Report Number: WUCSE-2006-46

2006-01-01

A view-based deformation tool-kit, Master's Thesis, August 2006

Nisha Sudarsanam

Camera manipulation is a hard problem since a graphics camera is defined by specifying 11 independent parameters. Manipulating such a high-dimensional space to accomplish specific tasks is difficult and requires a certain amount of expertise. We present an intuitive interface that allows novice users to perform camera operations in terms of the change they want see in the image. In addition to developing a natural means for camera interaction, our system also includes a novel interface for viewing and organizing previously saved views. When exploring complex 3D data-sets a single view is not sufficient. Instead, a composite view built... **Read complete abstract on page 2.**

A view-based deformation tool-kit, Master's Thesis, August 2006

Nisha Sudarsanam

Complete Abstract:

Camera manipulation is a hard problem since a graphics camera is defined by specifying 11 independent parameters. Manipulating such a high-dimensional space to accomplish specific tasks is difficult and requires a certain amount of expertise. We present an intuitive interface that allows novice users to perform camera operations in terms of the change they want see in the image. In addition to developing a natural means for camera interaction, our system also includes a novel interface for viewing and organizing previously saved views. When exploring complex 3D data-sets a single view is not sufficient. Instead, a composite view built from multiple views may be more useful. While changing a single camera is hard enough, manipulating several cameras in a single scene is still harder. In this thesis, we also present a framework for creating composite views and an interface that allows users to manipulate such views in real-time.

2006-46

A view-based deformation tool-kit, Master's Thesis, August 2006

Authors: Nisha Sudarsanam

Corresponding Author: nisha.sudarsanam@gmail.com

Web Page: http://www.cs.wustl.edu/~nsudarsa

Abstract: Camera manipulation is a hard problem since a graphics camera is defined by specifying 11 independent parameters. Manipulating such a high-dimensional space to accomplish specific tasks is difficult and requires a certain amount of expertise. We present an intuitive interface that allows novice users to perform camera operations
in terms of the change they want see in the image. In addition to developing a natural means for camera interaction, our system also includes a novel interface for viewing and organizing previously saved views. When exploring complex 3D data-sets a single
view is not sufficient. Instead, a composite view built from multiple views may be more useful. While changing a single camera is hard enough, manipulating several cameras in a single scene is still harder. In this thesis, we also present a framework for creating composite views and an interface that allows users to manipulate such views in real-time.

Type of Report: Other

WASHINGTON UNIVERSITY

THE HENRY EDWIN SEVER GRADUATE SCHOOL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

A THESIS ON A VIEW-BASED DEFORMATION TOOL-KIT

by

Nisha Sudarsanam , Bachelor Of Engineering, Computer Engineering

Prepared under the direction of Professor Cindy M Grimm

A thesis presented to the Henry Edwin Sever Graduate School of
Washington University in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

August 2006

Saint Louis, Missouri

WASHINGTON UNIVERSITY
THE HENRY EDWIN SEVER GRADUATE SCHOOL
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

A THESIS ON A VIEW-BASED DEFORMATION TOOL-KIT

by

Nisha Sudarsanam

ADVISOR: Professor Cindy M Grimm

August 2006

Saint Louis, Missouri

Camera manipulation is a hard problem since a graphics camera is defined by specifying 11 independent parameters. Manipulating such a high-dimensional space to accomplish specific tasks is difficult and requires a certain amount of expertise. We present an intuitive interface that allows novice users to perform camera operations in terms of the change they want see in the image. In addition to developing a natural means for camera interaction, our system also includes a novel interface for viewing and organizing previously saved views. When exploring complex 3D data-sets a single view is not sufficient. Instead, a composite view built from multiple views may be more useful. While changing a single camera is hard enough, manipulating several cameras in a single scene is still harder. In this thesis, we also present a framework for creating composite views and an interface that allows users to manipulate such views in real-time.

To my family and friends

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

These days, both virtual 3D environments and data-sets are becoming increasingly complex. As a user you want to explore such scenes in the simplest manner. In graphics, exploring scenes is accomplished by using a virtual camera, analogous to the real world. However, unlike a real world camera, the graphics camera is manipulated in the same 2D environment that the 3D scene is rendered in. Since the graphics camera is specified using 11 parameters, manipulating it is hard. For novice users, changing the camera in order to acquire a desired view can be very difficult. Current techniques prevalent for manipulating cameras are also quite unintuitive. A few of these techniques are menu-based, thus resulting in a myriad of menu options corresponding to the different possible camera operations. Others are mouse-based, which require mapping 11 degrees of freedom (dof) to a 2 dof device. These therefore use shortcut keys and key modifiers in order to disambiguate different operations. Developing an intuitive camera manipulation interface is thus an active area of research.

Often data-sets are so complex that a single view does not suffice to adequately examine the different features of the data-set. On the other hand, if multiple windows are used for different views, it can be difficult to mentally stitch together a composite view of the whole data-set from these views. One common method of combining different features into a single view is to deform the data-set. The problem with such techniques is that the deformation is dependent on the current view of the data-set: changing the view even slightly produces an inconsistent deformation. The methods

we present in this thesis combine different views in order to generate a seamless composite view of a data-set. Our methods leave the underlying geometry of the model intact. The idea of using view-based deformations for exploring data-sets is an new idea in graphics. Finally, we use view-based deformations to re-create a special kind of perspective effect commonly used by artists called curved perspective.

1.2 Goals

Our work has been directed towards accomplishing two key goals.

- To develop and evaluate a camera manipulation interface that moves away from menu-based techniques.
  - The interface should be based more on how the user wants to view a 3D scene rather than relying on the user's familiarity with the internal parameters of the camera.
- To create a framework that allows users to easily tie together multiple views of a data-set to form a composite view.
  - To develop an effective real-time interface that allows users to manipulate this composite view.

1.3 Outline

Chapter 2 introduces key concepts and conventions that will be used in the rest of the thesis. Chapter 3 discusses the workings of our camera manipulation interface. Chapter 4 provides a general introduction to non-linear perspective projections while Chapter 5 explains the non-linear perspective framework and the interface for visualizing data-sets . Chapters 6 and 7 deal with two other applications of our framework, namely generating images that have a curved perspective and creating panoramic views. Chapter 8 discusses implementation of the framework on the GPU while Chapter 9 summarizes with the conclusion.

Chapter 2

Key Concepts

In the following chapter I define terms that are used throughout this thesis.

2.1 Models

The 3D models used in this thesis were either obtained from 3D laser scans of real-world objects or created using 3D modeling software. The coordinates of all the vertices in the model, along with how they are connected, are stored in a text file. Generally, triangles are used to represent the connectivity of the vertices. The coordinates of the vertices are based on a canonical coordinate frame, which is located at the origin and follows the right-handed coordinate system. This coordinate-space is known as **world-space**. When the connectivity of the vertices on the model is known, the model is termed as a **mesh**. Other types of data-sets include point-sets (only the 3D positions of the points are known) and volume-data (slices through an object are stored as images, used mainly for representing medical data). In addition, to storing the position of points, normals are associated with the vertices for lighting calculations (Section 2.4).

2.2 Transformation matrices

Common transformations used are the translation, rotation and scaling of points and vectors. These transformations are represented as 4×4 matrices assuming that points

are represented using homogenous coordinates i.e (X, Y, Z, W). Thus, transforming points (or vectors) boils down to performing matrix multiplications.

2.2.1 Rotation

Rotation around an axis (r_x, r_y, r_z) by an angle θ (axis-angle representation) can be denoted by a matrix. However, in this thesis, I have chosen to represent rotation through quaternions. Conversions between matrices and quaternions is an easy and common operation.

Quaternions have 4 dimensions, one real and 3 imaginary dimensions (i, j, k). Each of the imaginary dimensions has a unit value of $\sqrt{-1}$ (but are different square roots of $\sqrt{-1}$) and are mutually perpendicular to each other (Equation 2.1).

$$q = w + xi + yj + zk \tag{2.1}$$

Quaternions provide convenient methods for representing and combining 3D rotations. For example, when a quaternion is used to represent a 3D rotation, w corresponds to the angle of rotation while x, y, z corresponds to the axis about which rotation takes place. Thus a quaternion q, defined using this representation is given by Equation 2.2.

$$q = cos(\theta/2) + (r_x sin(\theta/2))i + (r_y sin(\theta/2))j + (r_z sin(\theta/2))k \tag{2.2}$$

A unit quaternion is a quaternion of unit length and can be considered as a point on a unit hyper-sphere S^3. A body starting at orientation q_1 and ending in an orientation of q_2 will rotate uniformly with a constant angular velocity $v = log(q_1^{-1} q_2)$. The shortest path on S^3 between q_1 and q_2 is called a geodesic on S^3. The geodesic norm is given by Equation 2.3.

$$dist(q_1, q_2) = \|log(q_1, q_2)\| \tag{2.3}$$

We use this metric when measuring the difference between camera rotations for organizing our camera bookmarks (Section 3.10).

Finally, a point, $P(X, Y, Z)$ can undergo translation, rotation and scaling simultaneously. To do this a composition of translation, rotation and scaling matrices is built

to form a single matrix, M (Equation 2.4).

$$M = TRS$$
$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = M \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{2.4}$$

2.2.2 Transforming vectors

When P is multiplied by M, it is said to be **transformed**. A vector, $\vec{N}$ is transformed in a slightly different way (Equation 2.5).

$$\overline{n} = (M^{-1})^T \overline{N} \tag{2.5}$$

2.2.3 Convention

In this thesis, transformed quantities are represented in lower case. Untransformed quantities are represented in uppercase, with matrices represented in a similar manner. Vectors are represented with a line on top.

2.3 Camera Model

This thesis is devoted to manipulating a graphics camera in a 3D scene and hence understanding the mathematical model behind it is important. The graphics camera is an extension of a simple pin-hole camera. 11 parameters are used to represent the camera, which can be divided into external and internal camera parameters.

- Position & Orientation - 6 parameters (External)
- Zoom - 1 parameter (Internal)
- Center of projection - 2 parameters (Internal)

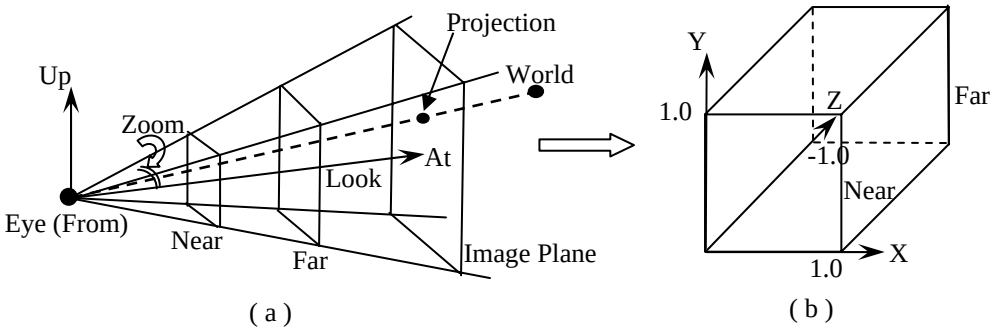


Figure 2.1: (a) An arbitrary view-volume located in world-space (b) Canonical view volume

- Skew - 1 parameter (Internal)

Position (also known as eye of the camera) controls the location of the camera in world-space while the orientation specifies the angles made by the camera with world-space coordinate axes. The additional parameters are the camera-zoom, which corresponds to a uniform-scale applied to the world, and the center-of-projection, which is the center of the image-plane. Usually set to $(0, 0)$, the center-of-projection of real-world cameras is non-zero which contributes to a slight shift in the image. Skew is the shear of the image-plane and is usually zero.

Figure 2.1(a) shows the visible region of the world in world-space, known as the view-volume. The *near* and far planes represent the field-of-interest i.e anything that is projected to lie before the *near* plane or beyond the far plane cannot be seen. This process of rejecting points based on their position with respect to these planes is known as clipping. Performing clipping in an arbitrary view-volume is hard, so the view-frustrum is transformed to the canonical view-volume (Figure 2.1(b)). Transforming arbitrary 3D points to the canonical camera-space is done by multiplying with a single matrix, C (Equation 2.6). C is a composition of translation, rotation and scaling matrices similar to those seen in Section 2.2.

$$C = PSRT \tag{2.6}$$

P : Perspective matrix built from the center-of-projection and skew
S : Scale matrix built from zoom
R : Rotation matrix built from the orientation of the camera
T : Translation matrix built from the eye.

Finally, in this thesis I additionally define the camera using the `From-At-Up` model. `From` correspond to the camera eye, `At` corresponds to the point the camera is looking at and `Up` corresponds to the vertical axis of the camera. At several points in the thesis, when I refer to the camera's `look vector`, I am referring to the axis along which the camera is looking down ($At - From$).

2.3.1 Camera operations

Camera operations refer to the process of changing one or more of the camera parameters mentioned in the previous section. A pan corresponds to translating the camera in a direction parallel to the image-plane, a zoom corresponds to increasing or decreasing the zoom parameter, and a dolly corresponds to moving the camera up or down the look vector.

2.4 Lighting

Lighting refers to the calculations that are performed in order to determine the final color of a point. There are several types of lights that are available such as spot-lights and directional lights. The important point to note about lighting calculations are that all the light parameters, including light-direction and light-position, are stored in camera-space. Thus, changing the camera changes the lighting effects found in a scene. Also, lighting is based on the geometry of the object, namely the normal at a given vertex, thus changing the geometry also affects lighting.

2.5 Deformations

Transforming a vertex in a scene is also called deforming the vertex. A vertex can be deformed in two ways. Its world-space position can be changed in which case such a deformation is referred to as an object-space deformation. A vertex can also be deformed using the camera, since all points are finally multiplied by the camera matrix, C. This type of deformation is known as a view-based deformation. The important point to note is that applying a view-based deformation does not change the world-space position of a point, it only changes the way in which the point is projected.

2.6 Widgets

This thesis discusses interfaces that are based on widgets. A widget is an interface module that the user can interact with. Screen-space widgets are widgets that are defined in 2D (defined in the image-plane) as opposed to 3D widgets (which are defined in world-space).

Chapter 3

CubeCam : A screen-space camera manipulation tool

3.1 Motivation

Virtual 3D environments are becoming larger and increasingly realistic. Manipulating a virtual camera in such 3D environments is a crucial concern. Unfortunately, effective camera manipulation techniques have been difficult to develop. In orderto completely specify a camera, 11 distinct degrees of freedom need to be specified : six for positioning and orienting the camera and five for controlling the projection of the image. Standard mouse-based techniques can control only a subset of this large 11-dimensional space. In order to map the whole camera space to the 2D space of the mouse, current techniques use an array of menu options, shortcut keys and key modifiers. This mapping is not always transparent to the user. Given the myriad of tools available to manipulate the camera, it is hard for the user to know which tool to use in order to obtain their desired view.

Another problem with current 3D applications is the method used to represent "camera bookmarks", or previously saved camera views. Current techniques represent camera bookmarks as a textual list that users can organize. However, a textual representation of a bookmark conveys no visual information about the view point corresponding to that bookmark. There is also no way to find bookmarks close to the current viewpoint.

3.2 Goals

Our basic goal is to create a camera manipulation interface that supports intuitive selection of the various camera operations. One way to do this is to specify a camera operation in terms of the desired change in the projected image. Additionally, we would like to visualize camera bookmarks and the relationship between them.

3.3 Background

In linear perspective, the perspective effects depend on the relative position of the camera with respect to the 3D scene. Artists explicitly visualize this relationship using geometric proxies which are used to reduce objects to sets of points, lines and curves. Thus, geometric proxies visually reflect the artists perception of the 3D scene. We borrow this idea by presenting the user with camera primitives which not only serve as geometric proxies for visualizing the current camera but also can modify the projection of the scene. These camera primitives allow camera operations to be defined with respect to any point in the scene and also provide visual feedback with regard to how specific camera operations affect the projected image.

3.4 The CubeCam Interface

The camera is visualized through the projection of a cube in the scene. The cube can be viewed in three different perspective views, namely 1-pt perspective, 2-pt perspective and 3-pt perspective (Figure 3.1). We collectively label the three perspective views of the cube as *CubeCam*. Each of these perspective views form a camera primitive that encapsulates a subset of all the possible camera operations. This subset is selected based on the operations that map naturally to the shape of the cube in that particular view. This allows camera operations to be defined in terms of the change the user wants to see in the camera primitive. For example, when the cube is in 1-pt perspective making the the cube bigger results in the camera being zoomed in or out. This is analogous to saying "perform the camera operation that makes the cube

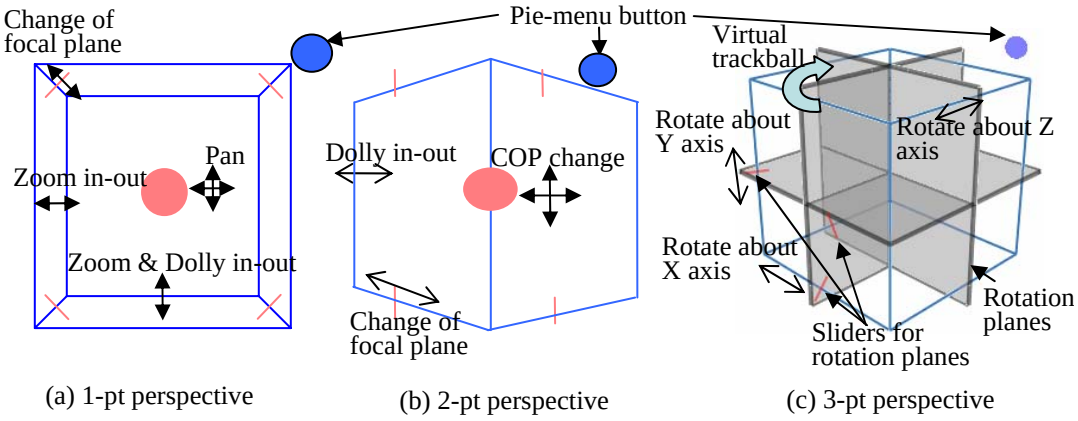


Figure 3.1: Figure shows each camera primitive and the associated camera operations

appear bigger". In this case, the camera operation is a camera zoom. Additionally, we employ pie-menus for displaying the different non-camera actions.

3.5 Previous work

For mouse-based systems, camera control paradigms fall roughly into two categories, camera-centric and object-centric. In the camera-centric paradigm, operations are applied to the camera as if it were a real object in the scene. This mirrors camera placement in the real world, and many of the camera operations (dolly, pan, and roll) reflect that. The external parameters, position and orientation, can be specified either "through the lens", or by manipulating a pictorial representation of the camera in a second window. The internal camera parameters, with the exception of focal length, are changed through textual input.

In the object-centric paradigm, the camera is centered on an object and the viewpoint is rotated relative to the object (as if there were a virtual trackball around the object [12], [11]). The camera can also be zoomed in and out. This paradigm is useful when there is a single object in the scene (or one object of importance) and the user is simply choosing a direction from which to view it.

An alternative approach to directly specifying the camera is to use image-space constraints [3], [8]. In this approach, points in the scene are constrained to appear at

particular locations, or to move in a specified direction, and the system solves for the camera parameters that meet those constraints.

The recently-introduced IBar [19] is, in some sense, a specialization of the constraint approach, where the points are the points of the edge of a cube. Like CubeCam, the IBar is a screen-space widget where changing the widget changes one or two camera parameters. The IBar and CubeCam have similar goals; both systems move beyond current menu-based camera manipulation techniques to a unified screen-space camera primitive. The underlying mathematical framework of the two systems are similar. Thus, both systems support the same set of camera operations. However, CubeCam improves and extends the interface presented to the user. One of the key differences is manner in which CubeCam encapsulates camera operations. CubeCam splits up all the camera operations between three different primitives unlike the IBar, which has only one camera primitive for all of the camera operations. Ghosting, visualization of the focal plane and the rotation planes are features that are unique to CubeCam [20] introduces navigation of large collections of photos. This work and our bookmarks have different goals. The goal of this work was to help users organize photographs and allow navigation of the photos by mapping the photos onto the object which was photographed. Our goal is to help users navigate the camera using the bookmarks as a guide.

3.6 1-pt perspective

1-pt perspective allows for camera zooming, camera panning and perspective manipulation.

3.6.1 User's view

Figure 3.1(a) shows the camera operations associated with the 1-pt perspective view. The camera is zoomed in or out by dragging the outer edges of the cube outward or inward. Perspective distortion (3.6.2) is achieved by dragging the inner edges. The camera is panned by moving the whole cube. The focal plane is slid through the scene

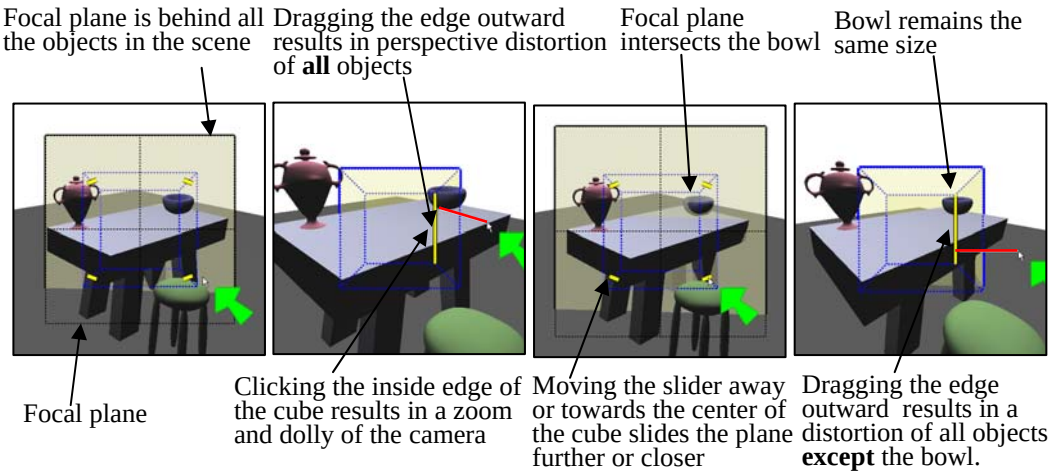


Figure 3.2: Perspective distortion controlled by the focal-plane

using the sliders on the receding edges of the cube. The pie-menu button is used to bring up the pie-menu.

3.6.2 Perspective Distortion

Perspective distortion is a function of the distance of the camera eye point to the object in question. Unfortunately, changing the camera distance also changes the size of the object in the scene. To counter-act this, the camera is zoomed out simultaneously to keep the object the same size [19]. Thus, objects at a specific distance d (focus-distance) along the look vector remain the same size in the image plane. This distance is visualized by a plane drawn at depth d (focal-plane) , which can be changed by moving a slider along the receding edges of the cube in the 1-pt perspective view (Figure 3.2).

3.7 2-pt perspective

2-pt perspective allows for camera dollying and center of projection change

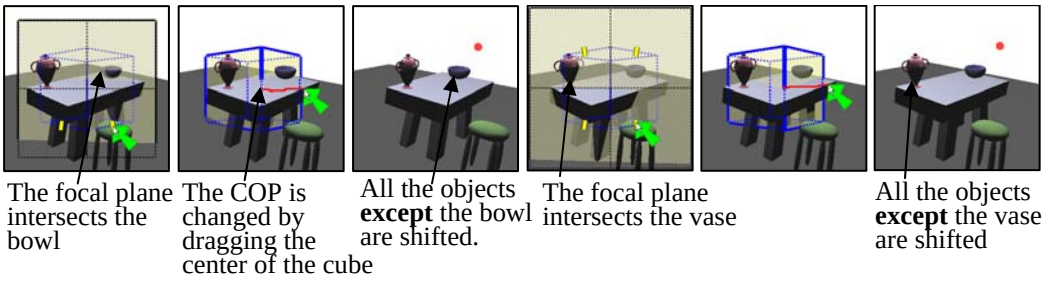


Figure 3.3: Center of projection and camera pan

3.7.1 User's view

Figure 3.1(b) shows the camera operations associated with the 2-pt perspective view. The camera is dollied in or out by dragging the outer edges of the cube outward or inward. The camera's center of projection is changed by moving the whole cube. As in the 1-pt perspective, the sliders on the cube are used to change the position of the focal plane in the scene.

3.7.2 Center-of-projection

We extend the focal plane idea to the 2-point perspective view. When the center of projection is changed the whole scene slides in the opposite direction. To prevent this, the camera is panned in the opposite direction ensuring that objects at a depth d remain stationary. A focal plane is rendered at this depth and its position is controlled by sliders located on the slanting edges of the cube. The position of the focal plane controls which objects stay fixed on the screen (Figure 3.3).

3.8 3-pt perspective

3-pt perspective is used only for either constrained rotation or virtual trackball of the camera

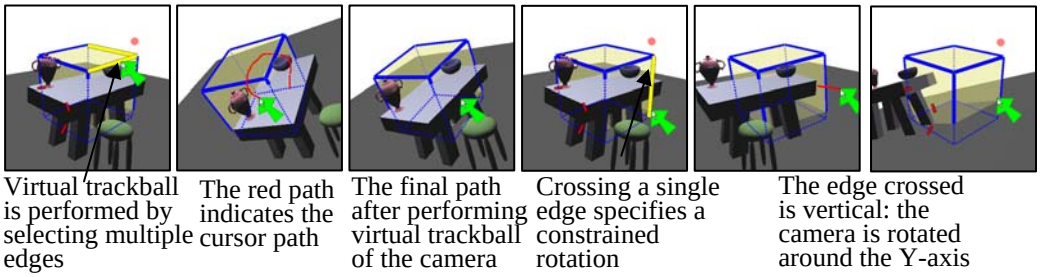


Figure 3.4: Virtual trackball and constrained rotation

3.8.1 User's view

Figure 3.1(c) shows the camera operations associated with the 3-pt perspective view. The semi-transparent planes are rotation planes. The intersection of these planes form the rotation point about which the camera can be rotated. The position of each rotation plane is controlled using the corresponding slider located on an edge of the cube. Constrained rotation about a particular axis is selected by crossing the appropriate edge. Clicking and dragging any point on the cube then performs the actual rotation. A virtual trackball rotation of the camera is selected by crossing multiple edges of the cube.

3.8.2 Rotation

In the 3-pt perspective view, the user can perform either a constrained rotation or virtual trackball of the camera. A constrained rotation is specified by crossing a single cube edge. In constrained rotation, the camera is rotated about a single axis. The rotation axis depends on whether the primitive is centered in the scene or on an object. If centered in the scene, the camera is rotated about one of the X,Y,Z world axes, depending on the selected cube edge. For example, if the cube edge selected is vertical, the camera will be rotated about the Y axis. If centered on an object, the camera is rotated about one of the X,Y,Z axes of the object. A virtual trackball rotation is specified by crossing multiple multiple edges of the cube. Once the rotation type has been selected, the user can click and drag any point on the cube to perform the rotation (Figure 3.4).

3.8.3 Rotation Planes

Our system visualizes the rotation point about which a camera rotation takes place. This is done using three semi-transparent planes. The planes are aligned along the principal axes of the cube in the 3-pt perspective view (Figure 3.1(c)). Sliders, located on three edges of the cube, are used to change the position of the rotation planes. The final rotation point is the intersection of the rotation planes.

3.8.4 Scene-centric versus Object-centric CubeCam

CubeCam can be either centered in the scene or at the center of an object in the scene. Users can switch between the two configurations using the pie-menu. Clicking on an object centers CubeCam about that object. All of the previously described camera operations can be performed in both configurations of CubeCam with the only difference being the axis used for rotating the camera in the 3-pt perspective. If CubeCam is scene-centered then the camera primitives snap back to the original position after the user has modified them. In the case of object-centric CubeCam, the position of the camera primitives change with the position of the object.

3.9 Ghosting

Our goal is to present the user with a simple camera manipulation interface. By creating multiple camera primitives we simplify the overall interface but introduce the problem of expecting the user to remember the functions associated with each primitive. In order to help the user learn these functions we provide ghosting. When ghosting is active, users manipulate the camera primitive as they would normally. However, as they change the camera primitive, a "ghost camera" is changed. This ghost camera is used to render a ghosted scene rendered over the current scene. The original scene camera remains unaffected (Figure 3.5). For advanced users, this mode provides an opportunity to test out possible camera changes before actually making them.

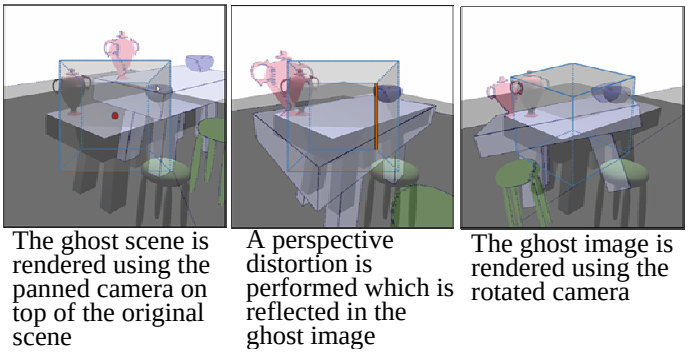


Figure 3.5: Ghosting helps users learn functions of the camera primitives

3.10 Camera Bookmarks

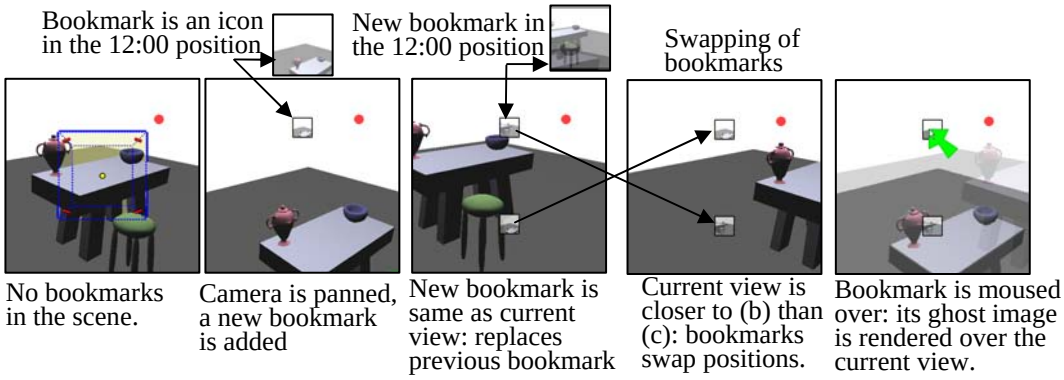


Figure 3.6: Bookmarks ordered based on translation parameters

It is very useful to be able to save cameras and snap back to them at will. For example, when modeling a surface, a user might bookmark a handful of orthogonal views and close-ups of complex geometry. An animator might also use bookmarks to start laying out an animation sequence. In both of these cases, we need to provide the user with a method for quickly searching through existing cameras. Although the user could simply create a text list, appropriately naming each camera, we believe that a visual search mechanism is more useful. We have implemented two bookmark placement algorithms, one of which is static placement and the other is automatic placement. Each bookmark is represented as an icon with an image of the scene. In static placement, the user simply places the icon on the image plane. As the user changes the current camera, the bookmark icons remain at the same place on the screen. Mousing over an icon renders a ghost image of the bookmarked view on top

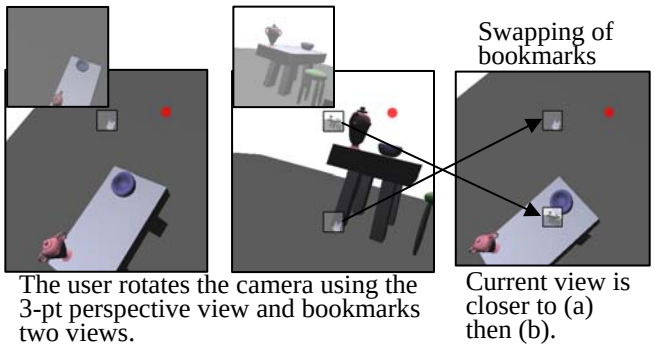


Figure 3.7: Bookmarks ordered based on rotation parameters

of the current view. Double clicking an icon switches to the bookmark.
In the automatic method, bookmark icons are arranged in a circular fashion around the center of the screen. The cameras are arranged in an increasing order of distance from the current camera in a clockwise manner. If the 1-point or 2-point perspective view is active, the ordering is based on the translation parameters, otherwise, it is based on the rotation parameters. To order cameras based on the translation parameters, each bookmarked camera's eye point is projected onto the film plane of the current camera. The magnitude of the vector from the origin to the projected eye point is used to order the bookmarks (Figure 3.6).

The rotation distance is calculated by measuring the length of the geodesic path between the quaternion of the current camera and the quaternion of the bookmarked camera (Figure 3.7).

$$GeodesicPath(q_1, q_2) = Log({q_1}^{-1}, q_2) \tag{3.1}$$

q_1 : Quaternion of the current scene camera.
q_2 : Quaternion of the bookmarked camera.

As the user changes the current scene camera, bookmarked cameras move further or closer to the current camera. Accordingly, the ordering of the bookmark icons change.

3.11 Implementation

3.11.1 Camera primitives

If scene-centric CubeCam is active, a cube of unit scale is placed at the origin. Since scene-centric CubeCam snaps back after the camera has been changed, inverse transformations have to be applied to the cube to reset it. The cube is translated back to the point which the camera is looking at, translated appropriately down the **look** vector so that it appears the same size. An inverse rotation is applied and finally a scaling proportional to the camera zoom is applied. In case of object-centric CubeCam, the cube is transformed as the object it is attached to. Finally, the cube is translated down the $\vec{look}$ so that it appears at the right size on the screen.

3.11.2 Rendering a ghost scene

The current scene is rendered to the back-buffer using the ghost camera. If ghosting is turned on, the ghost camera is the camera created by modifying the current camera. If bookmarks are turned on, the ghost camera is the bookmarked camera. The scene is rendered under the original lighting in a non-photorealistic style with the silhouette edges highlighted. The contents of the back-buffer are copied into a texture which is alpha-blended with the original scene, rendered using the current camera.

3.11.3 Bookmark Icons

As the number of bookmarks in the scene increases, the circle of bookmarks created by the automatic placement algorithm becomes more crowded. To prevent the occlusion of bookmarks in the circle, we vary the size of the icon, r, as a function of the number of bookmarks in the scene, n.

$$SizeOfIcon = \sqrt{2} \times r \times \frac{2\pi}{3n} \tag{3.2}$$

Chapter 4

Non-linear perspective projections

The previous chapter in the thesis dealt with building an interface for manipulating a single camera. In the chapters to follow I will discuss building interfaces for manipulating multiple cameras in a scene, as well as a framework for integrating multiple cameras.

4.1 Background

Non-linear perspective forms a part of non-photorealistic rendering, a set of techniques inspired by the work of artists. Even though linear perspective is a good approximation of the human visual system, artists such as Escher, Picasso and Hockney intentionally incorporated distortions of perspective in their work (Figure 4.1). By simply changing the perspective over the scene, these artists could introduce subtle mood changes and vary the relative importance of objects in the scene.
A scene viewed through a non-linear perspective can be thought of as being seen through a linear perspective in certain local regions while overall having a smooth perspective change. In other words, even though in some regions there are certain local perspective changes, global scene coherence is maintained. This is the reason why, even though we are aware of the perspective distortion in the examples illustrated, we are not overly disturbed by it.

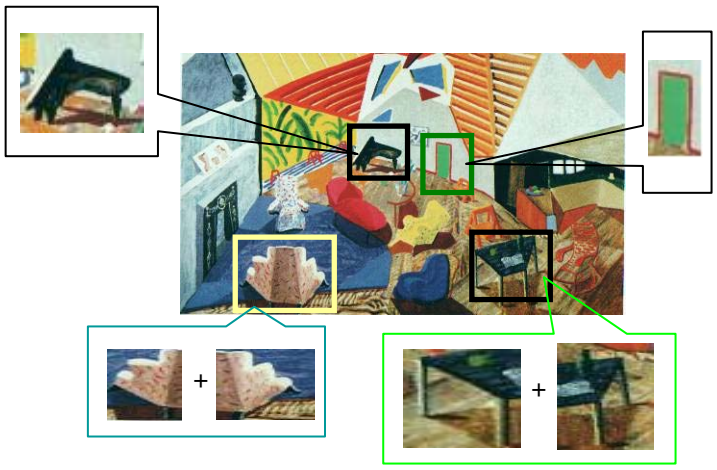


Figure 4.1: Above is an example of non-linear perspective. Several linear perspective pieces are combined together to form a global view. Perspective varies over even individual objects such as the side-table and chair.

4.2 Non-linear perspective rendering framework

Our goal is to create and manipulate non-linear perspective projections in real-time through simple and intuitive interfaces. As mentioned in the previous section, a single non-linear perspective view can be thought of as collection of linear perspective views tied together in a coherent manner [18]. Towards this end, our framework assigns a unique camera to each vertex of a mesh (or point in case of a point-set). Several vertices within a particular region may be assigned the same camera. This corresponds to the local perspective seen in a non-linear perspective image. A global camera is used for viewing the rest of the scene. Blending is done between the local perspective view and the global view to ensure a smooth transition between the two. This ensures the global scene coherence.

In the next few chapters we will introduce our non-linear rendering framework in more detail and illustrate its applicability in three main areas namely, exploring complex data-sets, creating panoramic views and creating interesting artistic effects. Although the interface is adapted to suit each particular application, the underlying framework remains the same for all of the applications.

4.3 Contributions

I have developed a framework that helps users create and manipulate non-linear perspective projections. Specifically, I have created an interface that is can be used for creating non-linear projections of complex data-sets and simulating interesting artistic effects. This system guarantees seamless blending of multiple camera views and real-time manipulation of the final composite view. In order to sucessfully implement the system I had to implement the framework on the GPU, perform adaptive subdivision of meshes and on the front-end develop a set of intuitive tools which would be general enough and yet useful for different visualization applications.

Chapter 5

Visualization of complex data-sets

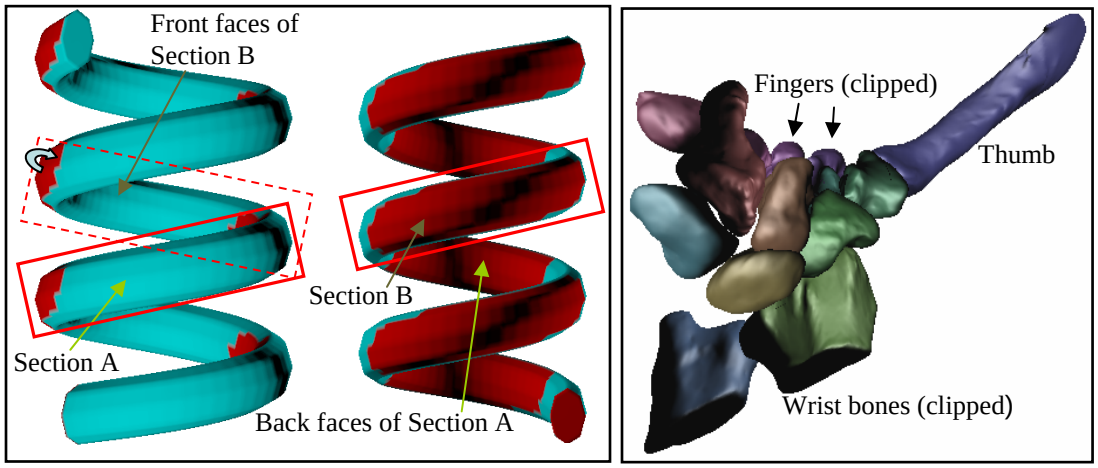
5.1 Motivation

In general, data-sets can either be composed of inherently complex geometry or a large collection of relatively simple geometry (Figure 5.1). Finding the "best" view in which all of the interesting features can be seen distinctly is challenging. As certain features come into a view, others move out. Also, it can be quite difficult to manipulate a virtual camera to a view a particular feature. Knowing which of the many camera parameters to change in order to get a desired view of the scene can be quite an art. Thus, instead of trying to find a single view in which all of the features are visible, a better approach is to view the dataset using a composite view. This composite view is made up of subviews, each of which allows particular features to be seen clearly.
The idea of combining multiple views into a single one is not new. This concept has been illustrated in the works of artists such as M.C. Escher, David Hockney and Picasso. However, the novel contribution lies in applying and adapting these concepts to the domain of data visualization and exploration.

5.2 Goals

We have three key goals.

- To visualize complex data-sets through non-linear projections.



(a) Front and back faces of a helix

(b) Bones in a hand

Figure 5.1: (a) Figure shows the front (left-side) and back (right-side) faces of a helix. Trying to examine section A and section B simultaneously in a single camera view is impossible. The unwrap widget rotates the front faces of section B (shown in dashed lines) to show its back faces while leaving rest of the helix to be seen as is in the front-view (Figure 5.3). (b) Figure shows several overlapping bones lying at different depths along any given viewing direction. Different parts of the bone are seen in different views. The clipping widget exposes sections of the bone lying at different depths without deforming the model.

- Create a tool-kit of widgets for modifying such projections. These widgets must function in real-time.
  - The widgets must be well-defined. More specifically, the functionality of a widget must be clearly defined to the user.
  - When used in isolation, a widget should produce a "simple" projection but it should be possible to combine multiple widgets to produce more complex projections.
- Finally, the framework should be independent of the underlying representation of the data-set.

5.3 Previous Work

A large body of research has been devoted to interactive viewing of volume data-sets. However, these methods actually deform the underlying geometry of the data-set in order to expose interesting structures [5, 10, 15]. The problem with such methods is that the deformation depends on the current view of the model. Changing the view results in an inconsistent deformation. In general, view-based deformations work around this problem by applying deformations based on the current view of the model.

Lenses such as fish-eye lenses and magnification lenses [22, 14] have been used for visualization of large amounts of information given a restricted amount of screen real-estate. Similar to these papers, we introduce a transformation of the view-space for exploring data in this manner. However, lenses are a subset of the entire set of view-transformations possible using our framework.

General linear cameras introduced in [24] presented a variety of camera models. Our framework includes the cameras that were presented in [24]. The main advantage of our framework is intuitive interface that is presented for generating non-linear projections. The idea of breaking down a general non-linear projection into a set of local projections which can then be seperately manipulated by the user through our widgets is the key contribution of this work.

Work has been done in producing multi-perspective panoramas given either a 3D model and a camera path [23] or a set of images [17]. Our work is similar in that a non-linear perspective widget implicitly defines a camera path. However, our approach ensures that the surrounding context of the data-set is maintained while creating a panorama locally. The above methods are also not suited for interactive manipulation.

Multi-projection techniques [2, 9] allow each object to be rendered from a different view-point. Neither work allows for continuously varying projections over a single object. The idea of creating a non-linear perspective projection from multiple linear perspectives was first introduced in [18]. Singh's approach required users to specify individual camera parameters of different cameras in the scene. Our work ensures global scene coherence by allowing users to specify a default view with which we blend different view-points.

A different approach [7] defined a simple set of primitives such as points, lines, and bounding boxes that were used to express image-space constraints. A simplex solver

found the camera that satisfies these constraints. These primitives were then used to generate a set of cameras that form a non-linear projection. A greedy approach was employed to group features that are satisfied by the same camera. While it is useful to express constraints for a single camera through these primitives, it is sometimes hard to know which primitives to use in order to get a specific non-linear projection. The framework is not interactive and the solver occasionally gets stuck in local minima.

5.4 Approach

Each of our non-linear perspective widgets consists of a 3D volume and a 2D area in screen-space. We refer to the 3D volume as the "source volume" and the screen-space area as the "destination area". The source volume specifies the region of the data-set which the user is interested in viewing differently. The destination area controls the "after-projection" aspects of the region such as placement on the screen and projected size of the volume.

The user first specifies a default view, which controls the overall view (global perspective) of the data-set. Next, the user selects a source volume using our 3D widget. The widget controls the position, orientation and scale of the selected volume. To simplify the interface, we make certain assumptions about the destination area. Thus in general, the user only needs to specify a volume in order to generate a simple non-linear projection.

Once the volume is created, the system automatically creates and assigns different cameras to different parts of the data set, based on the data points location relative to the source volume. Exactly how these cameras are constructed from the default one depends on the particular widget being used. The net effect is that data inside of the source volume is viewed differently than the data outside of the source volume. We refer to these new cameras as "local cameras".

More specifically, each widget modifies a subset of the default cameras parameters to create each local camera. The remaining set of parameters are the same as the corresponding parameters in the default camera. For example, a widget could change just the zoom parameter of all of the local cameras, creating a fish-eye effect in the source volume. In order to ensure a continous transition between the default view and the local views, the cameras smoothly interpolate to the default one at the boundaries of

the source volume. In order to combine cameras, we build fall-off functions around the selected volume. The fall-off functions are user-controlled. Finally, our framework makes no assumption regarding the representation of the data. The only constraint we require is that the data be sufficiently sampled. If meshes are used to represent the data, then our framework expects triangular meshes. This is the underlying structure of all the widgets presented here. The exact description of a source volume along with the blending techniques used varies from one widget to another. The specifics of each widget will be expanded upon in the following sections.

5.5 The non-linear projection tool-kit

Our tool-kit is composed of three widgets, namely, the unwrap widget, the clipping widget and the fish-eye widget. We believe that these widgets are general enough to be used to generate useful non-linear projections.

5.5.1 Unwrap Widget

Exploring a data-set is commonly done through a series of camera rotations and translations. However, in certain data-sets a small camera rotation results in a significant portion of the data going out of the field of view. The unwrap widget applies a view transformation to bring such data back into the current view. The data remains in the field-of-view irrespective of how the current view is rotated. An example of such data-sets are helices where two surfaces are interleaved. In Figure 5.1(a), it would be impossible to see both section A and section B of the helix in the same view since looking at either part would involve a rotation of the camera that would result in the other moving out of the view. The unwrap widget allows user-selected regions to be rotated into the current field-of-view while ensuring that remaining parts of the model are seen as before.

5.5.2 Clipping Widget

The goal of the clipping widget is to expose structures that are occluded in a particular view. Usually, parts of the model that are farther from the current view-point are occluded by those that are closer (Figure 5.1(b)). The clipping widget allows regions of the model located at different depths along a particular view-direction to be exposed. Changing the global view, results in exposing structures lying at those depths but along a different view-direction.

5.5.3 Fish-eye Widget

The fish-eye widget simulates a fish-eye lens. When a scene is viewed through a fish-eye lens, regions within/close to the center of fish-eye lens appear more magnified than regions that are further away (Figure 5.8).

5.5.4 Design Rationale

Our widgets are inspired by commonly-performed camera operations and object-space transformations. Frequently used camera operations include camera panning, camera dollying, camera rotation and camera zoom. Camera rotation is encapsulated within the unwrap widget, dollying is encaspulated within the fish-eye widget. The clipping widget is derived from the common object-space transformation of using slicing planes to view the internal structure of a model. Combining these widgets aggregates the functionality of these operations and simultaneously applies these deformations to the model.

5.6 Unwrap widget

5.6.1 Overview

As described in the previous section, the unwrap widget is used to rotate selected regions which may not be seen into the current field-of-view of the camera. Only the vertices located within the region are affected, the rest of the model is seen as before.

5.6.2 User's View

The user specifies a source volume which corresponds to the part of the model that the user is interested in viewing from a different direction. The source volume is selected using a 3D widget which contains handles for controlling the position, orientation and scale of the selected volume. If we directly used the camera assigned to volume then the region would be projected to the center of the screen and would therefore be occluded by the rest of the model in the default view. Thus we need to determine a good screen-space position of the projected volume. Usually, the user would have to specify a 2D destination area. The position of this area corresponds to the position on the screen where the volume will be projected while the scale corresponds to the final projected size of the volume.

To simplify our interface, we make a few assumptions about the destination area. First, we calculate the screen-space position of the volume in the default view and which quadrant of the screen it mostly lies in. We then shift the unwrapped region in that direction such that it is lies immediately outside the projected bounding box of the model in the default view. The final calculated position is then displayed and can be changed by the user. We also assume that the final scale of the unwrapped region is the same as the scale of the model in the default view.

Representing a source volume

Internally, the source volume is represented by an implicit function which represents the position, orientation and scale of the volume. Additionally, a fall-off function is associated with the source volume. The fall-off function is used to blend between the

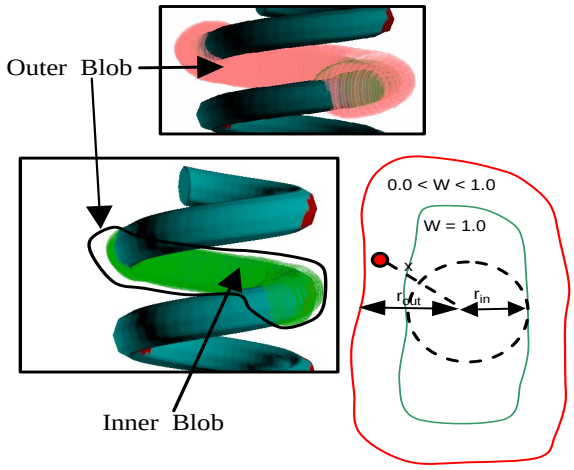


Figure 5.2: The green blob corresponds to the source volume created by the user. The pink blob corresponds to the outer region of the fall-off function which is automatically calculated by our system.

points projected using the new camera and the default view (Figure 5.2).

5.6.3 Methodology

Each vertex in the model is assigned a local camera. The parameters of the local camera are computed based on the location of the vertex. Vertices lying within the source volume are assigned a rotated camera C_{rot} looking at C_{at} whose view-direction l_{rot} is given by Equations 5.1a - 5.1c:

$$\overline{v_{rot}} = Cross(\overline{Norm}, \overline{Look}) \tag{5.1a}$$

$$l_{rot} = R(v_{rot}, \theta)\overline{Look} \tag{5.1b}$$

$$C_{at} = Centroid \tag{5.1c}$$

Essentially, Equations 5.1a - 5.1c states that the source volume is viewed through a rotated camera that looks at the centroid of the region along the surface normal at that point. In other words, for the unwrap-widget the local camera is a camera that

R	:	Rotation matrix, axis v_{rot} and magnitude θ
$\overline{Look}$	:	View-direction of the default camera
$\overline{Norm}$	:	Surface normal at the centroid of the source volume
θ	:	Angle between $norm$ and $look$
C_{at}	:	At point of the camera

looks at the surface volume from the intended direction. The screen-space position of the rotated region is calculated as explained in the previous section.

$$COP^{'} = (-c_x, -c_y) \tag{5.2}$$

$COP^{'}$	:	Center-of-projection of local camera
$(-c_x, -c_y)$	:	Calculated center of the destination area

Finally, all vertices lying outside of the source volume are assigned the default camera. Thus, the rest of the model is guaranteed to be projected as before while only the source volume is assigned a different local perspective.

5.6.4 Blending Views

The advantage of our method is that while the projection of only a part of the model is modified to be seen in the current view, the rest of the model remains seen as before. The transformation is entirely view-dependent leaving the underlying geometry intact. Therefore as the user rotates the model, the selected region is always seen.
Unfortunately, applying a view-transformation to only a section of the model can introduce artifacts at the boundaries. Artifacts occur along the seams where triangles share vertices that are projected with different cameras. To reduce such artifacts and create smooth transitions from the rotated view to the default view, we blend the different views. We employ fall-off functions for blending camera projection of the vertices that lie within an intermediate region between the source volume and the

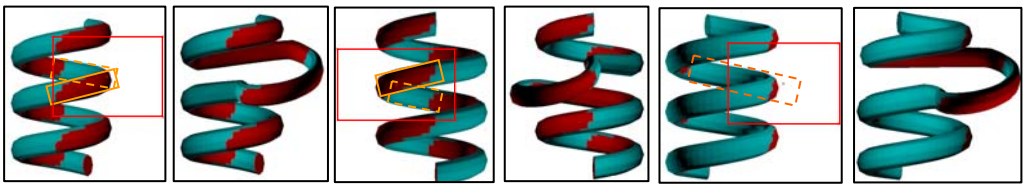


Figure 5.3: The orange box denotes the 3D volume that was selected, while the red box denotes the 2D destination area.

rest of the model. The fall-off function is built according to Equation 5.3.

$$w(Q) = \begin{cases} 1 & ||Q - C|| < r_{in} \\ g(\frac{\|Q-C\|-r_{in}}{r_{out}-r_{in}}) & r_{in} \leq \|Q - C\| \leq r_{out} \\ 0 & \|Q - C\| > r_{out} \end{cases} \tag{5.3}$$

Q	:	3D vertex position
w	:	Weight of Q
C	:	Center of the source volume
$g(x)$	=	$(x^2 - 1)^2$, $x \in [0, 1]$
r_{in}	:	Inside radius of the source volume
r_{out}	:	Outer radius of the source volume

Equation 5.3 defines a fall-off function based on 2 parameters namely r_{in} and r_{out}. r_{in} is calculated based on the scale of the source volume (Figure 5.2). r_{out} is user-controlled.

Blending projections of points

After assigning a weight to those vertices which lie in and around the source volume, the final projection of the vertex is computed according Equation 5.4. Each vertex is simply a weighted combination of the projections using the local camera and the default camera.

$$q = (1 - \sum_{k=1}^{n} w_k)D(Q) + \sum_{k=1}^{n} w_k C_k(Q) \tag{5.4}$$

n : Total number of local cameras
Q : 3D vertex position
D : Default camera view
C_k : k^{th} local camera
w_k : Degree of influence of k^{th} local camera on vertex Q

5.6.5 Examples

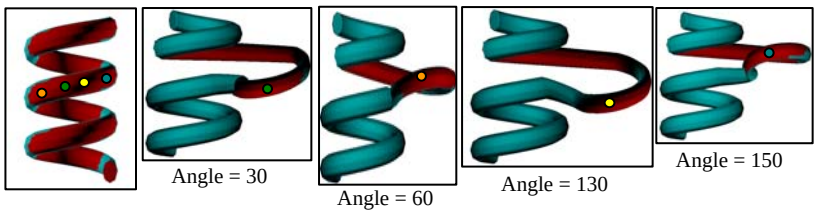


Figure 5.4: The different points correspond to the different positions of the source volume. The angles in each picture denote the angle between the surface normal at the center of the source volume and the default look vector

Figure 5.3 shows the different unwrappings obtained when changing the default camera. As expected, the unwrapping remains consistent even as the default view changes. Since our transformation is entirely view-based the unwrapping is invariant to any change in the default camera's rotation or translation parameters. Figure 5.4 shows the different unwrappings obtained when changing the position of the source volume.

5.7 Clipping widget

The clipping widget is used to view occluded structures in a model.

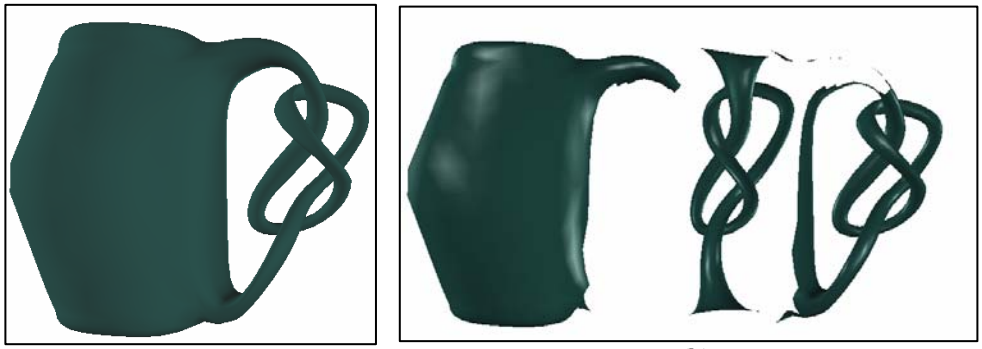
5.7.1 User's View

As before the user specifies a default view-point which determines the overall projection of the model. In addition, the user specifies one or more sections of the model that need to be exposed. These sections can be thought of as a set of slices taken along a particular viewing direction and are specified by sliding planes through the

Figure 5.5: (a) The original model drawn with a complicated set of sillhouete lines. (b) The final non-linear perspective rendering with sillhouetes. Note that the original frontal view of the model and the isolated section are seen in the same window. The original model retains all the sillhouete lines except in the isolated section

model. This is the source volume. The destination area is computed using the same method as in the unwrap widget. One important difference between the clipping and the unwrap widget is that in case of the clipping widget no blending is performed. This is because a given vertex should be associated with only one camera : the default camera or the local camera. Thus, only a cut-out of the relevant section is done. (Section 5.6.2)

Figure 5.6: The final non-linear perspective view. The isolated section shows some features occluded in the original model. Note that both the frontal and isolated section are seen in the same window.



(a) (b)

Figure 5.7: (a) The front-view of the vase: The internal knot is completely occluded. (b) The front-view and the knot seen side-by-side.

5.7.2 Methodology

We employ the near and far planes of a camera in order to control the occlusion of different sections. Vertices that lie within the source volume are assigned a local camera whose near and far clipping planes align with the boundaries of that volume. Vertices that do not lie within any source volume are assigned to the default camera.

5.7.3 Examples

The key advantage of this widget, is that both the occluding and the occluded structures can be viewed simultaneously in screen-space.

The Bone data-set

The model of the bones in a human hand is complex because there are several overlapping bones lying at different depths along any given viewing direction. Different parts of the bone are seen in different views. We illustrate the clipping widget in two contexts. One is to simplify viewing of the sillhouetes on this model (Figure 5.5). The other is to view different parts of a bone that may not have been seen in the default view (Figure 5.6).

Knotted vase

We used the clipping widget on the knotted vase. In this model, there is a knot in the middle of the vase, which is completely invisible in the front view of the vase. The

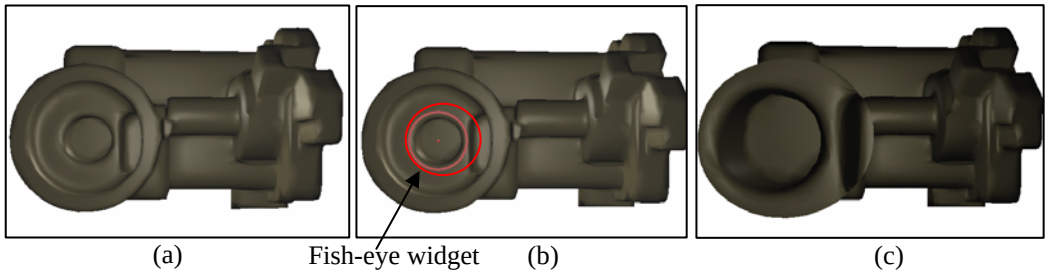


Figure 5.8: (a) Original model (b) Fish Eye widget (b) After applying the widget, the central hole appears magnified while the rest of the model is intact.

clipping widget allows both the front view and the knot to be seen simultaneously (Figure 5.7).

5.8 Fish-Eye widget

5.8.1 User's View

When a fish-eye lens is applied to a model, only the front-facing faces are affected. A fish-eye is represented by two circles (Figure 5.8(b)). The inner circle represents the area in which the actual fish-eye zoom takes place. All vertices that project to the inner circle in the default view are subjected to a fish-eye zoom. This implicitly defines the source volume. The area between the inner and outer circle represents the blend region from the fish-eye zoom to the default view. Moving the widget around the model changes the region over which the fish-eye is applied. Each fish-eye is also associated with a magnification factor, m. Changing m results in a larger amount of zoom associated with the fish-eye. Finally, all vertices that lie outside the area covered by the outer circle are projected using the default view.

5.8.2 Methodology

Similar to the methods outlined in the previous section each vertex on the model is assigned a camera. Vertices projected into the inside circle of the fish-eye are

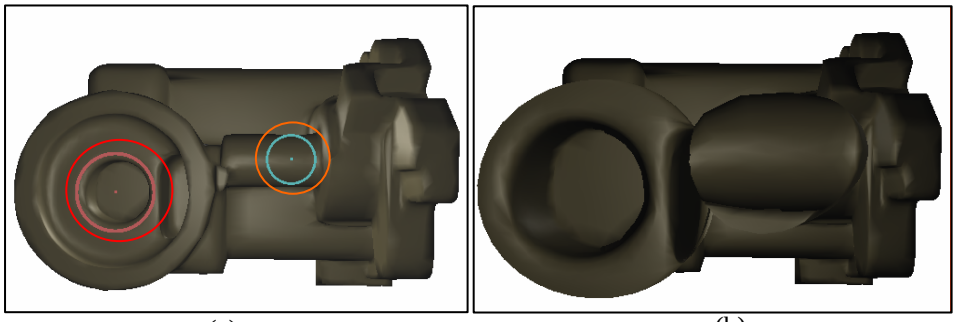


Figure 5.9: Multiple fish-eye cameras

assigned a camera given by Equations 5.5a - 5.5b. Essentially, this equation builds a camera that has been translated down the view-direction of the default view while simultaneously changing the focus distance of the camera. In addition, the center-of-projection of the camera is adjusted so that the 3D point corresponding to the center of the fish-eye remains at the center even in the new camera. Vertices that are projected into the transition area between the inside and outside circles are assigned a camera that is a weighted combination of the fish-eye camera and the default camera. Blending of the cameras is done by combining the corresponding camera parameters (Section 5.6.4).

$$f_d^{'} = f_d \times (1.0 - 1.0/m) \tag{5.5a}$$

$$COP^{'} = COP + (p - c) \tag{5.5b}$$

$f_d^{'}$, $COP^{'}$: Focus distance and center-of-projection of fish-eye camera
f_d, COP : Focus distance and center-of-projection of default camera
m : Magnification factor
c : Center of fish-eye widget
p : Projected version of 3D point P corresponding to c
Figure 5.8 and Figure 5.9 illustrate the working of single and multiple fish-eye widgets on a single model.

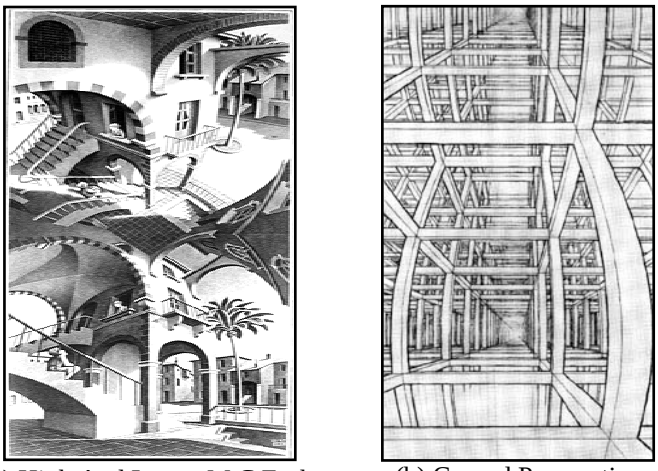
Chapter 6

Curved Perspective

In the previous chapter, we illustrated the application of non-linear perspective for data exploration. In this chapter, we demonstrate another application of non-linear perspective, namely, to simulate a few specific artistic effects.

6.1 Motivation

Usually, linear perspective is used to project 3D scenes onto the 2D image plane. In classical perspective, points at infinity are projected to vanishing points on the 2D plane. Straight lines tending toward infinity in the real world converge as straight lines at the vanishing points of the scene. This is an approximation of our perception of the real world. In reality, our perception of the real world is not so linear. Our overall view of a 3D scene is an aggregation of several individual views. Thus, converging lines in the 3D world are not perceived as straight lines but are instead perceived as converging curves. This leads to the concept of "curved perspective".
A scene viewed in curved perspective has curves converging at its vanishing points rather than straight lines. The advantages of curved perspective over linear perspective are several. Curved perspective produces perceptually accurate projections. It can be used to emphasize the importance of certain parts of the scene over others since the viewer is naturally attracted to these curved lines. It can also be used to communicate a sense of three-dimensional space on the 2D image plane. Figure 6.1 is a classic example of curved perspective in art. Escher manages to convey the height of



(a) High And Low – M.C.Escher (b) Curved Perspective

Figure 6.1: (a) All M.C. Escher works ©Cordon Art B.V.-Baarn-the Netherlands. All rights reserved. Escher seamlessly combines two panoramic views (views from top and bottom of the building) into a single lithograph while still conveying a sense of expansive space. (b) This grid was used as a stencil for determining the vanishing points of High and Low. Notice the similarity between the curved outlines of the buildings and this sketch.

this building in a limited field-of-view using the central point as a common vanishing point for the two curved perspective views.

6.2 Goal

Our goal is to develop a simple and intuitive interface which allows for real-time creation and modification of projections with a curved perspective.

6.3 Approach

Curved perspective projections form a subset of non-linear projections. In the real-world example, we can consider each world-point the eye looks at as being projected by a different camera. As the eye shifts from one point to another, the view-plane of the camera is shifted. This fits into our non-linear perspective rendering framework

where each point in the scene is assigned a camera. The difference lies in how the camera parameters change from one vertex camera to another. Thus, given the generality of our approach for constructing non-linear projections, we can construct such projections using our framework.

In order to vary the perspective continously across a single 3D model we automatically assign a unique camera to each vertex of the 3D model. The vertex camera is found by varying the center of projection of a default scene camera. The new center of projection is a function of the position of the projected vertex with respect to the vanishing points defined in the scene. We will refer to this function as the warp function.

6.3.1 Mathematical Model

If M represents the linear perspective projection matrix then the transformation of a world space point P into the corresponding camera space point c is given by the matrix equation:

$$c = MP \tag{6.1}$$

As explained before, M is a 4×4 matrix which is a function of 11 camera parameters, one of them being the center of projection of the scene (Section 2.3).

We define a warp function g, which represents the amount of warp c is subjected to, given the current set of vanishing points in the scene. Thus, g takes in the camera point c and the vanishing points of the scene. The range of g is the interval $[0, 1]$. g can be any monotonically increasing function. In the simplest case, g is a constant function resulting in linear perspective. The examples illustrated g is a sinusoidal function. Given n vanishing points in the scene we can define the new center of projection for c as:

$$COP' = \mathsf{OldCOP} + \sum_{i=1}^{n} w_i g(c, V_i)\vec{v} \tag{6.2}$$

V_i : i^{th} vanishing point in the scene.
$\vec{v}$: Vector from V_i to c.
w_i : Weight of the i^{th} vanishing point in the scene. $w_i \in [0, 1]$.

A new projection matrix $M^{'}$ is constructed using the new center-of-projection, $COP^{'}$. The world space point P is then reprojected using $M^{'}$ resulting in a new camera space point $c^{'}$.

$$c^{'} = M^{'}P \tag{6.3}$$

This concept can be extended to defining multiple curved cameras on a single 3D model. This feature is useful when the user wants to define multiple warps in order to highlight certain local features on the model. Given m curved cameras defined on a model, Equation 6.2 can be rewritten as:

$$COP^{'} = \mathsf{OldCOP} + \sum_{j=1}^{m} w_j \sum_{i=1}^{n} w_i g(c, V_i)\vec{v} \tag{6.4}$$

w_j : Weight of the j^{th} curved camera. $w_j \in [0, 1]$.

6.4 User's view

Our goal is to present users with an interface that allows for real-time modification of curved perspective projections. The simplest interface for manipulating perspective in a scene is based on vanishing points. In general, people can intuitively position the vanishing points given a scene, and conversely given the vanishing points reconstruct the perspective projection of a scene. Thus, our interface is based on vanishing points.

Perspective primitives are screen-space widgets defined for one or more vanishing points in the scene and are analogous to the primitives that artists sketch to layout 3D scenes on a 2D canvas (Figure 6.2(a)). The user can click and drag a widget handle on these primitives to modify that particular vanishing point. The handles are reset after a vanishing point is modified. Thus it is possible to place vanishing

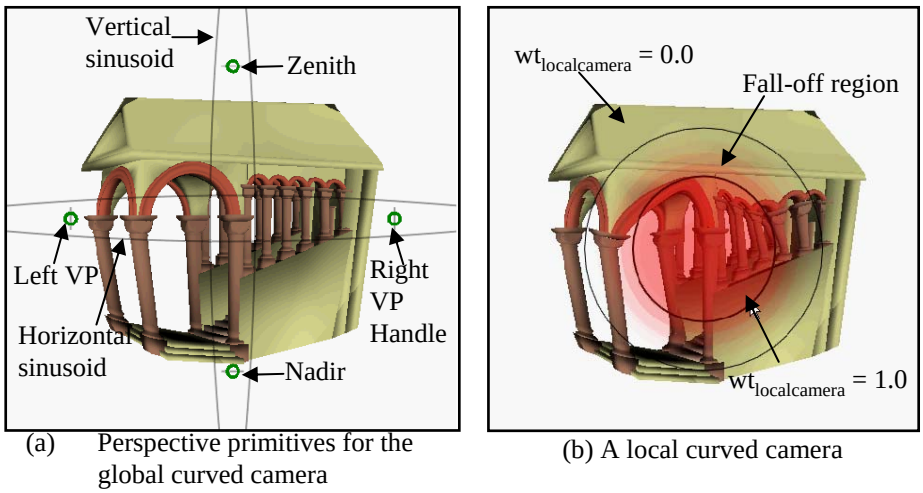


Figure 6.2: (a) Figure shows the perspective primitives for manipulating a single curved camera that is active over the entire model. Two sinusoids are drawn, each containing handles for manipulating the vanishing points that lie on that particular principal axis. (b) Figure shows a local camera represented by a single fall-off function.

points outside the image plane just as artists do on a real canvas.

In the case of multiple local curved cameras defined over a single model, the user can pick a region on the model to place the camera. Currently, a curved camera is represented by two circles, the inner circle representing the region of the model over which the local camera is active and the region between the inner and outer circle represents the transition region (Figure 6.2(b)). The transition region represents the region over which the local curved camera is blended with the global curved camera using a fall-off function similar to the type introduced in the previous chapter (Section 5.6.4). The position and size of the fall-off function are all user controlled.

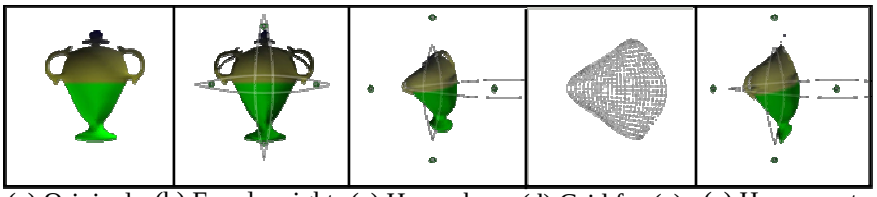


Figure 6.3: Figures show the effect of changing the weights on the vanishing points of a curved camera

6.5 Examples

6.5.1 Single curved camera

We begin with a linear perspective view of a vase as shown in Figure 6.3(a) . The curved perspective of the vase is shown in Figure 6.3(b). The perspective primitives are drawn as sinusoids whose ends represent the vanishing points of the scene. Thus there are 2 perspective primitives : one controlling the zenith and nadir of the scene, the other controlling the left and right vanishing points of the scene. Note how the handles of the vase have curved inwards and the bottom of the vase has become narrow. Our method retains the original lighting of the vase even after re-projection of the vase using curved perspective. Figure 6.3(c) shows the curved perspective of the side-view of the vase obtained by assigning very heavy weights to the left and right vanishing points. Note how the right vanishing point is located outside the picture. This is equivalent to placing a vanishing point very far away or in the extreme case, at infinity. Thus the right-side of the vase is not as warped as the left side of the vase. Figure 6.3(e) is obtained by assigning very heavy weights to the zenith and nadir. The zenith and nadir are very close to the vase itself resulting in the lid of the vase almost disappearing.

6.6 Multiple curved cameras

Figure 6.4(b) shows the curved perspective view of the building obtained by applying a global curved camera. Figure 6.4(d) is obtained by defining two local curved cameras

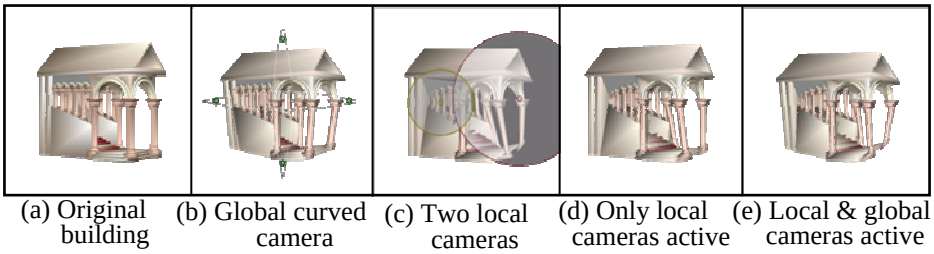


Figure 6.4: Figure demonstrates the effect of adding two local curved cameras in addition to the global curved camera

represented by the circles (Figure 6.4(c)). The radius of the circle defines the region of influence of that camera and the grayscale value represents the weight of the camera. Thus the right side camera influences a larger number of vertices of the model and has a slightly larger weight than the left side camera. Note how the edges of the building remain the same as in the original model but the side pillars curve inwards as do two of the large pillars. The rightmost pillar remains straight since it lies exactly along the central axis of the right side camera. Figure 6.4(e) shows the resulting projection obtained by combining the global camera of Figure 6.4(b) and the local cameras of Figure 6.4(c). Note that the curving of the side pillars is retained from Figure 6.4(b) while the curving of the rightmost pillar is introduced from Figure 6.4(c).

Chapter 7

Panoramic views

7.1 Motivation

Panoramas have been used extensively for various purposes, from generating fly-throughs for animations [23] to creating virtual 3D worlds [21]. We implemented panoramas as a part of a novel visualization interface for the Bandit Project, conducted in the Department of Mechanical Engineering. The Bandit project involves deploying a space-probe equipped with a camera, from a satellite. The probe goes around the satellite taking pictures for several reasons that could range from periodic surveys to trouble-shooting mishaps. In order to direct the probe, the controller needs to consider where the probe is, where it is expected to be in the future and then change its path accordingly. The state of the probe is based on what it is "sees" which indirectly reflects where it is relative to the main satellite. Thus, one of the visualization questions in this project is how to aggregate and present to the controller all that the probe sees (and will see) so that the user can make an informed decision regarding the probe's navigation. Panoramic views provide a convenient mechanism for combining multiple such views into a single field-of-view. Panoramas are also a good artistic tool (Figure 7.1) for visualizing scenes that do not fit in a single view.

7.2 Problem statement

Our goal is to construct a single continous panoramic view, given a single camera path $C(t)$.

Figure 7.1: Panoramas are used to concatenate multiple views into a single one. Agrawala et al. ([1]) used automatic techniques to combine photographs into a single view. Their work was mainly motivated art work done by Michael Koller.

7.3 Approach

Panoramic views can be thought of as a special-case of non-linear projections. Thus, they fit quite naturally into our framework. As explained in Section 4.2, in order to generate a non-linear projection, vertices are assigned a local camera followed by blending with a global camera. When generating a panoramic view, we follow the same procedure, the main difference being the local camera that is assigned to each vertex. Local cameras are selected from various points along the camera path provided by the user (Figure 7.2). We then unwrap the camera path on the image plane.

7.4 Implementation

In case of the bandit example we are given a discrete set of cameras, representing the predicted state of the probe. We use the position and orientation of the probe to build a corresponding camera. In order to assign cameras to vertices, we need to create clusters. Each cluster corresponds to the camera. For the general case, we just need a function representing the camera path.
A preliminary algorithm for clustering the different vertices is presented here. In the current method, a given vertex is projected onto the view-direction of every camera to find the one that is the closest. Thus, each cluster contains those vertices that are closest to that particular camera. The center-of-projection of each camera must be

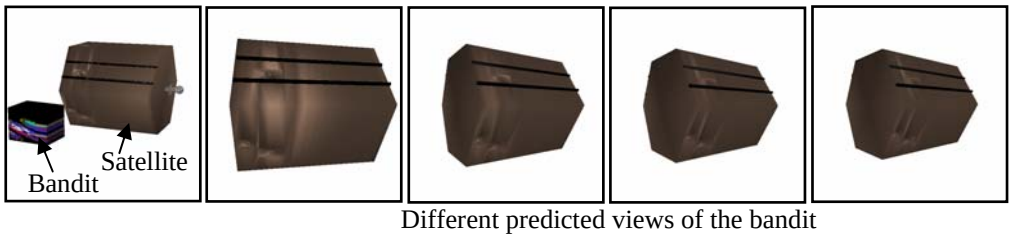


Figure 7.2: (a) The initialization of the bandit. (b) Different views of the satellite from the Bandit. Note how the perceived size of the satellite decreases between views.

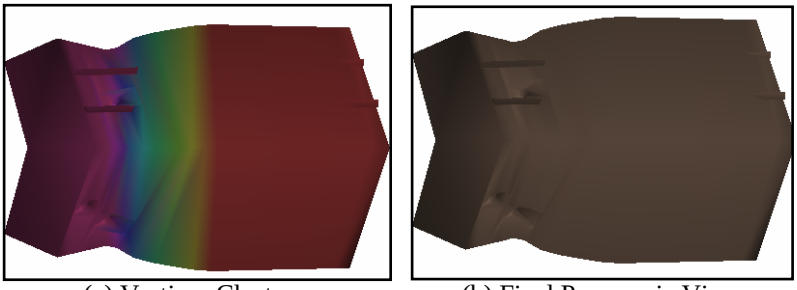
changed in order to create a final smooth image. In order to do this, we determine the projected bounding box of each cluster. From the bounding box we can determine the projected span of any given cluster. Using these spans, clusters are projected such that those corresponding to consecutive cameras (based on the input path) are adjacent to each in the final image.

7.5 Result

The input camera path consisted of 12 views of the Bandit panning away from the satellite. Assigning each vertex to one of these views, creates clusters as shown in Figure 7.3(a). The panorama itself is show in in Figure 7.3(b). The substantial dip in the size of the satellite seen in the left-side of the panorama corresponds to the decrease in the projected size of the satellite as the Bandit moves away.

7.6 Future Work

The algorithm presented here is a prototype. It demonstrates that panoramic paths can be generated using our framework . However, to make the final panorama more artifacts-free we need to make several additions to the current algorithm. One is a better clustering algorithm that should be based on clustering triangles as opposed to vertices and takes into account the direction the camera is moving. Another is



(a) Vertices Clusters (b) Final Panoramic View

Figure 7.3: (a) Each vertex is colored based on the camera it is associated with. (b) The final panorama.

allowing further sampling of the camera path. Finally, blending between adjacent clusters will improve the overall result.

Chapter 8

GPU Implementation

In the non-linear perspective rendering framework, every vertex on the model is projected differently. However, we would like to retain the original lighting effects. As mentioned in Section 2.4, changing the projection changes the lighting. Thus, we have to recalculate the lighting on a per-vertex basis. Assigning different projections to vertices and calculating their original color is a time-consuming process. If done on the CPU, our interface would run no where close to the real-time interface we wish to create. Thus, we implemented the framework on the GPU.

Programs written on the GPU (Graphics Processing Unit) consist of two modules. One module, the **vertex program**, performs transformations on the world space vertex position. Input to this module includes a world-space vertex normal and position along with the color. In addition to these per-vertex quantities, the program also takes in global data (uniform data) which is shared by all vertices. The output of the vertex program is the position of the vertex in camera (screen) space, the transformed normal, and the original color. Finally, these quantities go through a **fragment shader** (**pixel shader**) which performs lighting calculations to output the final color of the pixel.

In our case, we use the vertex program to perform the projection of a given vertex by combining the projections of the different cameras in the scene. Thus, the uniform quantities given to the vertex program are the fall-off functions and the different cameras in the scene (including the default camera). The output is the newly projected vertex in camera-space, the normal projected using the default camera, and the color. The fragment shader then performs the lighting calculations based on the old normal.

8.1 Rendering times

Rendering a non-linear perspective projection with even a single local camera takes an extremely long time on the CPU. We ran a comparison study on a scanned model of a human pelvis (number of vertices = 1289815, number of faces = 49989). The GPU rendering took 0.775422016 seconds to render while the CPU rendering of the same model took 13.42311523 seconds, approximately 17.31 times slower. Our algorithm for rendering on the CPU computed and stored the new projection of vertices every time a local camera was created or changed. In case of the pelvis model, this pre-projection step took 115.95372054 seconds. Thus, even interactive manipulation of the CPU rendering of the final composite image was extremely slow.

Chapter 9

Conclusion

In this thesis, I have presented a tool-kit for performing view-based deformations. CubeCam was presented as an interface for manipulating a single camera which is essentially a view-based deformation (a very simple one). Our non-linear perspective framework is flexible for adapting to different applications while the design of the front-end of the framework varies from one application to another.

References

[1] Aseem Agarwala, Maneesh Agrawala, Michael Cohen, David Salesin, and Richard Szeliski. Photographing long scenes with multi-viewpoint panoramas. *ACM Trans. Graph.*, 25(3):853–861, 2006.

[2] Maneesh Agrawala, Denis Zorin, and Tamara Munzner. Artistic multiprojection rendering. In *Proceedings of Eurographics Rendering Workshop 2000*, pages 125–136. Eurographics, 2000.

[3] Jim Blinn. Where am i? what am i looking at? In *IEEE Computer Graphics and Applications*, volume 22, pages 179–188, 1988.

[4] Stefan Bruckner, Sören Grimm, Armin Kanitsar, and Meister Eduard Gröller. Illustrative context-preserving volume rendering. In *Proceedings of EuroVis 2005*, pages 69–76, May 2005.

[5] Stefan Bruckner and Meister Eduard Gröller. Volumeshop: An interactive system for direct volume illustration. In H. Rushmeier C. T. Silva, E. Gröller, editor, *Proceedings of IEEE Visualization 2005*, pages 671–678, October 2005.

[6] Patrick Coleman and Karan Singh. Ryan: rendering your animation nonlinearly projected. In *NPAR '04: Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering*, pages 129–156, New York, NY, USA, 2004. ACM Press.

[7] Patrick Coleman, Karan Singh, Leon Barrett, Nisha Sudarsanam, and Cindy Grimm. 3d screen-space widgets for non-linear projection. In *GRAPHITE '05: Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 221–228, New York, NY, USA, 2005. ACM Press.

[8] Michael Gleicher and Andrew Witkin. Through-the-lens camera control. In Edwin E. Catmull, editor, *Siggraph*, volume 26, pages 331–340, July 1992. ISBN 0-201-51585-7. Held in Chicago, Illinois.

[9] Cindy Grimm. Post-rendering composition for 3d scenes. *Eurographics short papers*, 20(3), 2001.

[10] Sören Grimm, Stefan Bruckner, Armin Kanitsar, and Meister Eduard Gröller. Flexible direct multi-volume rendering in interactive scenes. In *Vision, Modeling, and Visualization (VMV)*, pages 386–379, October 2004.

[11] K. Henriksen, J. Sporring, and K. Hornbaek. Virtual trackballs revisited. In *IEEE Transactions on Visualization and Computer Graphics*, volume 10, pages 206–216, Mar 2004.

[12] Jeff Hultquist. A virtual trackball. In *Graphics Gems*, pages 462–463. 1990.

[13] X. Jiao and M.T. Heath. Feature detection for surface meshes. In *Proceedings of the 8th International Conference on Numerical Grid Generation in Computational Field Simulations*, Honolulu, HI, June 2002.

[14] Eric C. LaMar, Bernd Hamann, and Kenneth I. Joy. A magnification lens for interactive volume visualization. In H. Suzuki, L.P. Kobbelt, and A.P. Rockwood, editors, *Proceedings of Ninth Pacific Conference on Computer Graphics and Applications- Pacific Graphics 2001*, pages 223–232, Los Alamitos, California, 2001. IEEE, IEEE Computer Society Press.

[15] Michael J. McGuffin, Liviu Tancau, and Ravin Balakrishnan. Using deformations for browsing volumetric data. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 53, Washington, DC, USA, 2003. IEEE Computer Society.

[16] Chunhui Mei, Voicu Popescu, and Elisha Sacks. The occlusion camera. In *Computer Graphics Forum*, volume 24, 2005.

[17] Paul Rademacher and Gary Bishop. Multiple-center-of-projection images. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 199–206. ACM, ACM Press, 1998.

[18] Karan Singh. A fresh perspective. In *Proceedings of Graphics Interface 2002*, pages 17–24, 2002.

[19] Karan Singh, Cindy Grimm, and Nisha Sudarsanam. The ibar: A perspective-based camera widget. In *UIST*, October 2004.

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

[21] Richard Szeliski. Video mosaics for virtual environments. *IEEE Computer Graphics and Applications*, 16(2):22–30, 1996.

[22] Lujin Wang, Ye Zhao, Klaus Mueller, and Arie Kaufman. The magic volume lens: An interactive focus+context technique for volume rendering. In *Proceedings of IEEE Visualization (VIS) 2005*, pages 367–374, 2005.

[23] Daniel N. Wood, Adam Finkelstein, John F. Hughes, Craig E. Thayer, and David H. Salesin. Multiperspective panoramas for cel animation. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 243–250. ACM, ACM Press/Addison-Wesley Publishing Co., 1997.

[24] Jingyi Yu and Leonard McMillan. General linear cameras. In *ECCV (2)*, pages 14–27, 2004.

Vita

Nisha Sudarsanam

Date of Birth July 23, 1982

Place of Birth Chennai, India

Degrees B.E. Computer Engineering, May 2003
M.S. Computer Science, August 2006

Publications Nisha Sudarsanam, Cindy Grimm, Karan Singh (2005). "Interactive Manipulation of Projections with a Curved Perspective", *Eurographics short papers*, volume 24 : 105–108.

Karan Singh, Cindy Grimm, Nisha Sudarsanam (2004). "The IBar: A Perspective-based Camera Widget", *UIST*.

Leon Barret, Patrick Coleman, Nisha Sudarsanam, Karan Singh, Cindy Grimm (2005). "3D Screen-space Widgets for Non-linear Projection", *Graphite*.

August 2006