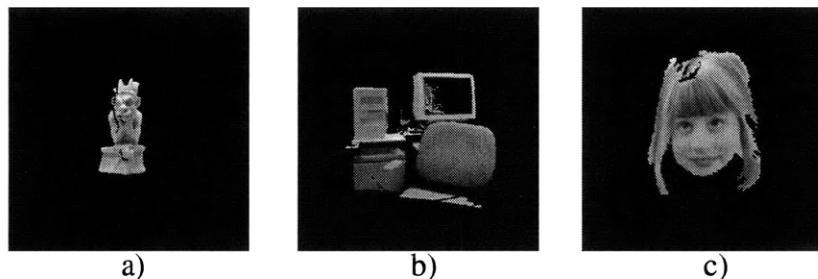


FIGURE 7.1 Laser-scanned inputs. The number of reference images for each scene is given in parentheses. a) A gargoyle statue (3). b) An office (3). c) A child's head (2).

Several inputs were created from computer-generated scenes. We used a modified version of the Rayshade raytracer [Kolb89] to directly output depth and camera data in our format. A raytracer generates this information as it renders an image from a geometric model. The teapot and dining room scenes, shown in Figure 7.2, were produced in this fashion. We raytraced these scenes from different viewpoints as well.

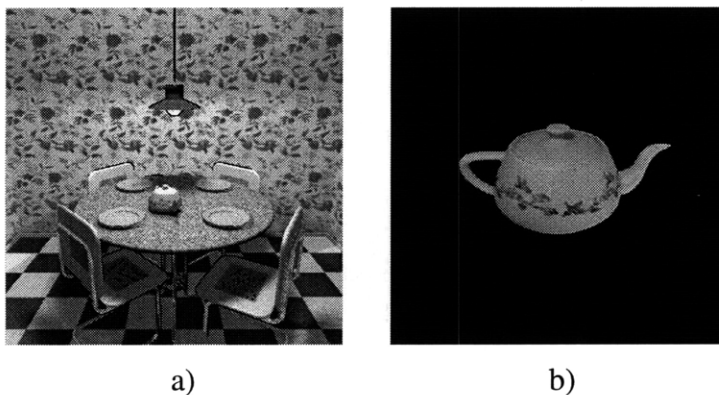


FIGURE 7.2 Rayshade inputs. The number of reference images for each scene is given in parentheses. a) A dining room (3). b) A teapot (4).

Finally, a polygonal model of one scene was constructed with Alias|Wavefront Studio [Alias96]. When this program renders an image from a 3D model, range data and camera parameters can be accessed and converted into inputs for our system. Again, we can render several views to produce multiple reference images.

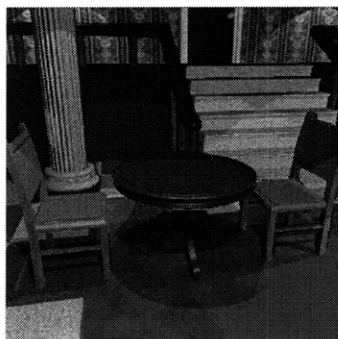


FIGURE 7.3 An input generated from Alias|Wavefront Studio. The scene is composed of two reference images.

7.2 Example Applications

This system achieves our goal of interactive editing in an image-based environment. To show the numerous possibilities open to the user with image-based editing, we present two examples. First, consider the scene in Figure 7.3. By painting with light, we can add specularities to the scene that change correctly with the camera viewpoint. In Figure 7.4, we see the same scene with the new lighting conditions. Note that the table and the handrails have a shiny appearance, due of our added specular effect. This re-illumination is also shown in Color Plate 2.



FIGURE 7.4 Adding specularities to a scene.

Now, we can import our teapot into the scene. In Figure 7.5a, the teapot has been positioned on top of the table. We see from this picture that the lighting appears incorrect. The teapot is too dim in comparison to the rest of the scene, and it casts no shadow on the surface of the table. We can again use the Re-illumination tool to add light to the teapot. Furthermore, we can remove light from the table where the shadow should be located. In Figure 7.5b, we see the updated scene, with more accurate lighting. This compositing operation is also illustrated in Color Plate 3.

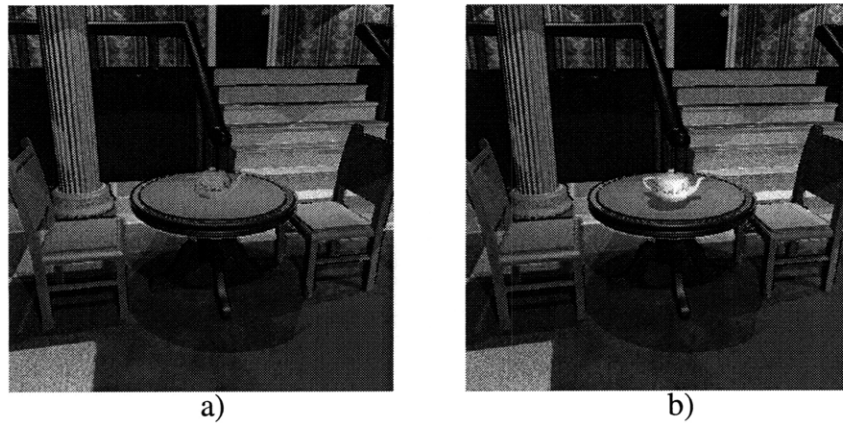


FIGURE 7.5 Compositing. a) A teapot is imported and placed on the table. b) A shadow is added and the teapot is re-illuminated to match the lighting conditions in the scene.

For our final example, we will start with the gargoyle statue, shown in Figure 7.1a. The head of the gargoyle can be removed using the cut operation, as seen in Figure 7.6a. Then, we can import the child's head into the scene and position it on top of the gargoyle's body. The result of this action is shown in Figure 7.6b:

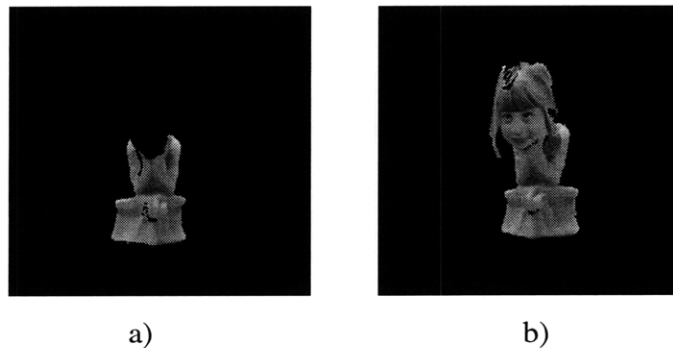


FIGURE 7.6 Cutting & Compositing. a) The head of the gargoyle is cut from the scene. b) A child's head is imported and placed on top of the gargoyle's body.

Suppose we save our changes and then load the office scene from Figure 7.1b into our system. Now, we import the modified statue into the office and position it on top of the computer. The final state of our scene is shown in Figure 7.7. Color Plate 4 also illustrates this sequence of events.

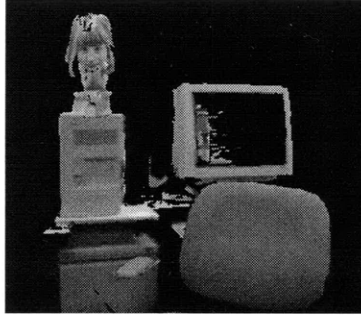


FIGURE 7.7 The modified statue is saved and imported into an office scene.

Clearly, our toolkit provides a variety of methods for manipulating and editing image-based scenes. With the examples in this section, we have demonstrated the usefulness of these methods.

7.3 Performance

Our application was primarily run on a Silicon Graphics workstation with an R10000 MIPS processor. Such a machine provided interactive rates for all of the editing operations. The program's space in memory varied, depending on the number of layers, reference images, and pixels in the system. Our rendering speed was also dependent on these factors. Recent efforts to construct specialized IBR hardware could significantly improve performance for our system.

7.4 Summary

With the above results, we have completed our discussion of the system's semantics. We have also demonstrated that our implementation works, through the use of examples. Finally, we can draw conclusions from this work and address any unresolved issues.

CHAPTER 8 *Conclusions & Future Work*

In this final chapter, we will answer two important questions. First, did we meet our objectives in designing this toolkit? Second, what improvements and extensions can be made to the current system in order to make it more useful? The answers to these questions will allow us to evaluate our system's contributions.

8.1 Meeting Our Objectives

In Chapter 1, a set of objectives was outlined for our editing system. We list them below:

- The system will allow interactive editing and navigation of scenes.
- It will be flexible enough so that both photographic and synthetic images can be used as input.
- It will allow the user to manipulate objects in a given scene (i.e. We can cut and paste objects, move them around independently, paint on their surfaces, etc.).
- The system will allow the user to change the illumination in the scene.
- Changes to a scene can be saved and restored.
- All changes are made via modifications to the input images and not through the specification of new geometry.

From the evidence given in previous chapters, we see that these objectives have been met. Our toolkit allows navigation and editing in an interactive environment. Its inputs can come from real or synthetic data. The program enables users to cut, paste, paint, re-illuminate, and transform scene elements. Modified scenes can be saved and loaded back into the system. Finally, the changes we make to a given scene do not add geometry to our representation.

Note that the process of converting photographic data into usable input can be difficult. Without a laser scanner, we must find correspondences between photographs and use them to build up depth information. This procedure requires assistance from the user to insure that the generated depth values accurately represent the given scene. Despite these complications, photographic input is possible. As computer vision techniques advance, we will be able to more easily obtain depth from real images.

8.2 Improvements to the System

There are aspects of our application that can be improved. In particular, we will investigate methods to better handle rendering, skin removal, and image warping. Our proposed solutions are meant to make the system either more efficient or easier to use.

8.2.1 Rendering Speed

The system may become slow when we are dealing with large reference images or numerous files. Slow performance can occur even though our rendering speed is not based on scene complexity. One possible solution is to parallelize the application, so that it takes advantage of a machine with multiprocessor capabilities. For example, we could assign an image warping task to each processor, and render from several reference images at once. Another possibility is to use

some sort of compression scheme to better manage the color and depth data. As we mentioned in Chapter 4, the system generates redundant coordinates for two reference image pixels that warp to the same location in 3D space. Another way to speed up rendering would be to devise a more efficient algorithm that identifies and ignores redundant pixels.

8.2.2 Skin Removal

While we have provided the Threshold lever and the Scalpel tool to remove unwanted triangles from the scene, these tools can be somewhat imprecise. One way to resolve the problem of erroneous skins is to use another rendering technique, called *splatting*. Instead of generating a mesh of triangles, splatting creates a scene from small, distinct polygons with texture-mapped color values. These polygons will break apart naturally as the user moves very close to an object. Since none of the polygons are actually connected to each other, the scene will not contain any skins. Splatting can be resource-intensive, however, especially when there are many texture-mapping operations.

8.2.3 Image Warping

In the image warping process, there are conversions between projection matrices, which are stored in our data structures, and camera matrices, which are used during computations. For clarity, we could use one consistent representation of the camera parameters for an image. This change would probably not affect the overall performance of the system.

8.3 Extensions to our Toolkit

One can imagine a multitude of other tools and features for an image-based editing system. Every function in an image processing program, such as Adobe Photoshop, could have an

analogue in the domain of image-based rendering. One could also imagine an entirely new suite of techniques that have no analogue in 2D. A tool that fills gaps in a scene's dataset would fit into this category. We will limit our discussion to a few possibilities. Specifically, we will look at additional painting and re-illumination tools, and a sketching tool that has already been implemented as a separate application.

8.3.1 Painting Tools

Our system currently allows the user to only paint flat colors onto object surfaces. Without re-illuminating these objects, we cannot achieve shading effects. One new painting tool could merge the illumination and coloring operations into a single routine, which could then be used to shade an object. We can implement this tool by giving the regular painting procedures access to the normals at each pixel in the display.

We could also add blurring and filtering tools to our system. These might be used to soften boundaries in the scene. In addition, we could make an object transparent by allowing the user to specify an *alpha* value for each pixel that contributed to the object. Alpha values are used to blend the colors of different coordinates that map to the same pixel in the display. Since transparency effects are view-dependent, this blending must occur while our renderer determines visibility in the scene. In other words, we must handle transparent pixels when we sort the Z-buffer.

8.3.2 Re-illumination Tools

Additional lighting tools might also prove useful. For instance, our system does not allow the user to re-position lights or cast light directly from a source. Instead, we paint light on objects. A new set of tools could give the user the ability to switch between object-based and

source-based lighting operations. The light sources could exist only during editing, so that no geometric data is created by the system.

8.3.3 Sketching & Modeling Tools

Often, we would like to add material to a scene where it may not currently exist. The ability to sketch or model new scene elements would allow the user to fill in gaps caused by a lack of data. It would also give the user the power to add any arbitrary object to a scene. For example, sketching tools enable us to draw a building into its proposed location to see how it would look in the scene. In this manner, we can visualize an object without having to explicitly acquire image, depth, and camera data.

Steven Gortler has developed an image-based sketching tool, in work concurrent to our own. The tool lets the user draw a new scene element, specify its depth, and observe the element from different viewpoints. We create an object by first drawing it on the screen and identifying a few feature points. The points are triangulated to form a pseudo-geometry. Then, we set a depth for each feature point by specifying correspondences between different views of the object. This step is shown in Figure 8.1, where a vase of flowers is being drawn on top of a table:

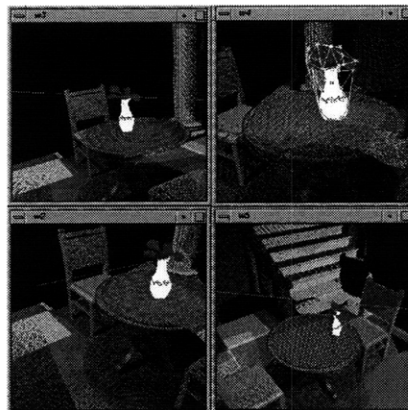


FIGURE 8.1 Specifying depth in a sketching tool.

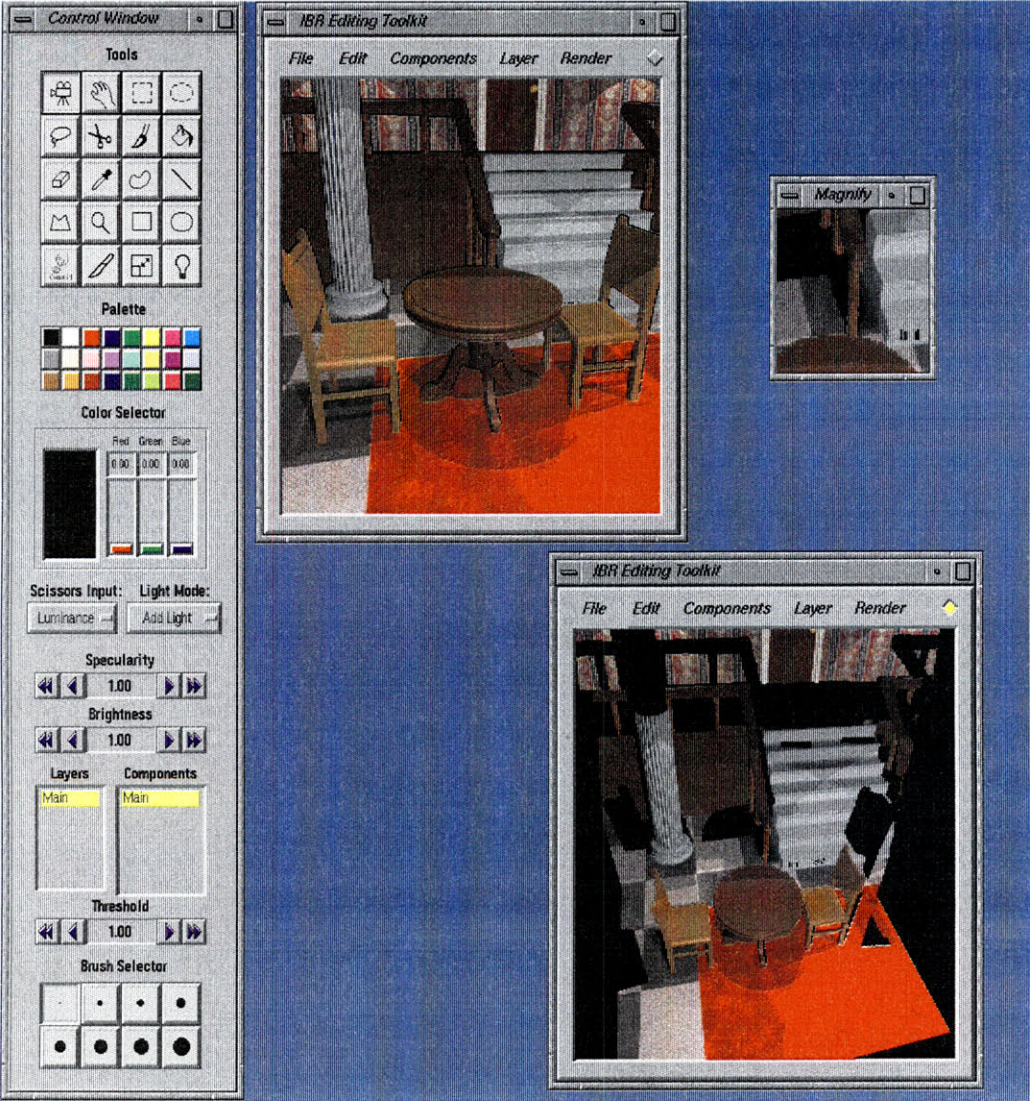
When we are done sketching, the program automatically outputs an image and its associated depth/camera file. The output is in the file format of our system. Thus, the new object can be loaded into our toolkit for editing. One could envision a sketching tool that was integrated into our toolkit, so that the user could access other editing operations while creating new objects.

8.4 Our System's Contributions

One of the strong points of this toolkit is its extensible nature. Our system was not designed as a comprehensive set of tools, but as a framework for performing editing operations. To this end, we have provided a set of tools that demonstrate the usefulness of image-based editing. Moreover, we have developed a structure for modifying the contents of a scene. Future work will build on these accomplishments, and add more capabilities for manipulating image-based environments.

In our introductory chapter, we argued that IBR systems are flexible. Because they do not rely on geometry, such systems remain highly interactive even when handling very complex scenes. We also discussed how IBR systems can produce photorealistic results, since they can make use real world data. In our editing toolkit, we have taken advantage of this flexibility and photorealism to provide the user with a means to control image-based scenes. Ultimately, it is this control over imagery that makes computer graphics useful to us.

APPENDIX A *Color Plates*



Color Plate 1: The interface.



a)



b)

Color Plate 2: Adding specularities. a) The original scene. b) The scene after painting with light.



a)



b)

Color Plate 3: Compositing. a) A teapot is imported and placed on the table. b) A shadow is added and the teapot is re-illuminated to match the lighting conditions in the scene.



a)



b)



c)



d)

Color Plate 4: Cutting, Compositing, & Importing. a) A gargoyle statue. b) The head of the gargoyle is cut from the scene. c) A child's head is imported and placed on top of the gargoyle's body. d) The modified statue is saved and imported into an office scene.

Bibliography

- [Adelson91] Adelson, E.H. and Bergen, J.R., "The Plenoptic Function and the Elements of Early Vision," in *Computational Models of Visual Proceedings*, M. Landy and J.A. Movshon, eds., MIT Press, Cambridge, 1991.
- [Adobe97] *Adobe Photoshop Version 4.0 User's Manual*, Adobe Press, 1995.
- [Alias96] *Alias|Wavefront Studio Version 7.5 User's Manual*, Alias|Wavefront, 1996.
- [Chen93] Chen, S.E. and Williams, L., "View Interpolation for Image Synthesis," Proc. SIGGRAPH '93. In *Computer Graphics Proceedings, Annual Conference Series*, 1993, ACM SIGGRAPH, pp. 279-288.
- [Chen95] Chen, S.E., "QuickTime VR- An Image-Based Approach to Virtual Environment Navigation," Proc. SIGGRAPH '95. In *Computer Graphics Proceedings, Annual Conference Series*, 1995, ACM SIGGRAPH, pp. 29-38.
- [Hanrahan90] Hanrahan, P. and Haeberli, P., "Direct WYSIWYG Painting and Texturing on 3D Shapes," Proc. SIGGRAPH '90. In *Computer Graphics Proceedings, Annual Conference Series*, 1990, ACM SIGGRAPH, pp. 215-223.
- [Gortler96] Gortler, S.J., Grzeszczuk, R., Szeliski, R., and Cohen, M., "The Lumigraph," Proc. SIGGRAPH '96. In *Computer Graphics Proceedings, Annual Conference Series*, 1996, ACM SIGGRAPH, pp. 43-54.
- [Kolb89] Kolb, C.E., *Rayshade Version 3.0 User's Guide*, 1989.
- [Levoy96] Levoy, M. and Hanrahan, P., "Light Field Rendering," Proc. SIGGRAPH '96. In *Computer Graphics Proceedings, Annual Conference Series*, 1996, ACM SIGGRAPH, pp. 31-42.

- [Lippman80] Lippman, A., "Movie-Maps: An Application of Optical Videodiscs to Computer Graphics," Proc. SIGGRAPH '81. In *Computer Graphics Proceedings*, Annual Conference Series, 1981, ACM SIGGRAPH, pp. 32-43.
- [Mortensen95] Mortensen, E.N. and Barrett, W.A., "Intelligent Scissor for Image Composition," Proc. SIGGRAPH '95. In *Computer Graphics Proceedings*, Annual Conference Series, 1995, ACM SIGGRAPH, pp. 191-198.
- [McMillan95a] McMillan, L. and Bishop, G., "Plenoptic Modeling: An Image-Based Rendering System," Proc. SIGGRAPH '95. In *Computer Graphics Proceedings*, Annual Conference Series, 1995, ACM SIGGRAPH, pp. 39-46.
- [McMillan95b] McMillan, L. and Bishop, G., "Shape as a Perturbation of Projective Mapping," *UNC Technical Report TR95-046*, University of North Carolina, 1995.
- [Schoeneman93] Schoeneman, C., Dorsey, J., Smits, B., Arvo, J., and Greenberg, D., "Painting with Light," Proc. SIGGRAPH '93. In *Computer Graphics Proceedings*, Annual Conference Series, 1993, ACM SIGGRAPH, pp. 47-51.
- [Seitz96] Seitz, S.M. and Dyer C.R., "View Morphing," Proc. SIGGRAPH '96. In *Computer Graphics Proceedings*, Annual Conference Series, 1996, ACM SIGGRAPH, pp. 21-30.
- [Seitz98] Seitz, S.M. and Kutulakos, K.N., "Plenoptic Image Editing," *Proc. Sixth International Conference on Computer Vision*, 1998, pp. 17-24.
- [Woo97] Woo, M., Neider, J., and Davis, T., *OpenGL Programming Guide: The Official Guide to Learning OpenGL*, Addison-Wesley, Reading, 1997.
- [Zakai96] Zakai, Y. and Rappoport, A., "Three-Dimensional Modeling and Effects on Still Images," *Proc. Eurographics Rendering Workshop*, 1996, pp. C3-C10.