NONLINEAR TEXTURE MAPPING TECHNIQUES FOR GENERAL PURPOSE GRAPHICS HARDWARE

By

Dongho Kim

B.S. in Electronics Engineering, February 1990, Seoul National University

M.S. in Electrical Engineering, February 1992, Korea Advanced Institute of Science and Technology

A Dissertation submitted to

the Faculty of

The School of Engineering and Applied Science

of the George Washington University

in partial satisfaction of the requirements for the degree of

Doctor of Science

January 31, 2003

Dissertation directed by

James K. Hahn

Professor of Engineering and Applied Science

ABSTRACTS

As graphics hardware is available on most PCs these days, hardware-assisted real-time rendering becomes more important in many applications, such as games or virtual reality. Texture mapping is also important in graphics rendering, because it gives much visual detail without modeling of complex geometry. Conventional texture mapping by graphics hardware relies on the linear interpolations of the texture coordinates inside the triangles. Although this approximation works fine for most applications, the nonlinear modeling of texture coordinates is needed in some applications.

This dissertation discusses how to model and implement nonlinear texture mapping with graphics hardware. Two applications using nonlinear texture parameterizations are introduced. They are cylindrical projective texture mapping and interactive image warping. Nonlinear texture mapping for these applications are modeled and implemented using environment mapped bump mapping (EMBM) capability and programmable pixel shader. By the decomposition of the nonlinear texture parameterization into the linear and nonlinear components, nonlinear residuals can be controlled by EMBM or pixel shader, while the linear components are used as texture coordinates for the conventional texture mapping.

TABLE OF CONTENTS

LIST OF FIGURES

1 Introduction

One of the main components of computer graphics is how to render a scene. Since the advent of computer graphics, researches in rendering has been pursued for two objectives, speed and quality.

Rendering speed and quality have been enhanced via novel algorithms or the implementations of powerful graphics hardware. Many algorithmic improvements have drastically changed what graphics rendering can do. But the developments of superior graphics hardware have also played important roles in the history of graphics research. Moreover, the recent advances in graphics hardware brought very powerful rendering capability to the ubiquitous personal computers. Therefore, real-time rendering with hardware assistance is receiving more attentions in academia as well as industry.

Texture mapping is a technique to represent surface details using texture images. It has played an essential part in rendering since it was introduced in 1970s, because it provides much visual detail without complex models. Texture mapping is now attracting more considerable attentions because the recent graphics hardware is equipped with very powerful texture mapping engine. Moreover, computer users are exposed to more texture-rich applications, such as games or virtual reality.

A texture map is generally a two-dimensional image, and accessed using the texture coordinates assigned to the geometric models in the scenes. Although colors are the contents of

texture maps in most cases, texture elements (texels) can have other attributes, such as transparency, bump perturbation, reflection, shadows, or surface displacements.

In order to map textures to given geometric models, the geometric entities should be parameterized to determine which portion to map from the textures. While the method of parameterization is free to choose with software implementation, there is the following limitation in the parameterization used by general purpose graphics hardware.

Most of the graphics hardware is optimized to render texture-mapped triangles very fast. The interior regions of the triangles are parameterized by linear interpolations of texture coordinates assigned to the vertices.

The main theme of this dissertation is to present the methods to overcome this limitation.

1.1 Motivation

There are many applications which use nonlinear texture mapping or image warping. These applications cannot take advantage of fast rendering with the assistance of graphics hardware, because of the limitation of linear interpolation. Polygonal models may be tessellated further to overcome this hurdle. However, it decreases rendering performance, and it is very hard to provide pixel-level nonlinearity even with in-depth tessellation.

There have been recent advances in graphics hardware capabilities that enable the control of the texture parameterization at the pixel level.

First, environment mapped bump mapping (EMBM) can perturb the texture parameterization according to the sampled texture values of a separate perturbation map. Its original objective is bump mapping, which simulates irregular surface details without complex models. But it can be used for any other applications using nonlinear texture mapping.

Second, programmable pixel shader brings the possibility of more general controls of texture parameterization. Pixel shader is programmed with its own assembly language, and loaded onto the graphics hardware. Then, whenever pixels are rendered, the execution of pixel shader can determine the texture parameterization or the method of texture composition.

Although these capabilities are available from recent graphics hardware, there have been no systematic approaches to apply them to the wide range of nonlinear texture mapping applications. In this dissertation, I want to address this problem and apply the methods to two applications, cylindrical projective texture mapping and interactive nonlinear image warping.

1.2 Problem Domain

Texture mapping is a key technique for high quality graphics rendering. It has been used for a long time to provide surface details to geometric models. In order to map textures, each geometric entity should be parameterized to determine where to sample the texture map.

Recent graphics hardware provides powerful texture mapping performance. Hardware-assisted texture mapping is more important these days, because there are many texture-rich applications and fast texture mapping engines are available for personal computers, with a lot of texture memory.

When graphics hardware performs rendering, only triangles are rendered. Although it is possible to model with parametric surfaces or general polygons on graphics API level, they are tessellated into triangles before rendering. For texture mapping, the texture coordinates are assigned to the vertices of the triangles, and the coordinates are interpolated linearly in the interior regions. Therefore, it is not possible to represent nonlinear texture parameterization with simple approaches.

In image-based rendering (IBR), the scene data are not geometric models, but images. Those images can be photographs or the results of previous synthetic rendering. IBR is widely used in many applications, because it provides fast and realistic rendering.

Among various approaches of IBR, cylindrical panoramic rendering is used a lot in many look-around applications. Input data are photographs or images parameterized in cylindrical coordinates, and they can be constructed easily by stitching process of several photographs. Rendering procedure is the projection of cylindrical panorama onto a planar screen. Because this process involves nonlinear image warping, it is not a simple task to implement with graphics hardware.

Projective texture mapping is a technique to assign the texture parameterization automatically. Texture coordinates are determined as if textures are projected perspectively onto the scene geometry. This technique is used in many IBR applications, since real or synthetic scenes captured by camera can be used as projective textures with the same projection parameters of the camera.

There are many algorithms to implement two-dimensional image warping, based on various geometric transformations. Among them, mesh-based warping computes the warp by cubic interpolation from the translations of the control points in the mesh. Feature-based image morphing computes a smooth transition between two images, so that the corresponding features are matched after the transition. Most of these approaches are implemented in software, because they need to evaluate nonlinear functions at all pixels. Therefore, it is difficult to obtain real-time or interactive implementation for these nonlinear warping applications.

1.3 Proposed Solution

In this dissertation, I present the methods to control nonlinear texture parameterization with general purpose graphics hardware.

Nonlinear texture parameterization can be decomposed into linear approximation and nonlinear residuals. When linear components are assigned to the texture coordinates of the vertices in geometric entities, the interiors of the entities have linearly approximated texture

mapping. Then, nonlinear components are used to perturb the texture coordinates, which are the results of linear interpolation. This perturbation is performed per pixel.

There are two methods that can be used for the perturbation.

First, multiple texture mapping is configured. This means that we can use multiple textures in a single rendering pass. Then, environment mapped bump mapping (EMBM) hardware can perturb the texture coordinates in the second stage by using the texel values sampled in the first stage.

Second, the recent version of programmable pixel shader provides the programmable control of texture coordinates. When the control parameters are applied to the pixel shader program appropriately, the recomputed texture coordinates can be used to access the texture map.

The first method can be implemented by pixel shader as well. But EMBM capability is available on a wider range of graphics hardware.

In this dissertation, two applications are presented using the above framework.

Cylindrical projective texture mapping is an extension of the conventional projective texture mapping. It uses cylindrical panorama as the source of the projection. Nonlinear texture parameterization involved in this process is implemented using the texture coordinate perturbation by EMBM hardware.

Interactive nonlinear image warping or morphing can be implemented using the perturbation scheme by programmable pixel shader. Specific warping model is applied only to the selected control points, and texture coordinates of all other pixels are determined by the nonlinear interpolations. Fast cubic interpolation is achieved, because programmable pixel shader computes the nonlinear components per pixel.

1.4 Original Contributions

This dissertation describes the work believed to be original and contributory in the following aspects.

Hardware-assisted nonlinear texture parameterization techniques are presented. These techniques take advantage of the recent advances of hardware capabilities, such as environment mapped bump mapping (EMBM) and programmable pixel shader.

Hardware-assisted projective texture mapping is extended to use cylindrical panorama as its input data. It uses EMBM perturbation described above. When the viewpoint is fixed at the source of the projection, it becomes the panoramic viewer. This panoramic viewer can render very fast, compared with many other commercial software implementations, such as QuicktimeVR.

Nonlinear image warping or morphing can also take advantage of hardware assistance. Nonlinear warping model is applied only to selected control points, and all other pixels are

computed by cubic interpolations with the help of the computations by programmable pixel shader. Since it provides smooth cubic interpolation at high speed, real-time interactive warping can be obtained with acceptable quality.

Part of the work described in this dissertation also has been published in two peer-reviewed journals [Kim02, Kim03].

1.5 Document Organization

Chapter 2 reviews the previous researches in various domains, such as texture mapping, hardware assisted texture mapping, projective texture mapping, image-based rendering, and two-dimensional image warping.

Chapter 3 proposes the main idea of this dissertation. It describes about nonlinear texture mapping and the decomposition of nonlinear texture parameterization. Then, it explains the uses of EMBM capability and programmable pixel shader.

Chapter 4 explains the application of cylindrical projective texture mapping. The formulas for the decomposition are presented, and the implementation details follow. The discussion on the special case of panoramic viewer is also presented.

Chapter 5 discusses the application of interactive image warping and morphing. It presents the method of cubic parametric modeling of texture coordinates, followed by pixel shader implementation. The strategies for the feature-based morphing are also presented.

Chapter 6 presents results from the two applications and chapter 7 concludes the dissertation and presents potential future research. Appendix A shows the implementation of the pixel shader program used in Chapter 5.

2 Problem Domain

2.1 Texture Mapping

Texture mapping has been used for a long time in computer graphics imagery, because it provides much visual detail without complex models. With texture mapping, varying attributes can be assigned to many modeling entities, such as polygons, implicit surfaces, and parametric surfaces. Without texture mapping, all the details should be modeled geometrically, which is very tedious and computationally expensive process.

A texture is an image or pattern to be applied to geometric models. Texture mapping is a process of applying textures to the geometry. So, the main task of texture mapping is to determine which part of the texture is used, given a point of the three-dimensional geometry. In other words, it is a mapping from $\boldsymbol{R^3}$ to $\boldsymbol{R^2}$. For some texture mapping techniques such as solid texturing or volume texturing, the mapping is determined as $\boldsymbol{R^3}$ to $\boldsymbol{R^3}$, because texture coordinates are defined in $\boldsymbol{R^3}$. Heckbert gives a good survey on classical texture mapping algorithms [Heckbert86].

Texture maps are usually two-dimensional color images, and accessed using texture coordinates assigned to the geometric entities of scenes. Although colors are the contents of texture maps in most cases, texture elements (texels) can have other characteristics, such as transparency, bump perturbation, and normal vectors [Blinn78, Heckbert86]. In environment

mapping, texture maps are indexed with reflection vectors on the geometry. When environment maps contain what is visible from the center of the scene, this technique gives the illusion of reflecting the environment via the geometry, although it is not so accurate as computationally expensive ray tracing [Blinn76]. Another extension is the use of three-dimensional texture, and it is called volume texture mapping. Volume texture contains 3D data sampled at regular 3D grids. It is widely used, especially in the visualization of medical data, such as MRI or CT [Levoy88]. In procedural texturing, texture coordinates are determined by a given procedure, rather than explicit assignments. Given three-dimensional positions, solid textures for various noises or marble can be generated by simple procedures [Perlin85, Peachey85]. Most of these exploitations of texture mapping were implemented as software, when they were introduced for the first time.

In order to map textures, texture coordinates should be given to any point on the geometry. There are many ways to determine the texture coordinates.

First, for parametric or implicit surfaces, the parameters of the equations for the surfaces can be used as texture coordinates. For example, two-dimensional parameters making B-spline surfaces can be used as texture coordinates as well, and the longitudes and latitudes can be used as texture coordinates for spheres [Foley90].

Second, intermediate surfaces, such as plane, cylinder, or sphere, can be used for the texture parameterization. With this method, the points on the geometry are projected to the intermediate surfaces to find their texture coordinates [Watt92].

Third, for polygonal meshes, texture coordinates can be assigned explicitly for the vertices, and texture coordinates for the interiors of the polygons are interpolated from the ones assigned to the vertices. In most cases, the interpolation is linear. But other interpolation methods, such as bilinear or affine, can also be used. In order to determine the texture coordinates for the vertices, the previous methods can also be used.

2.2 Texture Mapping by Graphics Hardware

Since texture mapping has been playing a major role in graphics rendering, hardware acceleration of texture mapping has been exploited for a long time together with the development of hardware rendering engines.

Recently, texture mapping has become more important. Many games or virtual reality systems contain the contents full of high detail textures or videos. In order to support these demands for superior texture mapping performance, state-of-the-art graphics hardware focuses on the enhancement of the texture processing speed and the size of the texture memory, and personal computers can be equipped with powerful graphics hardware with low costs. Special purpose texture mapping techniques, such as bump mapping, environment mapping, and 3D volume texturing, are now supported by graphics hardware for real-time purpose [Blythe00, Woo99, DirectX8, DirectX9].

With software implementation of texture mapping like ray tracing, texture coordinates can be determined by any procedural methods described in the previous section. However,

graphics hardware has some limitations. The biggest limitation is that it supports only triangles as rendering primitives. Although more complex entities such as quadrilaterals or parametric surfaces can be used on API or library level, only triangles are sent to the rendering engine as actual primitives for rendering.

With hardware texture mapping, texture coordinates are assigned explicitly to the vertices, and linear interpolations take place in the interiors of the triangles. This limitation prohibits the use of nonlinear texture coordinates for given polygonal meshes. In other words, there is no way of representing the interiors of the polygons with nonlinear parameterization.

With OpenGL or DirectX, texture coordinates for the vertices can be generated automatically without explicit assignments. They can be set as the locations in world coordinates, eye coordinates, or any other special coordinate system. Or projective texture mapping can be used as well, which will be discussed later.

2.3 Programmable Shader

Programmable shader is a new scheme that provides much flexibility in rendering pipeline. With programmable shader, shading procedure can be programmed in its own assembly language and executed inside the graphics hardware. Moreover, programmable shaders are equipped with powerful instructions for color or geometry processing, such as dot product or linear interpolation [Lindholm01, Proudfoot01, DirectX8, DirectX9].

There are two types of shaders in the programmable shaders. Vertex shader is applied to each vertex and can determine many vertex attributes, such as location, texture coordinate, or lighting calculation. When triangles generate pixels by scan conversion, pixel shader can be applied to each pixel in order to manage texture mapping and texture operations.

The pixel shader has been evolving, while changing versions. The most current version is 2.0 and it is specified in DirectX 9. Pixel shader 1.4 supported by DirectX 8.1 and later provides very flexible control of texture parameterization. For example, arithmetic operations can be applied to the texture coordinates or texel values, so that the results can be used for dependent texture read [DirectX8]. However, arithmetic operations in pixel shader 1.4 are not precise enough, because the registers contain only fixed-point numbers.

Pixel shader 2.0 supported by DirectX 9 provides floating-point registers and operations. It is based on 32-bit floating-point numbers, thus not full IEEE floating-point numbers. But it can represent and compute more precise arithmetic numbers and operations [DirectX9].

2.4 Image-Based Rendering using Cylindrical Panorama

In conventional rendering algorithms, we use geometric information defined in three-dimensional Euclidean space. In contrast, image-based rendering generates resultant images from another images as its inputs.

To understand this concept, let us consider the process of computer vision and conventional computer graphics rendering algorithms. Major objective of computer vision is acquiring geometric information from real scenes or photographs. Conventional graphics rendering algorithms are the reverse processing of computer vision. Here, rendered images are generated from three-dimensional geometric data. By combining these two processes in cascade, we can get desired images from other images as the scene representations. In other words, we can replace input information with other images (reference images). Therefore, image-based rendering is doing the rendering process without intermediate three-dimensional geometric information.

We can examine the advantages of image-based rendering in two aspects. First, photorealistic details can be employed easily. For many complex scenes, which we can see in real world, it is very hard to model them as three-dimensional geometric data. We can avoid this modeling process with image-based rendering. Moreover, photorealism can be achieved easily because scene representations can be photographs themselves.

Second, image-based rendering accelerates the rendering performance. Even if we have geometric information for very complex scenes, rendering of the geometry will consume much time because of the scene complexity. But, the rendering speed for most of image-based rendering algorithms does not depend on scene complexity. Therefore, we can get better performance with image-based rendering in general.

There are many approaches in image-based rendering. Among them, this dissertation wants to concentrate on the techniques using two-dimensional cylindrical panorama. Chen presented QuicktimeVR [Chen95]. As input data, QuicktimeVR uses a cylindrical panorama incorporating all horizontal viewing directions from a fixed viewpoint. The panorama can be obtained by stitching several photographs taken by low-cost cameras. Then, QuicktimeVR renders the panorama by the projecting it onto a planar screen.

This method can be viewed as a variation of environment mapping [Greene86], which is one of the traditional texture mapping algorithms. But environment mapping uses texture maps to simulate the reflection of the scene, while QuicktimeVR uses the reference image for direct viewing, to show some part of the panorama according to the current viewing parameters. Every time it renders an image, it performs viewing transformation from cylindrical coordinates of the panorama to the current viewing plane. Then, it performs the cylindrical projection. It also provides panning and zooming from a fixed viewpoint. Since the viewpoint is fixed, QuicktimeVR provides hot spots where the user can jump between pre-selected viewpoints.

To make a panoramic image, several photographs can be stitched together with correlation-based stitching algorithm. QuicktimeVR is now de facto standard for low-cost VR navigation based on the real photographs. The most important advantage of QuicktimeVR is that it can easily use photographs of real scene for simple VR navigation, without expensive

software or special device. There are many free or commercial products based on QuicktimeVR or its variations [PanoGuide].

Shum and He extended QuicktimeVR using three-dimensional input data, which is concentric mosaics [Shum99]. Concentric mosaics consist of many concentric cylindrical images. Each of the cylindrical images contains the collection of what is visible from the viewpoints on concentric circles. Therefore, by incorporating one additional dimension, the radius, the users can relocate their viewing positions in the navigation.

The advantage of cylindrical modeling is that it can capture full 360° viewing directions, and maintain the constant horizontal sampling rate. However, there is also some disadvantages. Rendering with cylindrical models include nonlinear projection from cylindrical coordinates onto a planar screen. Therefore, it is difficult to render it with general purpose graphics hardware, which usually deals with linear projection and linear texture coordinates. It is the reason why most of the current QuicktimeVR-like applications are implemented by software, and do not use hardware acceleration. Therefore, they usually render small images, and the rendering quality is degraded in order to maintain real-time performance when the users change viewing direction.

2.5 Projective Texture Mapping

Texture mapping is performed by the assignment of texture coordinates to all the points on the scene geometry. Usually, texture coordinates are assigned as parametric coordinates of

surfaces or assigned manually. But projective texture mapping provides another method of texture coordinate assignment. With this method, a texture is projected onto object surfaces [Weinhaus99, Blythe00].

Rather than assigning fixed texture coordinates to the geometry, projective texture mapping projects a texture map onto the geometry, like a slide projector. Projective texture mapping is more convenient to use in many applications than assigning fixed texture coordinates. For example, light mapping, which enables fast complex light contributions such as Phong shading or spotlight, can be implemented easily with projective texture mapping. Image-based rendering (IBR) draws more applications of projective texture mapping [Debevec98, Buehler01]. If the images are taken from photographs of real scenes, they can be thought of as being transformed perspectively in the capturing process. Therefore, virtual environment can be modeled easily using the images as projecting sources, making the same condition as the capturing process. Moreover, it eliminates the need for complex or laborious process of texture coordinate assignments.

Most of the current graphics hardware can handle projective texturing [Blythe00, DirectX8, DirectX9]. First, all the points on the scene geometry are given 3D texture coordinates which are their camera space locations. These coordinates are then transformed to the projection coordinate system, where the origin is at the center of the projection. Then, the coordinates are divided by their z-components in order to perform the perspective projections. This procedure is similar to the viewing and projection transformations in normal image

rendering, and can be represented as a single 4×4 matrix. This matrix is set as a texture transformation matrix maintained by the graphics hardware or rendering APIs. When polygons are drawn with this configuration, the automatic transformation of texture coordinates gives the final 2D coordinates, which can be used to access the texture map.

However, current projective texture mapping has some limitations. It can project only one rectangular texture map using one perspective frustum. Moreover, with the limited field of view, the coverage of the projection is limited only to a part of the scene, which may be a problem in many VR applications. In many image-based applications, we must consider the boundaries of projection carefully and two or more projectors should be used as needed. Moreover, cylindrical panorama cannot be used as projecting sources. This dissertation deals with how to overcome these limitations.

2.6 Two-dimensional Image Warping

Warping or deformation of geometric data has been researched for a long time. The geometry used for warping is usually 2D or 3D. When the geometry is defined in two-dimensional space, it can be used for image warping.

2-pass mesh warping algorithm is given in Wolberg's book on digital image warping [Wolberg90, Smythe90]. With this algorithm, two-dimensional rectangular lattice is set up on the input image. Then, the control points on the lattice are translated by the users' specification. In order to generate smoothly warped results, the Catmull-Rom curves [Foley90] are modeled

connecting the control points along *u* and *v* directions. The input image is warped in *u* direction, then *v* direction in 2-pass manner according to the interpolated values from Catmull-Rom curves. This algorithm was used in many movie special effects made by ILM in late 80's.

Beier proposed a feature-based image morphing solution [Beier92]. In order to morph between two given images, the users specify the corresponding features, shaped as line segments. The amount of movement for each pixel is determined by the weighted average of the distances to the features. For a smooth morphing sequence, in-between features are generated by linear interpolation between the corresponding features. In-between images are generated by the warping of the two source images according to the interpolated features, then blending the two images.

Milliron et al. gives a unified framework for geometric warps and deformations [Milliron02]. In this work, many geometric warping algorithms are classified and a general framework for image warping and morphing is presented.

3 Nonlinear Texture Mapping by Hardware

3.1 Nonlinear Texture Mapping for Real-Time Rendering

In this dissertation, nonlinear texture mapping means that the texture coordinates in the interiors of the polygons are modeled nonlinearly. It is not a problem at all with the software rendering systems, such as ray tracing. Any texture coordinate modeling can be used because the software can be implemented in its own way. However, it is not simple with the real-time rendering using graphics hardware.

Most of the real-time rendering APIs, such as DirectX or OpenGL, are based on rendering triangles. The texture coordinates are assigned at the vertices, and they are interpolated linearly inside the triangles. In order to represent complex shapes with complex textures, triangular mesh may be subdivided or textures may be warped according to the shapes of the objects. However, they may be a tedious process, and may decrease the rendering performances.

Recent advances in the graphics hardware and programming APIs brought new features for texture mapping. Texture coordinates can be managed in more flexible ways with these features. But, their current applications have not gone beyond bump mapping or simple example renderings.

3.2 Decomposition of Nonlinear Texture Coordinates

The use of linear texture coordinates inside the triangles is an approximation for the implementation of graphics hardware. Implicit surfaces or parametric surfaces can be rendered with real-time APIs only through the polygonal approximations. Therefore, most of the graphics hardware can perform texture mapping fast with the linear coordinates. However, the nonlinear control of the texture coordinates is needed in some applications.

First, we can start the discussion on the nonlinear texture coordinates by decomposing them into linear and nonlinear parts. With the idea of decomposition, the appropriate methods can be applied for nonlinear components, while linear approximations are used through the conventional texture mapping.

Nonlinear texture coordinates can be decomposed into linear approximation and nonlinear residuals. For linear components, standard texture mapping techniques can be used. The assignment of approximated texture coordinates to the vertices will render linearly approximated texture mapping results. For nonlinear texture mapping, the texture coordinates inside the polygons are perturbed from the linear interpolation, in order to perform nonlinear texture mapping.

If the nonlinear components are calculated again for every pixel, by the software, it cannot take advantage of fast hardware rendering. It is because the cost of the calculation is not less than that of software-implemented texture mapping. Therefore, it is important to extract

the characteristics of the nonlinear components and set up the procedures for the implementation using hardware acceleration, without software recalculation of nonlinear components. The procedures depend on the applications and the characteristics of nonlinear components.

Nonlinear texture parameterization is decomposed into linear and nonlinear part as follows.

$$\begin{bmatrix} u(x,y) \\ v(x,y) \end{bmatrix} = \begin{bmatrix} u_L(x,y) \\ v_L(x,y) \end{bmatrix} + \begin{bmatrix} u_N(x,y) \\ v_N(x,y) \end{bmatrix} \qquad (3.1)$$

In the above equation, texture coordinates, (u,v) is a given nonlinear function of (x,y), which are the local coordinates of the polygons. (u_L, v_L) is the linear approximation of (u,v). Nonlinear residual, (u_N, v_N) is also a function of (x,y).

This dissertation presents two methods of implementations for nonlinear texture mapping. One is the use of bump mapping capability called environment mapped bump mapping (EMBM). The other is using programmable pixel shader.

3.3 Using EMBM for Nonlinear Texture Mapping

Recent improvements in texture mapping hardware introduced multiple texture mapping. With multiple texture mapping, two or more textures can be applied to a polygon in a

single rendering pass. Different textures can be applied at two or more stages. The result of the previous stage and the texture for the current stage can be combined by various operations, such as add, subtract, or multiply, etc.

One of the operational modes of multiple texturing is environment mapped bump mapping (EMBM). With EMBM, texture coordinates in the second stage are perturbed by the texel values sampled in the first stage. This functionality is designed originally for bumped reflection, where the environment is reflected using environment mapping. In this dissertation, however, this functionality is used to perturb the texture coordinates and perform nonlinear warping with hardware support.

The bump perturbation map assigned to the first stage is encoded using (u_N, v_N) in Equation 3.1. Then, with the multi-texture setting, the texture coordinates used in the second stage are perturbed from the linear texture coordinates. This perturbation is sampled and applied for all the pixels rendered.

Cylindrical projective texture mapping described in Chapter 4 will use this method for nonlinear projection of cylindrical panorama.

3.4 Using Programmable Pixel Shader for Nonlinear Texture Mapping

Pixel shader version 2.0 provides a flexible control of texture parameterization. In this version of pixel shader, arithmetic operations can be applied to the texture coordinates or texel values. Then, the results can be used as texture coordinates for the subsequent accesses of textures. Different from the previous version, pixel shader 2.0 uses floating-point registers and operations for more correct results.

If (u_N, v_N) in Equation 3.1 can be implemented with arithmetic operations supported by pixel shader program, the amounts of perturbation can be computed per pixel within the pixel shader. The application using this scheme will be presented in Chapter 5.

The capabilities of pixel shader include the functions of EMBM. But, EMBM is available on a wider range of the graphics hardware. It is the reason why the application in Chapter 4 is implemented with EMBM, not pixel shader.

4 Projective Texture Mapping with Cylindrical Panorama

4.1 Introduction

Projective texture mapping is a relatively new technique of mapping textures. With projective texture mapping, texture coordinates are not assigned to the geometry directly. Instead, texture maps are projected onto the geometry, like a slide projector. Projective texture mapping is more convenient to use in many applications, rather than assigning fixed texture coordinates. For example, light mapping, which enables fast complex light contributions such as Phong shading or spotlight, can be implemented easily with projective texture mapping. Image-based rendering (IBR) draws more applications of projective texture mapping. Before they are used for IBR, photographs or synthetic images are generated by the projection onto the screen in the capturing process. Therefore, projecting the images into the scenes can be used to model image-based virtual environments.

However, projective texture mapping has some limitations. It can project only rectangular texture maps using perspective frustums. Therefore, the field of view determined by the projection is limited. Therefore, in many image-based applications, we must consider the boundaries of projection carefully and two or more projections should be used if needed. Moreover, cylindrical panorama cannot be used as the sources of the projection.

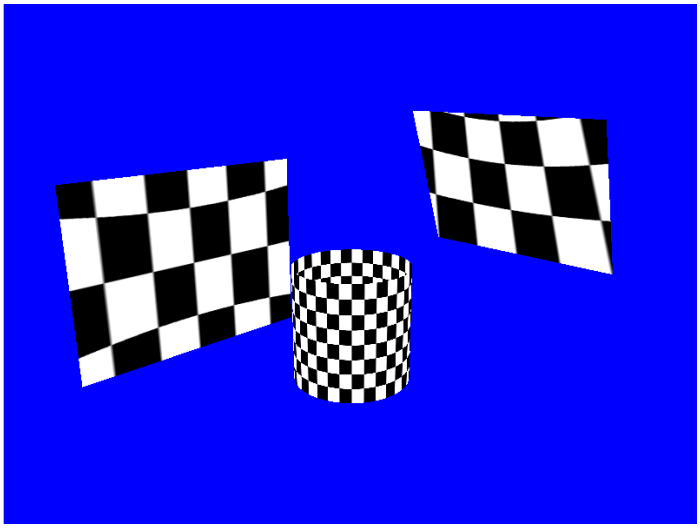


Figure 4.1: Cylindrical projective texture mapping

Cylindrical panoramas are rectangular texture maps, which contain texels parameterized in the cylindrical coordinates. They are being used in many applications, since they provide a 360° horizontal field of view and constant horizontal sampling density. However, it is difficult to use cylindrical maps directly with hardware acceleration, because cylindrical mapping involves nonlinear computations, which cannot be handled easily by graphics hardware. So, texture coordinates are determined by the software renderer in many look-around applications with cylindrical maps, such as QuickTimeVR,.

In this chapter, I propose the concept and algorithms for the projection of cylindrical panoramic images. This means that 360° panorama can be projected to the virtual environments directly. Therefore, many image-based applications can handle the projections in a more convenient way.

Figure 4.1 shows the concept of projective texture mapping from cylindrical panorama. The rectangles receive the projections from the cylinder located in the center. The method presented here achieves real-time performance, because it is implemented fully by 3D graphics hardware with DirectX 8.1 API. From now on, I will refer to the new method as cylindrical projective texture mapping (CPTM) to distinguish it from conventional projective texture mapping.

Projection with cylindrical panorama is the process of warping a cylindrical texture map onto the scene geometry. I propose an algorithm for this process using graphics hardware.

4.2 Projection from Cylindrical Panorama

First, I define *projection plane*, onto which a cylindrical map is warped locally. Then, conventional planar projective texture mapping is used with the warped texture map on the *projection plane*. In order to utilize graphics hardware and enhance rendering performance, we perform the nonlinear local warping on-the-fly using bump mapping hardware.

From now on, (θ, v) is used for the cylindrical texture coordinates in [0..1] for cylindrical panorama, and (u', v') for the planar texture coordinates on the projection plane. 3-vector (x, y, z) is a location in projection coordinate system, where the origin is at the center of the cylinder. The *z*-axis is aligned with the axis of the cylinder, and the *x*-axis is aligned with

$\theta = 0$. The transformation from the eye coordinates to the projection coordinates is performed in the same way as conventional projective texture mapping.

Let H be the height of the unit cylinder given as $\mathrm{H} = 2\tan(\frac{\mathrm{FOV}}{2})$, where FOV is the vertical field of view of the cylindrical panorama. Then, (θ, v) is related to (x, y, z) as follows. Here, *atan2*() is arctangent function in the standard C library, which gives the angle in the range of $[-\pi..\pi]$ as the output.

$$\theta = \frac{\mathrm{atan2}(y, x)}{2\pi} + 0.5$$
$$v = \frac{1}{\mathrm{H}} \frac{z}{\sqrt{\mathrm{x}^2 + \mathrm{y}^2}} + 0.5 \qquad (4.1)$$

For local warping, I define the *direction of projection (DOP)*, which becomes the normal vector of the projection plane. In other words, DOP is the direction that we are most interested in for the projection. DOP is directed so that it is perpendicular to the cylinder, so it can be defined simply by θ_0, which is the angle of DOP from the *x*-axis. Then, the projection plane is placed at unit distance from the origin with the normal vector along DOP. Figure 4.2 shows the two-dimensional top view of this configuration.

Suppose that the scene receiving the projection has the field of view larger than the maximum values for one local warping. Then, the scene can be divided into smaller groups, so that the groups have the different DOPs. Usually, a polygon or triangle does not have that large

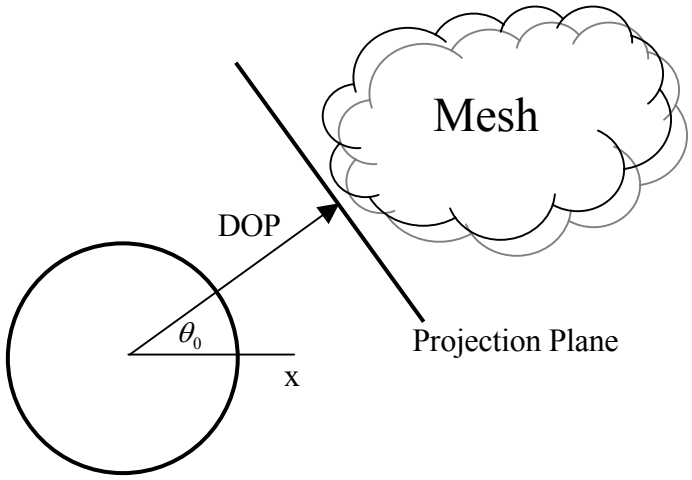


Figure 4.2: Projection Plane and Direction of Projection (DOP)

field of view. Note that the scene geometry receives the projection from the same cylinder after the grouping, and it does not incur any penalty in the performance.

Suppose that (x', y') is a projection of (x, y, z) onto the local coordinates of the projection plane. Then, the above equations become the following equations that determine (θ, v) for the given (x', y').

$$\theta = \theta_0 + \frac{\text{atan}(x')}{2\pi}$$
$$v = \frac{1}{\text{H}} \frac{y'}{\sqrt{1 + \text{x}'^2}} + 0.5 \qquad (4.2)$$

The main idea of the algorithm is to decompose these equations into the linear approximation and the nonlinear residuals as described in Chapter 3. The bump perturbation

Figure 4.3: Correct nonlinear projection and linear approximation

map contains the encoded values of nonlinear components. Then, the bump map is used to perturb texture coordinates determined by the linear part. The decomposition of Equation 4.2 is expressed as follows.

$$\begin{aligned} \theta &= \theta_0 + ax' + R_\theta(x') \\ v &= 0.5 + by' + R_v(x', y') \end{aligned} \tag{4.3}$$

Then, the residual parts are given as follows.

$$\begin{aligned} R_\theta(x') &= \frac{\operatorname{atan}(x')}{2\pi} - ax' \\ R_v(x', y') &= \frac{1}{\mathrm{H}} \frac{y'}{\sqrt{1 + \mathrm{x}'^2}} - by' \end{aligned} \tag{4.4}$$

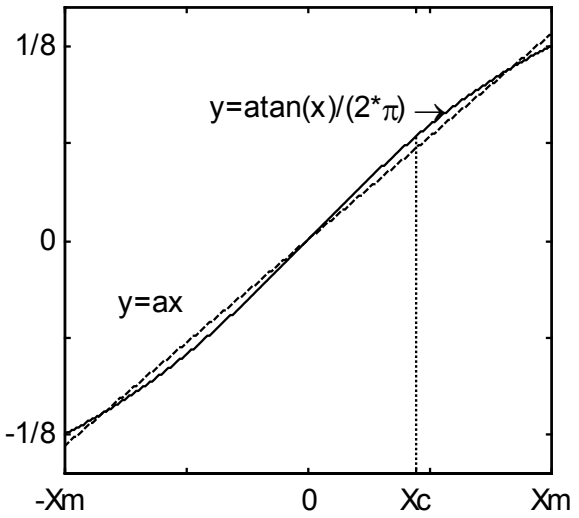
Figure 4.3 shows an example of the cylindrical projection. Assume that the cylindrical texture is projected onto a rectangle, which is aligned with the cylinder. With the linear

components in Equation 4.3, the rectangle will be texture mapped by the portion of the texture denoted by the rectangle in Figure 4.3. It is a good approximation, but the image will look distorted, because the texture is parameterized in the cylindrical coordinates. For correct results, the region in the curved area should be texture mapped. The differences between the two regions are handle by adding nonlinear components in Equation 4.4.

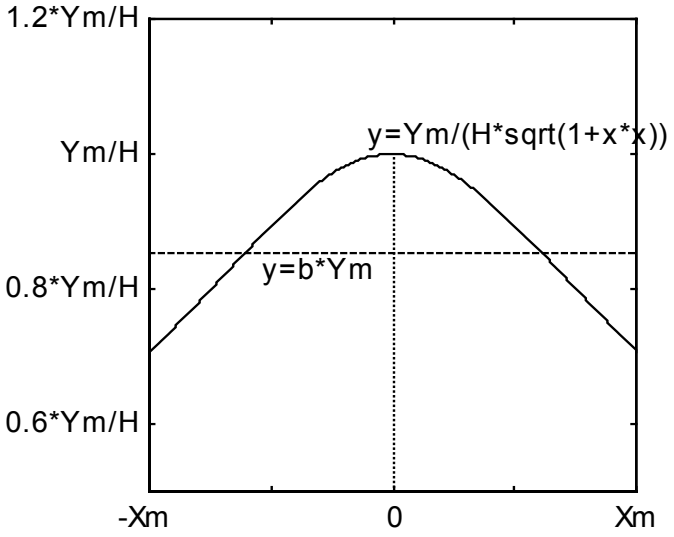
Here, a and b are the coefficients of the linear approximations of (θ, v) according to (x', y'). They are computed before rendering so that the magnitudes of the residual parts are minimized. Because texture coordinates are interpolated linearly by texture mapping hardware, the linear components are encoded as regular texture coordinates of the polygon. The residual components are encoded as bump maps to perturb the linear texture coordinates.

4.3 Computation of Linear Coefficients

Suppose that we consider x' in $[-X_m, X_m]$ and y' in $[-Y_m, Y_m]$. These ranges are determined by the possible maximum values of horizontal and vertical fields of view used by one DOP, from the center of the projection. For instance, if we always use the fields of view less than 90°, the ranges are [-1,1] and [-1,1] for x' and y', respectively, since $\tan(\frac{\pi}{4})$ is 1.



(a)



(b)

Figure 4.4: Determination of the linear coefficients

Figure 4.4(a) shows the first and second terms of $R_\theta(x')$ in Equation 4.4. While we change the slope *a*, the difference of the two terms are minimal, when $R_\theta(X_c)$ and $R_\theta(X_m)$ have the same magnitude and the opposite signs. This is the first condition. X_c is obtained by setting the derivative of $R_\theta(x')$ to zero, which becomes the second condition. Therefore, we have two unknowns, X_c and *a*, and two equations. The equations can be solved by substitution, followed by a few iteration of simple bisection root finding algorithm [Press88].

For $R_v(x', y')$ in Equation 4.4, it is apparent that the magnitude is at maximum when y' is equal to Y_m. Figure 4.4(b) is the plot of the two terms of $R_v(x', Y_m)$. In the similar way to $R_\theta(x')$, the maximum magnitude of $R_v(x', Y_m)$ is minimal, when $R_v(0, Y_m)$ and $R_v(X_m, Y_m)$ have the same absolute values and the opposite signs. This gives the equation to solve for *b*.

4.4 Implementation

Cylindrical projective texture mapping is a process of nonlinear texture mapping and can be implemented using multiple texturing capability and EMBM of the 3D graphics hardware described in Section 3.3. With multiple texture mapping, two or more textures can be applied onto the object geometry in a single rendering pass. Because residual parts shown in Equation 4.4 are approximated as a bump perturbation map, the resolution of the bump map may affect the quality of projective texturing. 256×256 bump map is used in this work, and there is no noticeable artifact due to the approximation.

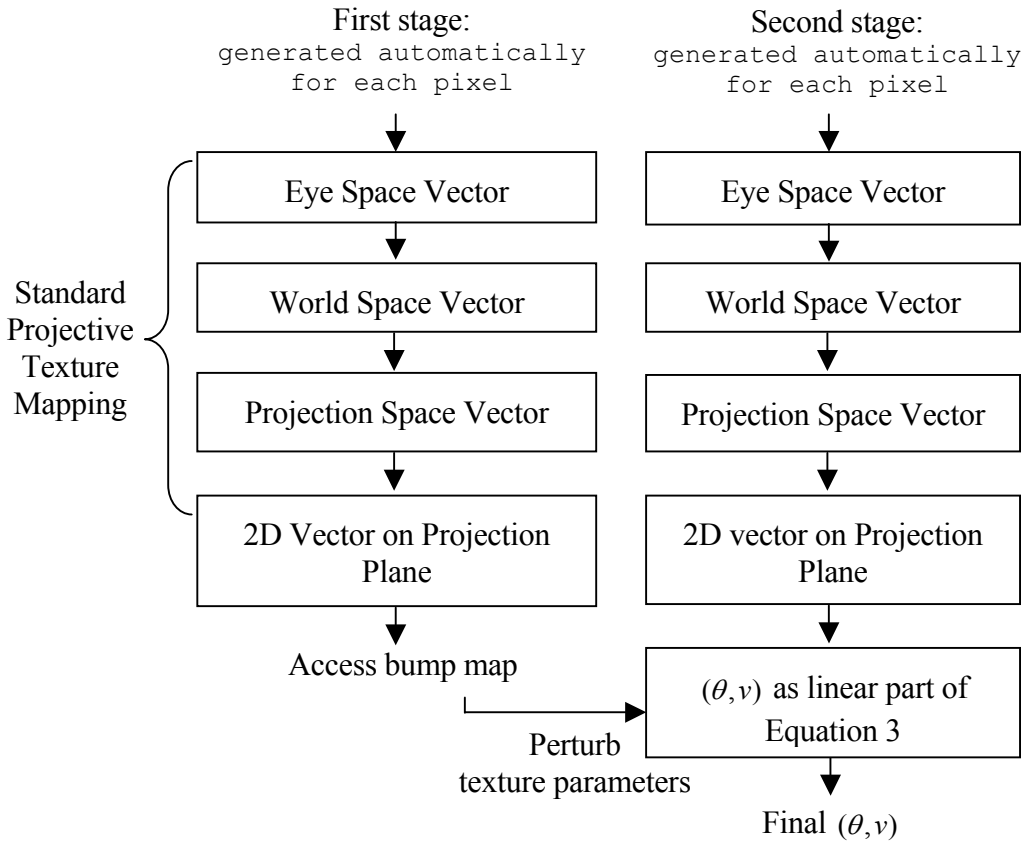


Figure 4.5: Determination of texture coordinates for CPTM

The algorithm can be implemented in two ways explained in the next two subsections. Although DirectX 8.1 is used for the implementation, OpenGL could also be used.

4.4.1 Implementation with single-pass rendering

Figure 4.5 shows the procedure for the determination of the texture coordinates for cylindrical projective texture mapping. In this figure, the projection space means the coordinate

system, where the x and y axes are in the projection plane and the z axis is in the direction of DOP.

Multiple texture mapping is configured as follows. For the first texture stage, texture operation is set as EMBM and bump perturbation map is used as texture. The bump map contains the amounts of the perturbations needed for all texels, which are computed from the residual parts of cylindrical parameterization given in Equation 4.4. For the second texture stage, the cylindrical panorama is used as the texture. For both stages, texture coordinates are generated as eye space coordinates by automatic texture generation, and transformed appropriately to the local coordinate system of the projection. It is done through the use of the texture transformation matrix.

Most of the procedure is similar to the conventional projective texture mapping. But the use of EMBM makes it possible to perform nonlinear warping with graphics hardware. With the conventional projective texture mapping, the eye space coordinates are transformed to texture space coordinates and the x and y components are divided by the z component, in order to obtain the final two-dimensional texture coordinates. In our case, the perspective division in projection space computes (x', y'), the coordinates on the projection plane used by the both stages of multiple texturing. Note that all the process is performed on a per-pixel basis by graphics hardware.

Texture transformation matrix for the first stage (M_0) and the second stage (M_1) are set as given in Equation 4.5. These matrices are multiplied to the eye coordinate vector for each

pixel, so that the following perspective division can give (x', y'). In Equation 4.5, M_{rot} transforms the eye coordinate vectors to the projection space coordinates. This matrix is determined by the location and orientation of the projection cylinder.

Note that a vertex is represented as a row vector and transformation matrix is post-multiplied in DirectX syntax. Minus signs appear at (2,2) elements in both matrices, because textures are addressed from the top in DirectX.

$$M_0 = M_{rot} \cdot \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & -b & 0 & 0 \\ \theta_0 & 0.5 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad M_1 = M_{rot} \cdot \begin{bmatrix} \frac{1}{2X_m} & 0 & 0 & 0 \\ 0 & -\frac{1}{2Y_m} & 0 & 0 \\ 0.5 & 0.5 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.5}$$

This implementation works well with very recent graphics hardware, such as nVIDIA GeForce4 Ti 4600 and ATI Radeon 8500 / 9700, as well as DirectX reference rasterizer. However, it is found that some 3D graphics hardware have problems with this implementation, although their specifications allow these operations. For an instance, nVIDIA GeForce3 does not work correctly when the texture coordinates obtained by projective texture mapping are perturbed in the next stage. ATI Radeon cannot perform projective texture mapping correctly when both stages use projective texture mapping. It seems that some hardware did not implement the entire functionality because this usage is beyond the scope of its original objective, which is bumped environment mapping.

Therefore, we propose another rendering method in the next subsection, which is less efficient but possible with more graphics hardware.

4.4.2 Implementation with multi-pass rendering

Since some graphics hardware does not function correctly when EMBM and projective texturing are used simultaneously, a multi-pass rendering approach is proposed here. It is similar to the single-pass approach in the previous subsection, but it renders in two passes, i.e., rendering takes place twice for the final result. Thus, it performs the explicit local warping onto the projection plane, while the single-pass implementation does it on-the-fly on a per-pixel basis.

In the first pass, texture operation mode and texture maps are set up in the same way as the single-pass implementation. But projective texturing is not set up, and viewing direction is aligned with the direction of projection (DOP). This results in the image, which contains a part of the panorama projected onto the projection plane. In the second rendering pass, this image is used as the texture and the appropriate projective texturing is set up for the final rendering. Figure 4.6 shows the images generated in this process.

(a) Cylindrical map

(b) Projected to Projection Plane

(c) Final Result

Figure 4.6: CPTM with multi-pass rendering

Because we warp the cylindrical map explicitly, the rendering performance with this method is lower than the single-pass method. Moreover, it is required to perform the first pass rendering more than once, since local warping may not cover all views from the projection source. Polygons should be grouped according to their locations, so that each group can be projected with each first-pass rendering result. This problem does not occur with the single-pass method.

4.4.3 Dynamic Range of Bump Map

For the texture formats of bump maps, DirectX provides 8-bit format (D3DFMT_V8U8) and 16-bit format (D3DFMT_V16U16). If 16-bit format is supported by the hardware, the perturbation values in [-1,1] are represented by 16 bits for *u* and *v*, respectively. Therefore, perturbation can be represented in much detail.

However, currently, ATI Radeon 8500 and 9700 are the only hardware with the support of D3DFMT_V16U16. 8-bit bump map can represent the values of only 256 levels in [-1,1] range. In order to utilize this dynamic range effectively, DirectX provides the scaling of bump map texels. This is done by setting D3DTSS_BUMPENVMAT00 and D3DTSS_BUMPENVMAT11 parameters with SetTextureStageState() API. In other words, the perturbation values are scaled up when they are encoded in the bump map, and they are scaled down to the original values, when they are used during rendering. Therefore, 8-bit bump map can be used effectively, if the magnitudes of perturbation values are small. This is the reason why optimal linear coefficients should be found in Section 4.3.

4.5 Special case: Panoramic Rendering from Fixed Viewpoint

When the source of the projection is located at the viewpoint, CPTM works as panorama viewer, which is similar to QuicktimeVR. Since the panorama is projected from the viewpoint and rendered from the same location, we can use any scene geometry as long as it covers the entire screen. Because it is the simplest geometry, cylindrical panorama viewer can

be implemented by rendering a rectangle for the screen with multiple texture configuration described in Section 4.4. This screen rectangle is always attached to the viewing frustum and rotated while the user changes the viewing direction or changes the field of view. In other words, the rectangle is rendered from the viewpoint, while it receives the projection from the viewpoint.

In the cylindrical panorama viewer, the direction of projection (DOP) changes according to the viewing direction. DOP is set as the component of the viewing vector in the normal direction of the projection cylinder. Cylindrical panoramic viewer provides the same rendering capability as QuicktimeVR. But it renders fast even with large resolution, because it takes advantage of hardware rendering. On the other hand, software renderer such as QuicktimeVR degrades image quality when the user changes viewing direction in order to achieve real-time functionality.

4.5.1 Implementation with Per-vertex Projection

Cylindrical panoramic viewer can be implemented using single-pass rendering method described in Section 4.4.1. But there is a better way to implement for this special case. While the implementation in Section 4.4.1 is limited to recent graphics hardware, the rendering method presented in this section can be used with any graphics hardware with EMBM capability. Moreover, the performance is better than the previous method, since it does not require per-pixel projective texture mapping. Therefore, there is no need to use multi-pass

method explained in Section 4.4.2 for panoramic viewer, even for the hardware without the support of single-pass rendering algorithm.

In this implementation, projective texture mapping is applied explicitly for each vertex of the screen rectangle before rendering takes place. This is performed on the vertices by software. Then, the resultant 2D texture coordinates are used for texture mapping. Therefore, there is no need to use the per-pixel projective texture mapping. For the four vertices of the rectangle, transformation matrices in Equation 4.5 are multiplied and perspective division is performed. This gives 2D texture coordinates for the two stages defined on the projection plane, for the vertices.

But correct result cannot be obtained if the screen rectangle is rendered as is. It is because 2D texture coordinates are distorted perspectively. In other words, the mapping is not linear any more inside the rectangle. In order to get correct texture mapping inside the rectangle, the rectangle should also be projected onto the projection plane. If the viewing direction is not horizontal, the rectangle is projected to a trapezoid. Figure 4.7 shows this trapezoid for the viewing direction facing up. Rendering of this trapezoid makes the same results as the method in Section 3.1, since the 2D geometry and texture coordinates are defined on the same two-dimensional domain, which is projection plane.

Figure 4.7: Rendering of trapezoid for per-vertex projection

4.5.2 Zoom In/Out

Zooming-in can be implemented by reducing the field of view for real-time rendering. Similarly, the enlargement of the field of view makes zooming-out. The screen rectangle should be scaled according to the change of the field of view.

4.6 Error Measurement

Errors are introduced by using look-up table, i.e., the bump perturbation map. In order to measure the errors introduced by the approximation, rendering results should be compared with the rendering equation of cylindrical projections. The average per-pixel error is defined as follows.

$$Error = \frac{\sum_{i,j} \sqrt{\left(u_{eq} - u_{approx}\right)^2 + \left(v_{eq} - v_{approx}\right)^2}}{N_x N_y} \tag{4.6}$$

This equation computes the per-pixel average distance between (u,v) texture coordinates determined by the rendering equation and the rendered approximation.

Since rendering is performed entirely within the graphics hardware, it is not possible to extract the exact texture coordinates used by hardware texture mapping. However, rough estimation of the errors can be obtained by rendering the texture encoded with the coordinate values, and reading the rendered results. First, the input cylindrical texture is encoded with its own texture coordinates. Then, it is rendered using panoramic viewer presented in Section 4.5.

After rendering, the frame buffer should be accessed in order to read the texture coordinates used for each pixel. In order to obtain as correct values as possible, D3DFMT_G16R16 texture format is used for the render target. This texture format has red and green channels with 16 bits each. Therefore, the texture map with this format can represent the coordinate values with 2^{-16} accuracy. Because DirectX cannot render directly to the frame buffer with 16 bit-per-channel colors, it is configured to render onto the texture memory with this format. Then, the image in the texture memory is transferred to the lockable region, which can be read by CPU.

For the measurement, cylindrical texture with 256×256 resolution is projected with 90-degree field of view and rendered with 512×512 resolution. Then, the measured average

per-pixel error is about 8.66×10^{-6}. The maximum per-pixel error is 9.9×10^{-4}. This means that the errors are smaller than the texel size even for very high-resolution textures.

For comparison, the same procedure was performed only with linear approximation, without the perturbation of the texture coordinates. In this case, the average error is 0.023 and the maximum error is 0.074.

This does not reduce the errors significantly to increase the bump map resolution beyond 256×256. It is because the bump map is also interpolated linearly, and piecewise linear approximation works fine.

5 Interactive Image Warping

5.1 Introduction

There are many algorithms for geometric warping of two-dimensional images, and they have been used in many applications. Some of them are simple linear transformations, such as scaling, rotation, or shearing. On the other hand, the nonlinear transformations can give more flexible warping from the source images.

With mesh-based warping, the images can be warped based on the given movements of control points. The translations of the control points can be specified by users or they can be evaluated by certain nonlinear warping function. Then all other pixels are computed from the cubic spline interpolation [Wolberg90].

Feature-based image morphing provides seamless image warping sequences between two given images based on the features specified by users. Image morphing provides the smooth transition between two input images. The transition sequence is generated so that the features are moved to their corresponding features.

However, most of the nonlinear image warping algorithms require expensive computations for all pixels. Therefore, it is very difficult to obtain real-time or interactive warping system. Even though cubic spline interpolation is applied, the computational cost prohibits it from being executed interactively.

In this chapter, I propose a method to implement the cubic interpolation with the help of graphics hardware. It is based on the modeling of nonlinear components and programmable pixel shader. It is applied to the image warping process and extended to the feature-based image morphing.

5.2 Basic Idea

Figure 5.1 shows the basic idea of the interactive image warping explained in this chapter. Figure 5.1(a) is the input image to be warped. Since hardware texture mapping is used, this image can be regarded as a texture for mapping. Texture mapping by graphics hardware deals only with triangles, therefore the image is tessellated into triangles shown in Figure 5.1(b).

When the vertices of the triangles have the original texture coordinates after the tessellation, Figure 5.1(b) is the rendering result of the texture mapping. Suppose that the texture coordinates are perturbed by random numbers in (u,v) space. Then, simple texture mapping by graphics hardware will render the triangles like Figure 5.1(c). When the coordinates are given to the vertices arbitrarily, the rendering results look unnatural at the triangle borders and the interior regions. It is because the texture mapping hardware interpolates the texture coordinates linearly inside the triangles, thus the texture coordinates are not C^1 continuous.

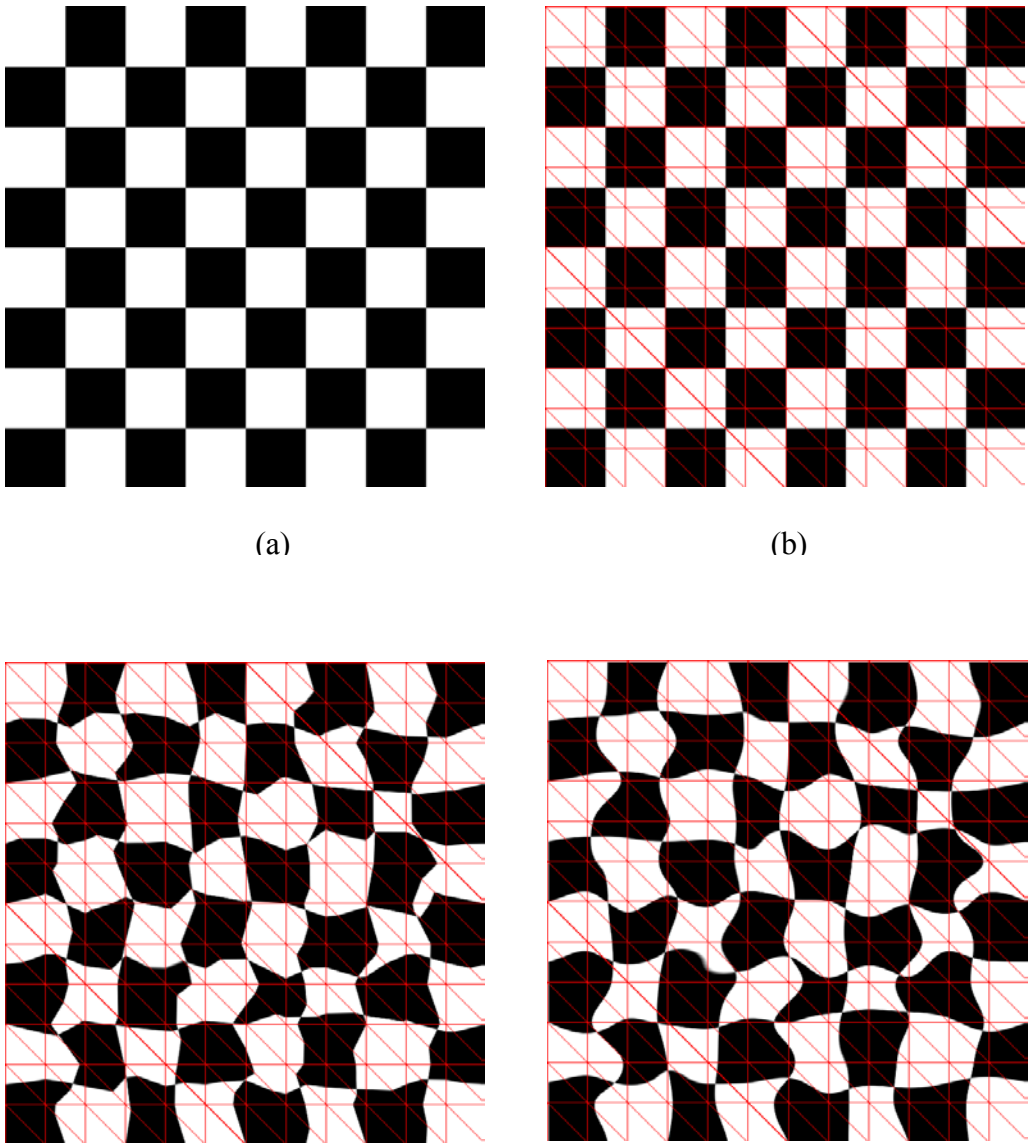


Figure 5.1: Basic idea of interactive image warping

What is more preferable is the warped image shown in Figure 5.1(d). The textures in the interior regions are warped so that the result looks natural, while the texture coordinates assigned to the vertices remain unchanged.

In order to obtain this result, the triangular mesh may be constructed densely. However, the assignments of the coordinates at more vertices can be costly in some applications. Moreover, there will still be some discontinuities unless the subdivision is done at the pixel levels, which also arises problems in the rendering performance.

The approach taken in this dissertation relies on pixel shader, a feature introduced to the graphics hardware recently. Pixel shader can compute the required perturbation amounts of the texture coordinates on a per-pixel basis, so that the warping looks more acceptable.

5.3 Formulation of mesh-based image warping

An input image for warping can be regarded as a texture for mapping. Then, the problem of how to warp an image becomes the problem of how to determine the texture parameterization for new images.

From now on, (x,y) denotes the coordinates in the target image, and (u,v) are the texture coordinates in the source image. x, y, u, and v have the values in [0,1]. Then, the image warp is defined by the following texture mapping function.

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} U(x,y) \\ V(x,y) \end{bmatrix} \tag{5.1}$$

Rather than evaluating the warp for all pixels, the above function can be modeled from the movements of the control points. Control points are located at the rectangular grids in the target image. The parameterization values for the control points can be given by the users or computed from nonlinear mapping function, according to the applications.

Let (x_i, y_j) be the set of the given control points located at the grids of the rectangular lattice. (u_i, v_j) are the texture coordinates for the corresponding (x_i, y_j). Then, the modeling of the warp is to find the appropriate $U(x,y)$ and $V(x,y)$, while they satisfy $(u_i, v_j) = (U(x_i, y_j), V(x_i, y_j))$.

From now on, modeling of $U(x,y)$ is explained, and $V(x,y)$ can be formulated in the same way.

5.3.1 Hermite bicubic surface patch

Given the texture coordinates at the rectangular grids, the texture parameterization within the rectangles can be specified by the interpolation from the boundary information. Hermite bicubic modeling is used in this work, since it gives smooth and reasonable texture parameterization.

Hermite bicubic modeling, as well as many other bicubic methods, is a method of defining parametric surface patches [Foley90]. The three-dimensional positions on the surface can be determined by the parametric equations using the information at the control points. However, in this work, the result of the modeling is texture coordinates, instead of three-dimensional positions. But, the formulation is the same as modeling surface patches.

First, Hermite curves are defined by the positions and tangent vectors at two end points, as in the next equation.

$$
\begin{aligned}
&Q(t) = T \cdot M \cdot G \\
&\textit{where} \\
&T = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \\
&M = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \\
&G = \begin{bmatrix} x(0) & x(1) & \dfrac{\partial}{\partial t}x(0) & \dfrac{\partial}{\partial t}x(1) \end{bmatrix}^T
\end{aligned}
\tag{5.2}
$$

$x(0)$ and $x(1)$ are the end points and their derivatives according to t are the tangent vectors at the points. t is the parameter running from 0 to 1. Then, a curve segment can be defined by using three equations for x, y, and z.

Hermite bicubic patch is the extension of the curve formation, and used for the modeling of the parametric surfaces. It is given in the next equation.

$$Q(s,t) = S \cdot M \cdot G \cdot M^T \cdot T^T$$

where

$$S = \begin{bmatrix} s^3 & s^2 & s & 1 \end{bmatrix}$$

$$T = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix}$$

$$M = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$G = \begin{bmatrix} x(0,0) & x(0,1) & \frac{\partial}{\partial t}x(0,0) & \frac{\partial}{\partial t}x(0,1) \\ x(1,0) & x(1,1) & \frac{\partial}{\partial t}x(1,0) & \frac{\partial}{\partial t}x(1,1) \\ \frac{\partial}{\partial s}x(0,0) & \frac{\partial}{\partial s}x(0,1) & \frac{\partial^2}{\partial s \partial t}x(0,0) & \frac{\partial^2}{\partial s \partial t}x(0,1) \\ \frac{\partial}{\partial s}x(1,0) & \frac{\partial}{\partial s}x(1,1) & \frac{\partial^2}{\partial s \partial t}x(1,0) & \frac{\partial^2}{\partial s \partial t}x(1,1) \end{bmatrix} \qquad (5.3)$$

In this equation, s and t are two parameters running from 0 to 1. Matrix G contains all the geometric information determining the bicubic surface patch, including the positions and the first and second derivatives at four corner points.

5.3.2 Texture coordinate modeling using Hermite bicubic patch

For a given rectangle, texture coordinates can be regarded as functions defined for all the points inside the rectangle. Therefore, the formulations for parametric surfaces patches can be used to model the texture coordinates for the rectangle using the information at the corner points. This section deals with how to model the texture coordinates of a rectangle using a Hermite bicubic patch.

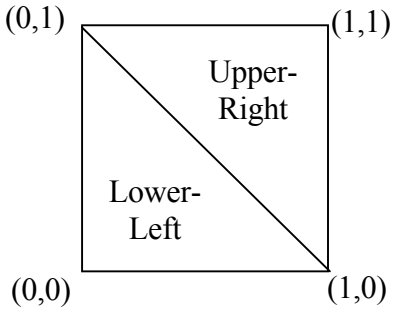


Figure 5.2: From rectangle to two triangles

Among many methods to model parametric patches, Hermite bicubic modeling is used in this work. The reason is that Hermite modeling gives the interpolations, not the approximations, hence it gives smooth and reasonable interpolating functions, while passing through the given values at the corner points.

When a rectangle is rendered by graphics hardware, it should be subdivided into two triangles, because hardware can render only triangles. As shown in Figure 5.2, we render the lower-left and upper-right triangles.

First, the vertices of the triangles are given the texture coordinates, i.e., $x(i,j)$ in Equation 5.3. If the triangles are rendered with simple texture mapping method, it will give the linear approximation of the texture coordinates. The nonlinear components should be added to the linear coordinates, and they are determined by the pixel shader program, which uses the derivative values in Equation 5.3 as inputs.

In order to program the pixel shader, Equation 5.3, the Hermite modeling equation is decomposed into the linear and nonlinear components. The linear approximation for the lower-left triangle is simply the plane equation, which passes the three texture coordinate values given at the vertices.

The plane equation can be represented in the same fashion as Equation 5.3, only with difference *G* matrix, shown in the next equation.

$$G_{L1} = \begin{bmatrix} x(0,0) & x(0,1) & x(0,1)-x(0,0) & x(0,1)-x(0,0) \\ x(1,0) & x(1,0)+x(0,1)-x(0,0) & x(0,1)-x(0,0) & x(0,1)-x(0,0) \\ x(1,0)-x(0,0) & x(1,0)-x(0,0) & 0 & 0 \\ x(1,0)-x(0,0) & x(1,0)-x(0,0) & 0 & 0 \end{bmatrix} \quad (5.4)$$

The nonlinear components are the differences between the original modeling equation and the linear approximation. If the linear component is subtracted from Equation 5.3, the result can be represented in the same way as Equation 5.3, with the different *G* matrix, as follows.

$$G_{N1} = G - G_{L1}$$

$$= \begin{bmatrix} 0 & 0 & \frac{\partial}{\partial t}x(0,0) & \frac{\partial}{\partial t}x(0,1) \\ & & -(x(0,1)-x(0,0)) & -(x(0,1)-x(0,0)) \\ 0 & x(0,0)-x(1,0) & \frac{\partial}{\partial t}x(1,0) & \frac{\partial}{\partial t}x(1,1) \\ & -x(0,1)+x(1,1) & -(x(0,1)-x(0,0)) & -(x(0,1)-x(0,0)) \\ \frac{\partial}{\partial s}x(0,0) & \frac{\partial}{\partial s}x(0,1) & \frac{\partial^2}{\partial s \partial t}x(0,0) & \frac{\partial^2}{\partial s \partial t}x(0,1) \\ -(x(1,0)-x(0,0)) & -(x(1,0)-x(0,0)) & & \\ \frac{\partial}{\partial s}x(1,0) & \frac{\partial}{\partial s}x(1,1) & \frac{\partial^2}{\partial s \partial t}x(1,0) & \frac{\partial^2}{\partial s \partial t}x(1,1) \\ -(x(1,0)-x(0,0)) & -(x(1,0)-x(0,0)) & & \end{bmatrix} \quad (5.5)$$

For upper-right triangle, G matrices of linear and nonlinear components are represented as follows, using the different set of vertices in the rectangle.

$$G_{L2} = \begin{bmatrix} x(1,0)+x(0,1)-x(1,1) & x(0,1) & x(1,1)-x(1,0) & x(1,1)-x(1,0) \\ x(1,0) & x(1,1) & x(1,1)-x(1,0) & x(1,1)-x(1,0) \\ x(1,1)-x(0,1) & x(1,1)-x(0,1) & 0 & 0 \\ x(1,1)-x(0,1) & x(1,1)-x(0,1) & 0 & 0 \end{bmatrix} \quad (5.6)$$

$$G_{N2} = G - G_{L2}$$

$$= \begin{bmatrix} x(0,0) - x(1,0) & 0 & \frac{\partial}{\partial t}x(0,0) & \frac{\partial}{\partial t}x(0,1) \\ -x(0,1) + x(1,1) & & -(x(1,1) - x(1,0)) & -(x(1,1) - x(1,0)) \\ 0 & 0 & \frac{\partial}{\partial t}x(1,0) & \frac{\partial}{\partial t}x(1,1) \\ & & -(x(1,1) - x(1,0)) & -(x(1,1) - x(1,0)) \\ \frac{\partial}{\partial s}x(0,0) & \frac{\partial}{\partial s}x(0,1) & \frac{\partial^2}{\partial s \partial t}x(0,0) & \frac{\partial^2}{\partial s \partial t}x(0,1) \\ -(x(1,1) - x(0,1)) & -(x(1,1) - x(0,1)) & & \\ \frac{\partial}{\partial s}x(1,0) & \frac{\partial}{\partial s}x(1,1) & \frac{\partial^2}{\partial s \partial t}x(1,0) & \frac{\partial^2}{\partial s \partial t}x(1,1) \\ -(x(1,1) - x(0,1)) & -(x(1,1) - x(0,1)) & & \end{bmatrix} \tag{5.7}$$

The (2,2) element in Equation 5.5 and the (1,1) element in Equation 5.7 denote how much the coordinates of two triangles are not coplanar.

Since the linear components are automatically interpolated by graphics hardware, the remaining task is to compute the nonlinear components and add them to the linear texture coordinates. It is performed by the pixel shader program. The pixel shader program utilizes the elements in Equation 5.5 and 5.7 as inputs to calculate the nonlinear components.

5.3.3 Multi-grid texture coordinate modeling using Hermite bicubic patches

This section proposes the main idea for the general nonlinear image warping. It explains how to determine the Hermite coefficients for the patches and use them, based on rectangular grid structure.

The source image is regarded as a texture for the mapping. Then, the target image area is subdivided into many rectangular regions. The vertices of the rectangular mesh are given the appropriate texture coordinates according to the applications. For example, they can be determined by the users' specifications, or complex nonlinear algorithms, such as the feature-based image morphing.

Once the texture coordinates are assigned to the vertices, they are used for regular texture mapping of each rectangle, i.e., two triangles, for the linear approximation. Then, pixel shader program computes the amounts of nonlinear perturbation.

In order to provide the inputs used by pixel shader program, we need the derivative values in Equation 5.5 and 5.7. Catmull-Rom spline is a simple, but effective method to generate a smooth spline, which passes the given control points. With Catmull-Rom spline, the first derivative at point i is set as half the vector connecting two adjacent points, i-1 and i+1 [Foley90].

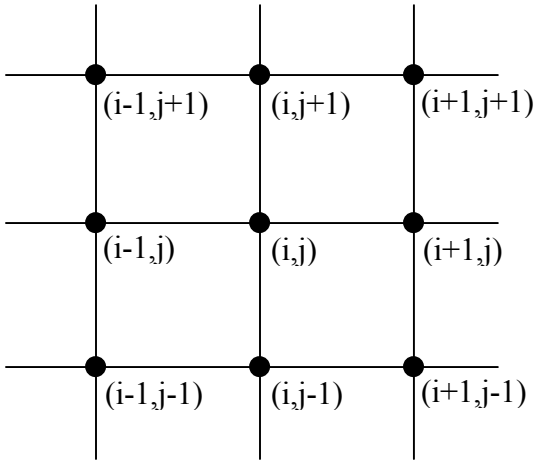


Figure 5.3: Grids for Catmull-Rom formulation

Suppose that we want to determine the derivative values at grid vertex (*i,j*) in Figure 5.3, where *i* and *j* are the indices for the grid vertices. Let *s* and *t* be the parameters in the directions of *i* and *j*, respectively and *u* and *v* be the resultant texture coordinates. *s* and *t* have the size of 1 for each grid. Then, the derivatives for *u* can be computed as Equation 5.8 using Catmull-Rom spline modeling.

$$\begin{aligned}
\frac{\partial u}{\partial s}(i,j) &= \frac{1}{2}\left[u(i+1,j)-u(i-1,j)\right] \\
\frac{\partial u}{\partial t}(i,j) &= \frac{1}{2}\left[u(i,j+1)-u(i,j-1)\right] \\
\frac{\partial^2 u}{\partial s \partial t}(i,j) &= \frac{1}{2}\left[\frac{1}{2}\left[u(i+1,j+1)-u(i-1,j+1)\right]-\frac{1}{2}\left[u(i+1,j-1)-u(i-1,j-1)\right]\right] \\
&= \frac{1}{4}\left[u(i+1,j+1)-u(i-1,j+1)-u(i+1,j-1)+u(i-1,j-1)\right]
\end{aligned} \tag{5.8}$$

The derivatives for v can be determined with the above equation except that they use v instead of u. For the boundary grids of the image, where two adjacent grids are not available, the missing u and v can be assumed to have extrapolated values.

5.3.4 Level-of-detail texture coordinate modeling

In some cases, it is time consuming to compute the nonlinear texture coordinates at grid vertices, while hardware-assisted texture mapping is rather fast. Therefore, it is required to reduce the number of the grid vertices as much as possible, without much quality degradation.

In this section, the algorithm to model the texture coordinates using a multi-level rectangular grid structure. Figure 5.4 shows an example of this structure. The level of refinement can be determined differently by the applications. In feature-based image morphing, for instance, texture coordinates have more possibility of abrupt changes in the neighborhood of the features. Therefore, more refinement is needed near the specified features.

Texture coordinates should be continuous functions in order to avoid cracks of the warped textures. So, texture coordinates should be modeled with caution along the change of the levels.

After the grid structure is determined, the vertices are classified into two groups. For one group, the vertices have all four links in the grid structure. The other group contains T-vertices, which have only three links due to the level differences.

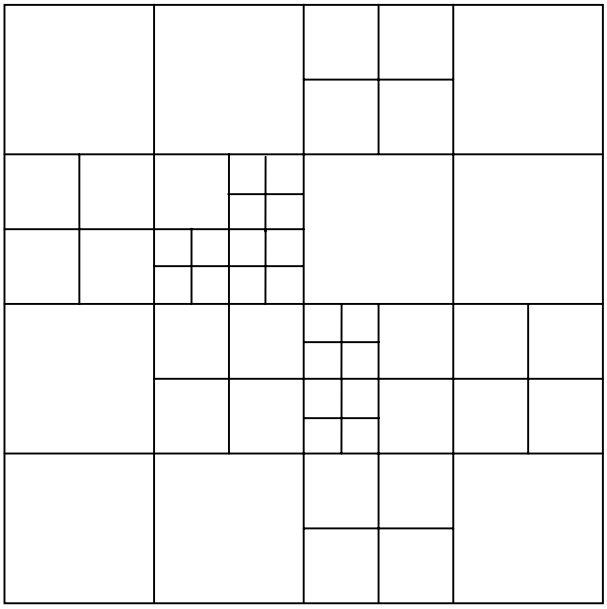


Figure 5.4: Subdivision of rectangular mesh

For the vertices in the first group, the evaluations of application-specific nonlinear function are performed to determine the texture coordinates. On the other hand, we cannot apply the nonlinear function to the T-vertices, because the adjacent regions in the coarse level are modeled by Hermite method. It would cause cracks in the warped image. In order to handle these situations, the texture coordinates for the finer level rectangles are determined by evaluating the Hermite model equations for the adjacent coarser level rectangles, instead of the calculation of the nonlinear functions.

5.3.5 Full Hermite modeling without the decomposition

Since DirectX 9 and pixel shader 2.0 support floating point operations, Equation 5.3 can be implemented without decomposing into the linear and nonlinear components. With this scheme, the results of the pixel shader program are used as the texture coordinates directly, instead of adding them to the linear components.

The rendering results will be similar with a little more use of registers and pixel shader instructions. However, the decomposition method is used in this dissertation because of the following reasons.

First, the graphics hardware provides high quality linear interpolation anyway, and there is no reason why we do not use it, even though the penalty of not using it is not much. Second, although pixel shader 2.0 supports the use of floating point numbers in its registers and arithmetic operation, the precisions are not IEEE standard yet. However, the linear interpolation by graphics hardware is more precise. Therefore, the smaller the magnitudes of pixel shader computations, the more correct the final texture coordinates.

5.4 Implementation of the pixel shader program

Pixel shader was introduced by recent DirectX API specification, and can be executed on the rather new graphics hardware. It is an assembly program that is executed for each pixel to be rendered. After it is loaded to graphics hardware, it is executed entirely within the

hardware without using CPU. Since the instruction set consists of many useful instructions such as vector and texture operations, its performance is much better than CPU for some specific tasks.

As the API version and the graphics hardware evolve, the capabilities of pixel shader are being enhanced. Programs with larger number of instruction can be used and more useful instructions are added. Before the release of pixel shader 2.0, the arithmetic instructions were implemented as integer arithmetic. But, DirectX 9 supports pixel shader 2.0, which uses floating-point arithmetic operations. DirectX 9 is under beta test as this dissertation is being written and there is only one graphics hardware in the market, which supports the entire functionalities of DirectX 9. But it will be used widely very soon.

Since Hermite modeling requires precise calculations of texture coordinates, pixel shader 2.0 is used in this dissertation. Moreover, the previous versions of pixel shader allow only small number of instructions in one pixel shader program, which prohibits complex computations.

The rectangles comprising the mesh are rendered using graphics hardware. The given texture parameterization at the control points are set as the texture coordinates for the rectangle corners. Then, programmable pixel shader perturbs the texture coordinates inside the rectangles. Pixel shader can receive multiple texture coordinate sets and use arithmetic operations to calculate the amounts of perturbations.

Appendix A shows the implemented pixel shader program, which is executed for each pixel. Since pixel shader can use multiple texture coordinates, two texture coordinates are given to the shader program. One is the original texture coordinates, which are the linear approximations. As second texture coordinates, *s* and *t* parameters are used. When *s* and *t* have 0 or 1 at the vertices of the triangles, the interior pixels have the interpolated *s* and *t* values.

All the derivative values in Equation 5.5 and 5.7 are constant for any given triangle. Therefore, they are given as constant register values. Then, what the pixel shader does is to compute the perturbation amounts of the texture coordinates, and add them to the original texture coordinates.

For efficient calculations, some cubic factors of Equation 5.3 are pre-computed as look up tables. In Equation 5.3, $S \cdot M$ is a 4D row vector and $M^T \cdot T^T$ is a 4D column vector. They are cubic functions of *s* or *t.* We use Equation 5.3 with *G* matrices given in Equation 5.5 and 5.7, since only nonlinear components are calculated. From this equation, all nonzero components in the *G* matrices are multiplied by one component in $S \cdot M$ and another component in $M^T \cdot T^T$ to contribute to the total summation. In other words, the multiplication of two cubic functions in *s* and *t*, each, can be assigned to one nonzero component in the *G* matrices. Because *G* matrices in Equation 5.5 and 5.7 have 13 nonzero components, 13 functions, which are cubic for both *s* and *t*, are evaluated. They are computed for *s* and *t* values in [0,1] range, and stored as two-dimensional texture maps for further look-ups. Because

texture maps can have up to four components as (R,G,B,A) format, four functions are gathered to make one texture map.

Because DirectX 9 and pixel shader 2.0 supports texture maps of floating point numbers, the look-up textures do not incur any precision problem through the arithmetic operations. For s and t values, which are not exactly at the sample points, the graphics hardware will interpolate nearby values in the look up table automatically.

5.5 Interactive Image Warping

The texture coordinates for the control points in the mesh can be moved easily by the users' inputs. Since the algorithm presented in this chapter provides real-time rendering of the warped images, the users can control the warping while looking at the instant feedbacks.

5.6 Interactive Feature-Based Image Morphing

5.6.1 Feature-based image morphing

Two-dimensional image morphing is being used in many applications, such as movies or commercials. Beier proposed the feature-based image morphing algorithm [Beier92]. It generates smooth transition between two source images with the given line segment features.

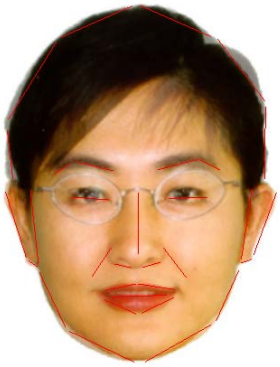
Figure 5.5 shows the procedure of image morphing. Figure 5.5(a) shows the left (t =0.0) and right (t =1.0) images. The line segments are assigned so that the meaningful features of the two images match.

For given parameter t, all the line segment features are interpolated linearly. Then, the source images are warped appropriately according to the interpolated features. Figure 5.5(b) shows the warped images at t=0.5. Note that the features are identical in the two images in Figure 5.5(b). Finally, the warped images are blended with the parameter t, to generate in-between morphed image in Figure 5.5(c).

The key of the morphing is how to determine the corresponding pixel in the two input images, for each pixel in the intermediate images in Figure 5.5(b).

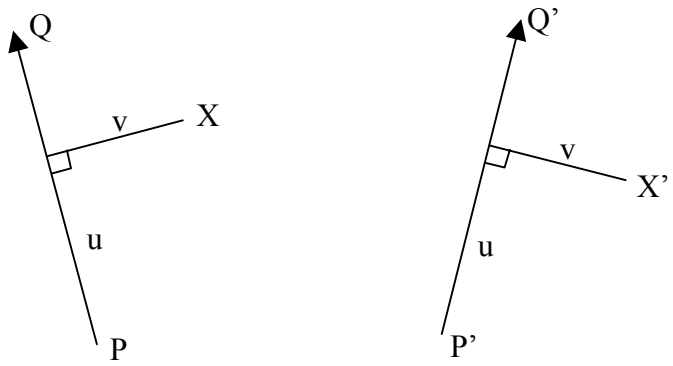
(a) Source images

(b) Warped with the interpolated features



(c) Morphed image (t=0.5)

Figure 5.5: Procedure for image morphing



(a) Destination image (b) Source image

Figure 5.6: Finding corresponding pixel in the source image

First, the pixel is parameterized according to the interpolated features. In Figure 5.6(a), X is the given pixel and P and Q are the end points of a feature. Then, (u,v) is parameterized so that it represents the location of X, relative to the interpolated feature, PQ. It is computed with the following equation.

$$u = \frac{(X-P)\cdot(Q-P)}{\|Q-P\|^2}$$
$$v = \frac{(X-P)\cdot Perpendicular(Q-P)}{\|Q-P\|} \tag{5.9}$$

Then, the corresponding location, X' in the source image is computed with the parameterization (u,v) and the original feature $P'Q'$ as follows. In this equation, *Perpendicular*() is a function which rotates a 2D vector by 90 degrees.

$$X' = P' + u \cdot (Q' - P') + \frac{v \cdot Perpendicular(Q' - P')}{\|Q' - P'\|} \tag{5.10}$$

Since a lot of features are assigned to an image, the final pixel in the source images are determined by the weighted average of the results of Equation 5.10 for all the features. The weights for the features are defined as Equation 5.11. In this equation, *length* denotes the magnitude of each feature and *dist* means the distance from the pixel to the line segment feature. *a*, *b*, and *p* are user-controlled parameters to control the quality of the morphing.

$$wegiht = \left(\frac{length^p}{(a + dist)} \right)^b \tag{5.11}$$

The disadvantage of this morphing scheme is that it computes the morphed image entirely by software. Moreover, the equations used for each pixel are complex, therefore it is very difficult to be implemented as real-time applications.

5.6.2 Interactive image morphing

In order to implement the feature-based image morphing for real-time or interactive applications, the advanced features of the hardware-accelerated texture mapping are used in this section.

It is similar to texture mapping to find the corresponding pixels in the source image with the feature-based morphing techniques in the previous subsection. Therefore, texture

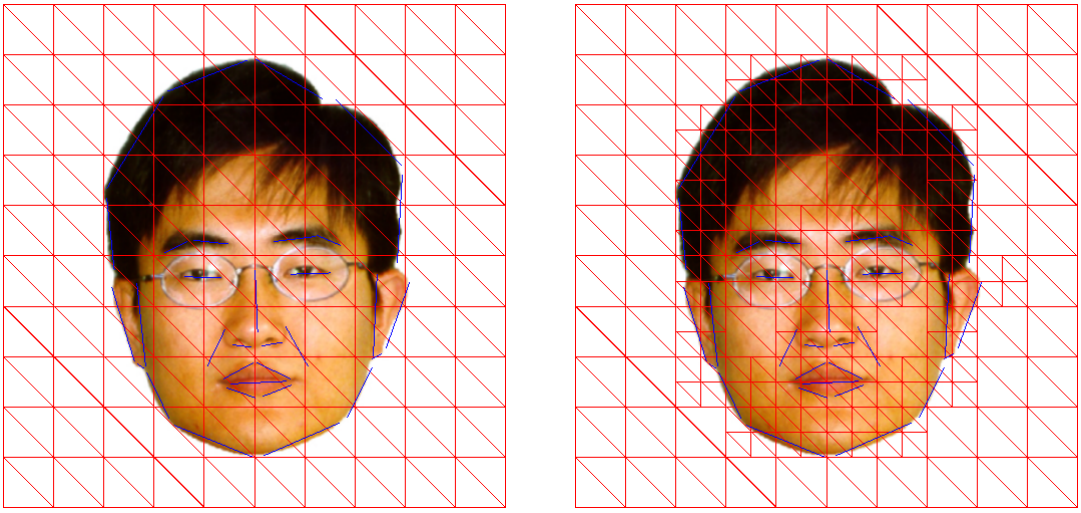
mapping can replace the procedure. However, simple texture mapping cannot be used, because the texture coordinates are interpolated linearly inside the polygons with hardware texture mapping, while image morphing requires nonlinear complex evaluations of texture coordinates.

The main idea is that the equations of image morphing are evaluated only at selected points, while all other pixels have their texture coordinates interpolated appropriately. For smooth interpolation, the techniques given in Section 5.3 are used. Generally, this gives acceptable results of image morphing.

First, the destination image is subdivided into a rectangular mesh as shown in Figure 5.7(a). The equations of image morphing are evaluated only at the vertices of this mesh. Then, the interior regions of the rectangles are rendered with nonlinear texture mapping techniques with Hermite modeling proposed in Section 5.3. The derivative values for pixel shader are computed from the texture coordinates at the vertices using Catmull-Rom method.

This scheme is similar to the 2-pass mesh warping presented in [Wolberg90]. But it is implemented by software, while our method uses hardware-accelerated texture mapping for real-time rendering.

The vertices of the rectangular mesh do not usually exist at the locations of features. Since the corresponding features should match after the morphing, it generates better results to sample more densely near the features. For this purpose, level-of-detail subdivision is used as proposed in Section 5.3.4.



(a) Initial mesh (b) After subdivision

Figure 5.7: Subdivision near the features

For the neighborhoods of the features through the morphing, the rectangles are subdivided further and more evaluations of morphing equations take place. Figure 5.7(b) shows an example of the resultant mesh. As described in Section 5.3, the derivative values should be determined carefully along the changes of the subdivision levels, in order to avoid unnatural textures through the image morphing.

5.6.3 Implementation of interactive morphing system

Interactive morphing system was implemented as shown in Figure 5.8.

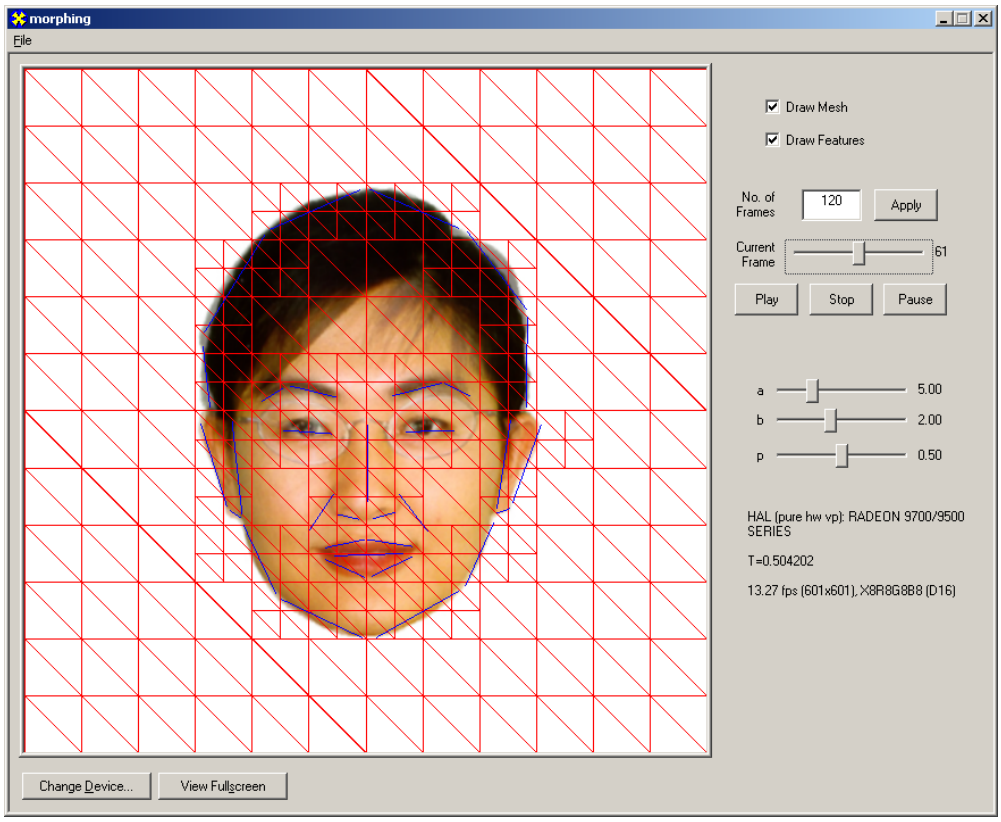


Figure 5.8: Interactive image morphing system

It is equipped with a lot of options to control the morphing sequence. The information on the input image files and the line segment features is provided from data file using File-Open menu. Rectangular mesh and features can be shown or not, through the use of toggle check boxes.

Users can assign the number of frames to be generated for the morph. Morphing can be played, stopped, or paused, like the controls of the streaming videos. Users can move the slide

bars to move forward and backward through the morph. In addition, a, b, and p parameters in Equation 5.11 can be assigned by using slider bar controls. Current blending parameter and rendering statistics are also displayed.

Since all the controls make immediate changes on the rendered images, this system can be used as morph authoring software. With slow software implementations of feature-based morphing, it is very difficult to find the control parameters to satisfy the users' intentions.

6 Results

Experimental results are shown in this chapter for the applications presented in Chapter 4 and 5. All the implementations of the ideas proposed in this dissertation are based on DirectX API. Similar implementations will also be possible with other real-time rendering API, such as the current or future version of OpenGL.

The implementation is based on DirectX 8.1, for the results in Section 6.1. For the image warping results in Section 6.2, DirectX 9 RC0 (Release Candidate 0) is used, since the public version of DirectX 9 is not available yet, at the time of writing this dissertation.

6.1 Cylindrical projective texture mapping

Figure 6.2 shows the results of the projection from the cylindrical panorama to a simple geometry. The cylindrical panorama is projected from the center of the cube geometry. Each face of the cube receives the projection with its own direction of projection (DOP). These results are a part of the animation, where the projection source is rotating and the viewpoint is moving. It is shown that cylindrical panorama is projected to the planes correctly.

Athlon XP 1700+ PC is used with nVIDIA geForce4 Ti 4600 graphics card. When single pass rendering algorithm is used with this hardware, the rendering speed is about 430 frames per second for 640x480 resolution and about 174 frames per second for 1024x768 resolution.

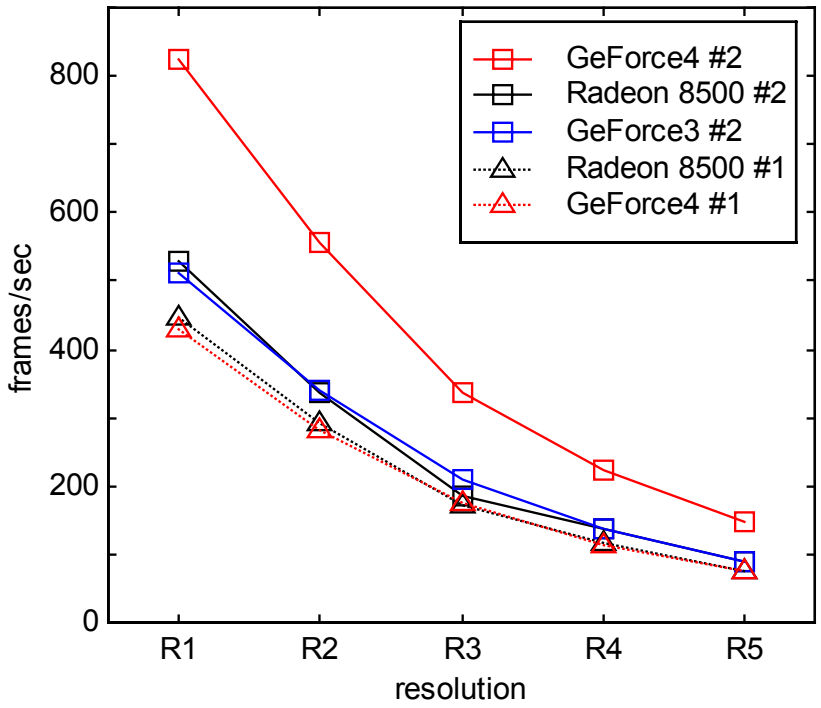
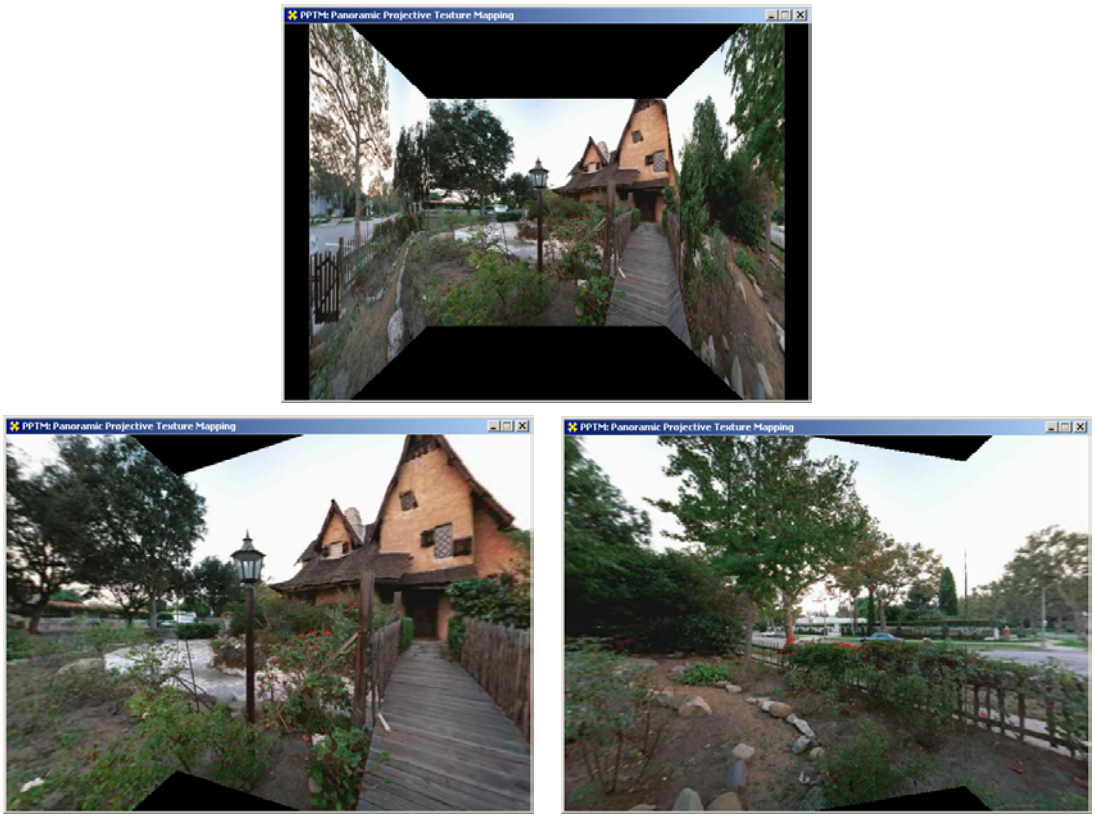


Figure 6.1: Rendering performance of cylindrical viewer

Figure 6.3 shows the rendering results with cylindrical panoramic viewer, which is a special case of cylindrical projective texture mapping. With the implemented panoramic viewer, the user can look around with horizontal and vertical rotations. Zooming in and out is also supported. Rendering performance is shown in Figure 6.1. In this figure, #1 means the implementation in Section 4.4.1 and #2 means the method in Section 4.5.1, which is per-vertex projection. R1 through R5 mean the resolutions of 640x480, 800x600, 1024x768, 1280x960, and 1600x1200, respectively. GeForce3, geForce4 Ti 4600, and Radeon 8500 were used for the test.

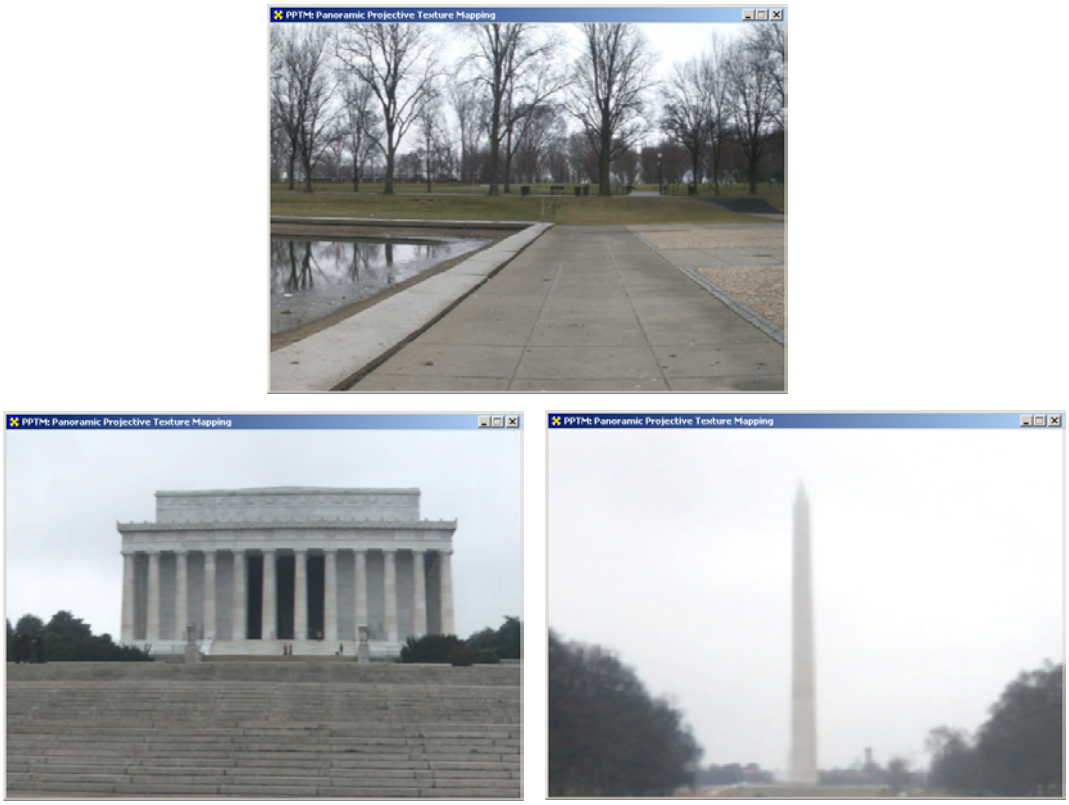
(a) Cylindrical panorama



(b) Rendering results

Figure 6.2: Rendering result of cylindrical panorama projection

(a) Cylindrical panorama



(b) Rendering Results

Figure 6.3: Rendering results of Cylindrical Panorama Viewer

6.2 Interactive image warping

Athlon XP 1700+ PC with ATI Radeon 9700 Pro was used for the test in this section.

Figure 6.4 shows the image warping results, when the texture coordinates are translated by random numbers in 2D coordinate space. The images are rendered with transition parameter, *t*, from 0 to 1. In-between frames are rendered with the fractional movements of the texture coordinates. The result shows that the source image is warped naturally, based on new texture coordinates assigned to the vertices. This example is executed at 57 frames per second, for 600×600 image resolution.

Figure 6.5 shows the result of feature based image morphing. Two source images and 32 line segment features are given as the inputs. The image is subdivided into 12×12 rectangles. It requires 169 evaluations of Equation 5.10, for all the features, including the boundary vertices. For the neighborhoods of the features, the rectangles are subdivided into one deeper level. This increases the number of the evaluations to 232. With this setup, morphed frames could be rendered at 27 frames per second, for 600×600 image resolution.

With the algorithm in [Beier92], the evaluations should take place for all pixels to render. With this algorithm, it takes about 30 seconds to render one frame by software implementation, with the above hardware configuration. Therefore, rendering performance is increased by a factor of about 10^3 with the hardware-assisted method, while the rendering quality does not drop much with the Hermite interpolation method using pixel shader.

Rendering performance is determined by the evaluations of morphing equations, since the cost of hardware-assisted texture mapping is rather fast compared with the required software computations. It means that the adaptive subdivision or the number of the features can be adjusted according to the power of CPU, in order to obtain the required frame rates.

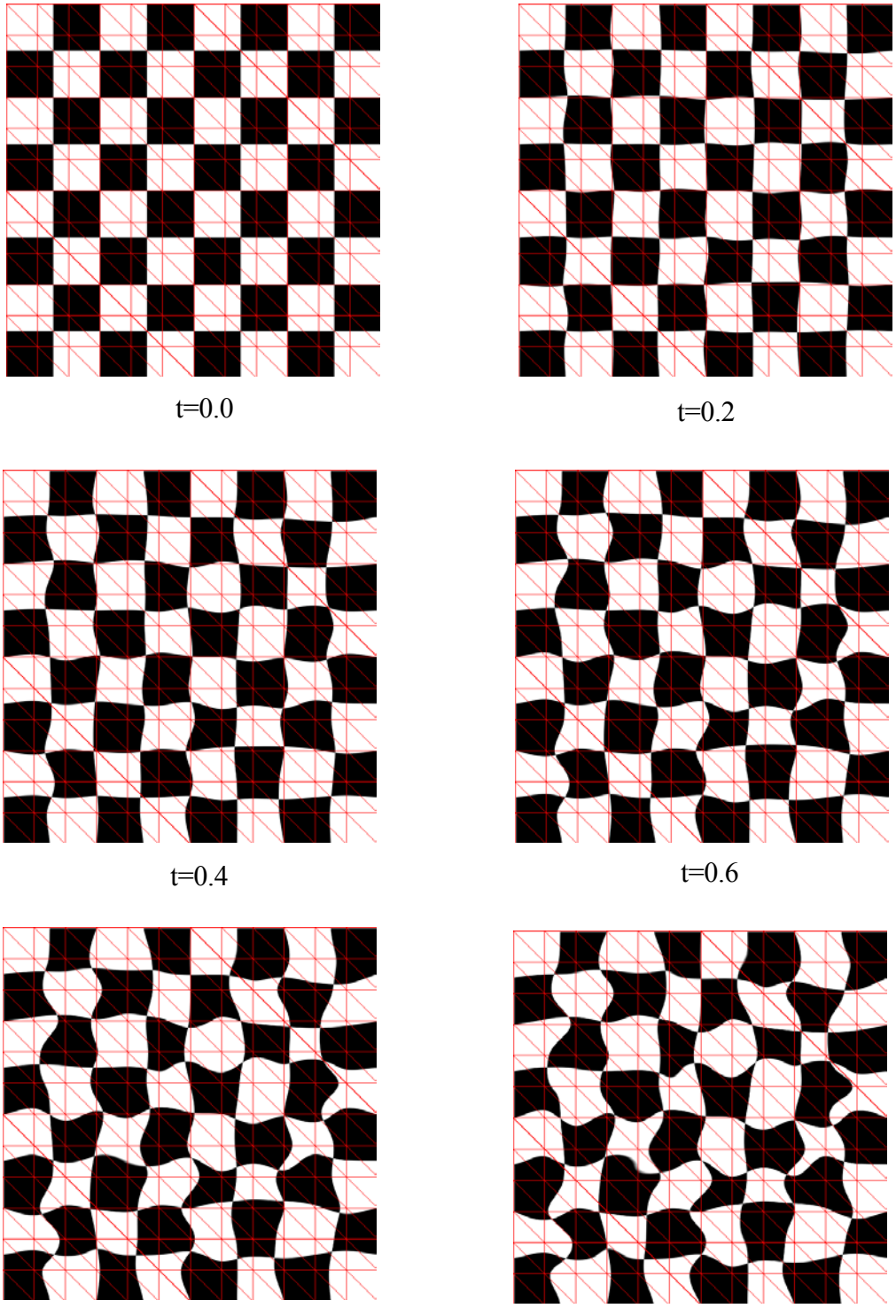


Figure 6.4: Image warping by random movements

t=0.0

t=0.1

t=0.2

t=0.3

t=0.4

t=0.5

t=0.6

t=0.7

t=0.8

t=0.9

t=1.0

Figure 6.5: Morphing example

7 Conclusion and Future Work

7.1 Conclusion and Original Contributions

The techniques for hardware-assisted nonlinear texture mapping were presented in this dissertation. The texture parameterizations are perturbed inside the polygons, for nonlinear texture mapping.

Two methods were proposed for the hardware-assisted perturbation. First, environment mapped bump mapping (EMBM) hardware can be used under multiple texturing configurations. Second, programmable pixel shader can be used. This provides more flexible control, but it is not widely available as EMBM.

As the applications of nonlinear texture mapping, cylindrical projective texture mapping and interactive image warping techniques were given. In the first application, cylindrical panorama can be projected directly onto the scene geometry. QuicktimeVR-like panoramic viewer is a special case of the cylindrical projection. In image warping application, nonlinear mesh-based warping can be accelerated using cubic interpolation, which is programmed as pixel shader. The extension to the feature-based image morphing proposes a valuable application of nonlinear texture mapping.

The contributions made by this dissertation are as follows. First, it proposes methods to analyze nonlinear texture mapping requirements and implement them using the advanced

features of the graphics hardware. Second, it makes it possible to use hardware assistance for real-time rendering in the applications proposed by this dissertation.

7.2 Future Work

The concept of cylindrical projective texture mapping could be extended to more complex forms of panorama. Concentric mosaics are the extension of simple cylindrical panorama, but it enables the movement of the viewpoint through the scene [Shum99]. The projection of concentric mosaics by graphics hardware will give a more realistic VR navigation based on the photographs.

Interactive image warping based on the irregular mesh could be used for many applications. The quality of the feature-based image morphing will be enhanced, since the discontinuities made by the features can be placed exactly at the edges of the mesh. It could also be used for texture mapping of polygonal mesh objects, so that the textures exhibit better appearances without increasing the number of polygons. Texture mapping for progressive meshes [Hoppe96] could be another application of this method.

Pixel shader can be regarded as a SIMD parallel computing unit. It can be used for the non-rendering purposes in some applications for better performance. A simple ray tracer was also implemented using pixel shader [Purcell02]. It would be possible to develop real-time ray tracer or radiosity renderers with this approach.

Pixel shader can also be used for the simulations on the rectangular grids, since it has parallel computing units in the two-dimensional lattice. FEM or cloth simulation could be the applications in this area.

References

[Beier92] Beier, T. and S. Neely, "Feature-based image metamorphosis," *Proceedings of SIGGRAPH 78*, pp. 35-42, 1992.

[Blinn76] Blinn, J.F and M. E. Newell, "Texture and Reflection in Computer Generated Images," *Communations of the ACM,* Volume 19, Number 10, October 1986, pp. 542-547.

[Blinn78] Blinn, J.F., "Simulation of Wrinkled Surfaces," *Proceedings of SIGGRAPH 78*, pp. 286-292, 1978.

[Buehler01] Buehler, C., M. Bosse, L. McMillan, S. Gortler, and M. Cohen, "Unstructured Lumigraph Rendering," *Proceedings of SIGGRAPH 2001*, pp. 425-432, 2001.

[Blythe00] Blythe, D., et al., "Advanced Graphics Programming Techniques Using OpenGL", *SIGGRAPH 2000 Course Notes.*

[Chen95] Chen, S. E., "Quicktime VR - An Image-Based Approach to Virtual Environment Navigation," *Proceedings of SIGGRAPH 95*, pp. 29-38, 1995.

[Debevec96] Debevec, P.E., C.J. Taylor, and J. Malik, "Modeling and Rendering Architecture from Photographs," *Proceedings of SIGGRAPH 96*, pp. 11-20, August 1996.

[Debevec98] Debevec, P.E., G. Borshukov, and Y. Yu., "Efficient View-Dependent Image-Based Rendering with Projective Texture-Mapping," *The 9th Eurographics Rendering Workshop*, Vienna, Austria, June 1998.

[DirectX8] Microsoft, DirectX 8.1 Programmer's Manual.

[DirectX9] Microsoft, DirectX 9 Programmer's Manual.

[Engel02] Engel, W. F., *Direct3D ShaderX: vertex and pixel shader tips and tricks*, Wordware, 2002.

[Faux79] Faux, I. D., and M. J. Pratt, *Computational Geometry for Design and Manufacture*, Wiley, New York, 1979.

[Foley90] Foley, J. D., A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practices*, Addison-Wesley, 1990.

[Greene86] Greene, N., "Environment Mapping and Other Applications of World Projections," *IEEE Computer Graphics and Applications*, Volume 6, Number 11, November 1986, pp.21-29.

[Heckbert86] Heckbert, S., "Survey of Texture Mapping," *IEEE Computer Graphics and Applications*, Volume 6, Number 11, November 1986, pp.56-67.

[Hoppe96] Hoppe, H., "Progressive Meshes," *Proceedings of SIGGRAPH 96*, pp. 99-108, 1996.

[Kim02] Kim, Dongho and James K. Hahn, "Projective Texture Mapping with Full Panorama," *Computer Graphics Forum*, Volume 21, Number 3, pp.421-430, September 2002 (also appeared at Eurographics 2002).

[Kim03] Kim, Dongho and James K. Hahn, "Hardware-Assisted Rendering of Cylindrical Panorama," *to appear in Journal of Graphics Tools*, Spring 2003.

[Levoy88] Levoy, M., "Display of Surfaces from Volume Data," *IEEE Computer Graphics and Applications*, Volume 8, Number 3, pp.29-37, 1988.

[Lindholm01] Lindholm, E., M. J. Kilgard, and H. Moreton, "A User-Programmable Vertex Engine," *Proceedings of SIGGRAPH 2001*, pp.149-158, 2001.

[Milliron02] Milliron, T., R. J. Jensen, R. Barzel, and A. Finkelstein, "A Framework for Geometric Warps and Deformations," ACM Transactions on Graphics, Volume 21, Number 1, January 2002, pp. 20-51.

[PanoGuide] http://www.panoguide.com/

[Peachey85] Peachey, D. R., "Solid Texturing of Complex Surfaces," *Proceedings of SIGGRAPH 85*, pp.279-286, 1985.

[Perlin85] Perlin K., "An Image Synthesizer," *Proceedings of SIGGRAPH 85*, pp.287-296, 1985.

[Press88] Press, W. H., B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C*, Cambridge University Press, 1988.

[Proudfoot01] Proudfoot, K., W. R. Mark, S. Tzvetkov, and P. Hanrahan, "A Real-Time Procedural Shading System for Programmable Graphics Hardware," *Proceedings of SIGGRAPH 2001*, pp.159-170, 2001.

[Purcell02] Purcell, T. J., I. Buck, W. R. Mark, and P. Hanrahan, "Ray Tracing on Programmable Graphics Hardware," *Proceedings of SIGGRAPH 2002*, pp. 703-712, 2002.

[Shum99] Shum, H.-Y. and L.-W. He, "Rendering with Concentric Mosaics," *Proceedings of SIGGRAPH 99*, pp.299-306, 1999.

[Smythe90] Smythe D. B., "A Two-Pass Mesh Warping Algorithm for Object Transformation ad Image Interpolation," *ILM Technical Memo* #1030, Lucasfilm Ltd., 1990.

[Watt92] Watt, A. and M. Watt, Advanced Animation and Rendering Techniques: Theory and Practice, Addison-Wesley, 1992.

[Weinhauss99] Weinhaus, F. M. and R. N. Devich, "Photogrammetric Texture Mapping onto Planar Polygons", *Graphical Models and Image Processing*, Volume 61, Number 2, 63-83, 1999.

[Wolberg90] Wolberg, *Digital Image Warping*, IEEE Society Press, 1990.

[Woo99] Woo, M., J. Neider, T. Davis, and D. Shreiner, OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2, 3rd Edition, Addison-Wesley, 1999.

Appendix A Pixel Shader Implementation for Chap 5

```
// c0~c5: derivative values
// c6.rg: the amount of non-planarity (two values for u and v)
// c6.a: alpha value for frame buffer alpha blending

// texture 0 (s0): regular texture
// texture 1 (s1): (R,G,B,A) texture for cubic function of s and t
// texture 2 (s2): (R,G,B,A) texture for cubic function of s and t
// texture 3 (s3): (R,G,B,A) texture for cubic function of s and t
// texture 4 (s4): (R) texture for cubic function of s and t

// t0: texcoord for the whole image
// t1: texcoord for each quad

ps.2.0

dcl t0.rg
dcl t1.rg

dcl_2d s0
dcl_2d s1
dcl_2d s2
dcl_2d s3
dcl_2d s4

texld r1, t1, s1  ; cubic parameters
texld r2, t1, s2  ; cubic parameters
texld r3, t1, s3  ; cubic parameters (off-planar component)
texld r4, t1, s4  ; cubic parameters

// planar components
dp4 r5.x, r1, c0
dp4 r6.x, r2, c1
dp4 r7.x, r4, c5
dp4 r5.y, r1, c2
dp4 r6.y, r2, c3
dp4 r7.y, r4, c6

add r5.xy, r5, r6
add r5.xy, r5, r7

// off-planar component
mad r5.xy, r3.r, c4, r5

// perturb texture coordinates
add r0.xy, r5, t0
```

```
texld r0, r0, s0
mov r0.a, c4.a

mov oC0, r0
```