Procedural Texture Mapping on FPGAs

by

Andy G. Ye

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science in
the Graduate Department of Electrical and Computer Engineering,
University of Toronto

Procedural Texture Mapping on FPGAs

Andy G. Ye

Master of Applied Science, 1999

the Graduate Department of Electrical and Computer Engineering

University of Toronto

Abstract

Procedural textures can be effectively used to enhance the visual realism of computer rendered images. Procedural textures can provide higher realism for 3-D objects than traditional hardware texture mapping methods which use memory to store 2-D texture images. This thesis investigates a new method of hardware texture mapping in which texture images are synthesized using FPGAs. This method is very efficient for texture mapping procedural textures of more than two input variables. By synthesizing these textures on the fly, the large amount of memory required to store their multidimensional texture images is eliminated, making texture mapping of 3-D textures and parameterized textures feasible in hardware. This thesis shows that using FPGAs, procedural textures can be synthesized at high speed, with a small hardware cost. Data on the performance and the hardware cost of synthesizing procedural textures in FPGAs are presented. This thesis also presents the FPGA implementations of six Perlin noise based 3-D procedural textures.

Acknowledgments

I would like to thank my supervisor, Professor David Lewis, for all the guidance and technical advice. His guidance has made this two years a remarkable learning experience for me.

I would like to thank Dave Galloway for laying the ground work by designing a 2-D texture mapping system on the TM-2. I also would like to thank him for all the TM-2 software and hardware support that he has provided.

I would like to thank Marcus van Ierssel for designing the VGA interface card and maintaining and constantly improving the TM-2 hardware.

I also would like to thank Michiel van de Panne, James Stewart, Jonathan Rose, Vaughn Betz, Qiang Wang, and Yaska Sanka for their technical help and advice.

I am tremendously grateful to my parents for their support.

Finally I would like to thank all the people who have worked in LP392 in the past two years for creating such a great research and learning environment.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In many computer graphic applications, polygon meshes are used to model geometrical surfaces. Texture mapping increases the level of surface detail of polygon meshes by mapping two-dimensional texture images on to the meshes. In common graphic cards, the 2-D texture images are pre-computed and stored in memory on the cards. Procedural texture mapping extends the concept of texture mapping by determining the surface coloring of polygon meshes using computer algorithms. These procedural texture algorithms typically model the structures of materials like concrete, wood and marble. They can be defined in 3-D space and be parameterized using input variables defining additional attributes other than the texture coordinates. The main advantages of procedural textures over conventional 2-D textures are much enhanced visual realism, reduced storage requirement, enhanced ease of use and user controllability.

Procedural texture mapping has become an important method of generating visually realistic images in many graphic applications. The computation, however, is often time-consuming. Procedural texture algorithms, when executed in software, often cannot achieve the real time performance demanded by many computer animation applications. While 2-D textures can be stored in RAM, 3-D textures require excessive memory. There are no efficient methods of performing texture mapping using three-dimensional or parameterized procedural textures using fixed hardware. The primary reason for this is the variety of procedural textures, which makes it difficult to design a single, efficient hardwired implementation for synthesizing all textures. Other reasons include the complexity of many procedural texture algorithms, and the ongoing development of new algorithms. A hardwired accelerator not only would be difficult to design to support all the

existing procedural texture algorithms, but also difficult to modify to support new algorithms.

This thesis investigates a new approach to synthesizing procedural textures in hardware in which FPGA hardware is used to provide high performance implementations of procedural texture algorithms. The primary technique used is to compile the procedural algorithms into hardware structures that can be programmed into FPGAs. This approach is more memory efficient than storing pre-generated textures in memory, since only the algorithms are stored. The use of FPGAs also results in the ability to exploit the parallelism presented in each individual algorithm.

A procedural texture generator was designed using the FPGAs. It is flexible enough to synthesize a variety of procedural textures in high speed, and is small enough to be implemented on one or two modern FPGA chips. The procedural texture generator was implemented using the Transmogrifier-2 (TM-2) rapid prototype system [LGI+98], as a part of a 3-D computer graphic rendering system design. The performance and hardware cost of synthesizing procedural textures in FPGAs are estimated using the data collected on the TM-2 system.

Chapter 2

Background and Motivation

This thesis uses techniques from the fields of fractal geometry, computer graphics, FPGAs and FPGA based custom computing machines. Relevant concepts and terminologies are reviewed in this chapter.

Fractal geometry was pioneered by Benoit Mandelbrot in 1970's. It is a mathematical framework for describing irregular shapes and is used by many procedural texture algorithms to model naturally formed objects. Section one reviews the major developments and concepts of fractal geometry. Section two reviews the development of procedural texture mapping. The Altera 10K50 FPGAs used to synthesize procedural textures are reviewed in section three. Section four reviews the development of FPGA based custom computing machines. Section five presents the major goals of this thesis and outlines the remaining chapters of the thesis.

2.1 Fractal Geometry

One of the most important properties of shapes and spaces is the number of dimensions. Classically, the number of dimensions are integer numbers. For example, a dot has an integer dimension of zero; a line has an integer dimension of one; a plane has an integer dimension of two; and a cube has an integer dimension of three. Similarly spaces are also defined by integer dimensions of one, two, and three. In three-dimensional space, one of the most widely used forms of geometry is the Euclidean geometry.

However, integer dimensions and Euclidean geometry cannot satisfactorily describe all shapes.

For describing shapes that are continuous but not smooth at any point (fractal), straight line segments of Euclidean geometry and the integer dimension numbers become inadequate. A example of one-dimensional fractal shapes is the silhouettes of coastlines [Hog, Gle87]. Coastlines are continuous curves that demarcate the boundary between the sea and the land. Like any fractal curves, coastlines cannot be approximated by any polygon path. This property was first discovered by the English scientist, Lewis F. Richardson in 1920s. Through extensive experiments, Richardson noted that if segments of length e are used to approximate a coastline in a polygon path, the total length of the polygon path $L(e)$ is approximately

$$L(e) \approx \frac{F}{e^{D-1}}$$

where F and D are constants and D is a real number between 1.0 and 2.0. As shown by the above equation, when e approaches zero, $L(e)$ approaches infinity. All similar curves, which have infinite length when approximated by a Euclidean polygon path, are defined to be fractal lines. Similar fractal shapes can also be found in two, three, and other multidimensional spaces.

The concept of fractal was first proposed by Benoit Mandelbrot when he rediscovered Richardson's finding in the 1970s [Gle87]. Mandelbrot also noticed that there are self similarities in the silhouettes of coastlines and in all fractal shapes. Coastlines have similar degree of roughness at all scales. If a segment of a coastline is magnified, the magnified segment possesses roughly the same degree of roughness as the unmagnified coastline. In fact, the magnified segment looks much like an unmagnified coastline. This kind of self similarity exists even when a coastline segment is magnified to several thousand times of its original size. And for some fractal lines, like the Koch curve, the self similarity can exist at an infinite scale of magnification.

To properly describe fractal shapes, Mandelbrot proposed the concept of fractal dimension. Fractal dimensions expand the classical definition of dimension from the positive integer domain into the positive real number domain. Each fractal dimension number is a real number. The integer part of a fractal dimension number is the same as the classical Euclidean dimension. The fractional part of a fractal dimension number represents roughness of the object. The fractal dimension of all Euclidean shapes, which are smooth at all points, are equal to their Euclidean dimension. However, the fractal dimensions of fractal shapes are always greater than their Euclidean dimension; and the difference between a fractal dimension number and a corresponding Euclidean dimension number can be as much as 1.0.

A formal mathematical definition of fractal geometry and fractal dimension is beyond the scope of this thesis. Interested readers can refer to the book "The Science of Fractal Images" [PS88] for more detail.

Many shapes and forms in nature have some fractal characteristics. These shapes exhibit self similarities across a range of scales. Clouds and air turbulence are other examples of fractal structures. Many biological systems, like the human blood vessel system, also have fractal structures. Because of the abundance of fractal shapes and forms in nature, many techniques have been developed in computer graphics to synthesize fractal images or use fractals to model phenomena. The Perlin noise function is one of the most successful methods. The next section reviews procedural texture mapping and the Perlin noise function.

2.2 Procedural Texture Mapping

In many computer graphic applications, polygon meshes are used to model the shapes of geometrical surfaces. To represent a surface using a polygon mesh, the surface is sampled at predetermined intervals; and the polygon mesh is then created by connecting the adjacent sampling points. The accuracy of the approximation can be increased by increasing the number of approximating polygons. For the most basic level of visual realism, each polygon in the polygon mesh can be assigned a single surface color. The polygons are rendered onto the screen using a series of geometrical transformations; and these transformation processes are often called the rendering pipeline.

Many algorithms can be used to increase the level of surface detail beyond the single color assigned to each polygon. These algorithms include various illumination and shading models, surface-detail polygons, and texture mapping. The technique of texture mapping was pioneered by Catmull [FHvD$^+$90, Cat74] and refined by Blinn and Newell [FHvD$^+$90, BN76]. It increases the level of surface detail by associating a digitized or synthesized two-dimensional image with each polygon. Using these images, texture mapping greatly increases the level of visual realism while requiring only a minimum amount of computation.

Texture mapping is performed after the polygons have been transformed into the screen coordinate system. To determine the color of a screen pixel, the polygon that covers that screen pixel is first identified. The screen coordinate of the screen pixel is then transformed into the texture

coordinate of the polygon. The texture coordinate is used to index into the texture memory containing the image. The corresponding color in the texture memory represents the surface color of the polygon at that pixel; and this color is used to calculate the final pixel color.

Peachey [Pea85] and Perlin [Per85] extended the traditional 2-D texture mapping to 3-D solid texture mapping. Traditional texture mapping uses two-dimensional images as textures, while solid texture mapping uses three-dimensional images as textures. Solid texture mapping is much simpler to use. It produces more realistic and accurate images for rendering objects whose surface textures are determined by the internal structure of the material that form them. Typical examples of these materials are concrete, wood and marble. Since 3-D textures are difficult to store and time consuming to obtain from real life objects, many solid textures are described as procedural textures.

Procedural texture mapping uses textures synthesized by computer algorithms. Many procedural textures are noise based textures. The use of noise functions introduces randomness and fractal to textures and allows the creation of more realistic images. One of the most widely used noise functions is the Perlin noise function [Per85]. Using this function, programmers can easily simulate the irregularities that characterize many naturally formed objects.

2.3 Altera 10K50 FPGA Architecture

Researchers at the Computer Engineering Department have been designing a multi-FPGA digital circuit prototype system, called the Transmogrifier. Several versions of the system have been designed; the current version, TM-2, is constructed using Altera 10K50 FPGAs. The Altera 10K50 FPGA architecture is well suited as a platform for generating procedural textures because of the following characteristics:

- large capacity of programmable logic
- inclusion of large programmable memory elements
- extensive software support
- register rich environment which can be used to implement circuits with many pipeline stages.
- support for implementing fast arithmetic circuits

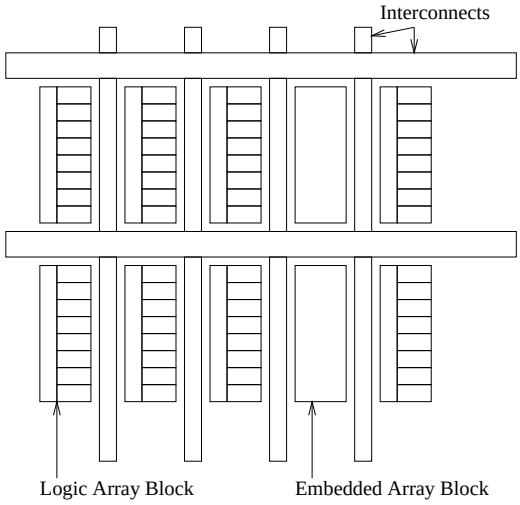

Figure 2.1: Altera 10K50 Architecture

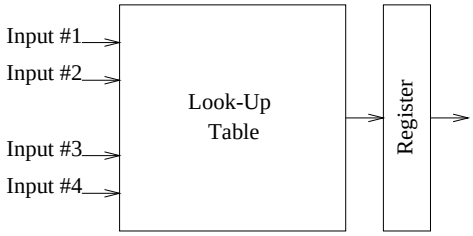

Figure 2.2: Logic Element Block Diagram

Each Altera 10K50 FPGA chip contains approximately $35,000$ programmable logic gates and over 300 programmable I/O pins. In addition, each chip contains $20,480$ programmable RAM bits, which are essential for the implementation of several procedural texture algorithms. The overall structure of the Altera 10K50 FPGAs are shown in Figure 2.1. There are two types of programmable logic structures, logic array blocks and embedded array blocks. The programmable logic structures can be connected to each other via vertical and horizontal interconnect channels.

The embedded array blocks are arrays of programmable RAM bits. Each of these blocks can be configured into 1×2048, 2×1024, 4×512, or 8×256 RAM arrays. Several embedded array blocks can be connected together to form larger RAM arrays. Each Altera 10K50 chip contains 10 of these embedded array blocks.

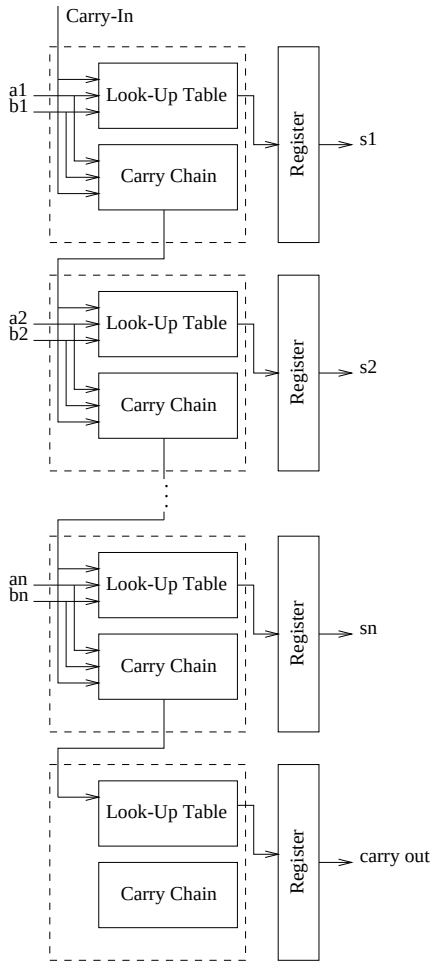

Figure 2.3: Logic Elements Configured for Arithmetic Operations

The main programmable logic structures of Altera 10K50 FPGAs are the logic array blocks. Each logic array block contains eight logic elements. The basic structure of a logic element is shown in Figure 2.2. It consists of a four-input look-up table and a programmable register. This architecture provides a register rich environment that can be used to implement circuits with many pipeline stages. Each logic element also provides support for implementing fast arithmetic circuits by providing fast carry chains. As shown in Figure 2.3, the four-input look-up table of each logic element can be configured into two three-input look-up tables. One of the three-input look-up table can be used to generate the sum signal of a full adder. The other three-input look-up table can be used to generate the carry signal. There is dedicated hardware for transporting the carry signals from one logic element to another, so fast carry propagation can be achieved for many arithmetic circuits.

The Altera 10K50 FPGAs are supported by the Altera Max+plus II software package. The software package provides a comprehensive digital design environment.

2.4 FPGA Based Custom Computing Machines

FPGAs have been used in many machines to accelerate computations. These machines are called the FPGA based custom computing machines. The graphic accelerator designed in this thesis can be classified as an FPGA based custom computing machine. This section briefly reviews the major progress in the field.

FPGA based custom computing machines can be classified based on the level of integration of their fixed unit and their reconfigurable unit. Early FPGA based custom computing machines, like PAM [BRV89, PB92] and SPLASH 2 [BAW96], were constructed using commercially available FPGA chips. Typically multiple FPGA chips are assembled on a computer board. The board is connected to a fixed unit, usually a stand alone computer, through a bus. The bus serves the purpose of communicating both data and control to and from the FPGA board. Typical applications include long integer multiplication, RSA decryption, text searching and genome sequence matching. As peripherals of their host computers, these systems are relatively simple to construct; however, the limited communication bandwidth between the reconfigurable and fixed system poses a severe limitation on the overall performance for many applications.

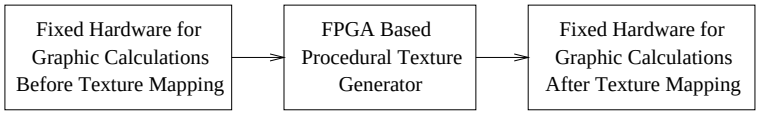

Figure 2.4: FPGA Based Custom Computer Machine for Procedural Texture Mapping

Later researchers investigated more closely coupled architectures. Machines like PRISM ĩteprism and UofT reconfigurable coprocessor [R94], connect their fixed and reconfigurable units using dedicated data and control buses. However, the reconfigurable unit, typically constructed using commercially available FPGA chips, and the fixed unit are implemented on separate chips.

Recent research has concentrated on constructing even more closely integrated machines. Machines like PRISC [Raz94], UofT OneChip [Wit95], and BRASS [Waw] closely integrate fixed and reconfigurable units on the same chip. The reconfigurable FPGA structures are treated as regular subsystems of the overall design. This level of integration provides the highest communication bandwidth between the reconfigurable units and the remaining system.

2.5 Goals and Outlines

The goals of this thesis are to design an FPGA based custom computing machine for procedural texture mapping, and implement the design on the TM-2. Figure 2.4 shows the block diagram of the machine. The fixed hardware handles tasks like geometric transformations and texture parameter calculation. Once the texture parameters are calculated, the system passes the parameters to the FPGA subsystem to generate texture values. For maximum performance, the FPGAs are design to be closely integrated with the rest of the system.

The performance goal of the design is to achieve close to real time animation performance. True real time animation is not expected given the limited resources and the implementation platform. A suite of procedural texture algorithms also have to be selected and translated into hardware designs.

The remaining portion of this thesis is divided into four chapters. Chapter three discusses the design of the machine. Chapter four describes the procedural textural algorithms selected and their hardware implementations. Chapter 5 presents the performance and cost data obtained from the

TM-2 implementation. Chapter 6 presents the conclusion and possible future work.

Chapter 3

3-D Rendering System

A 3-D computer graphic rendering system was designed to evaluate the implementation issues of synthesizing procedural textures in FPGA hardware. A prototype of the rendering system was also constructed. The architecture of the rendering system is briefly described in this chapter. The input to the rendering system is a list of triangles. Each vertex of these triangles is specified by two triplets. The first triple, (x, y, z), specifies the position of the vertex in a 3-D world space. The second triple, (u, v, w), specifies the position of the vertex in a 3-D texture space. The rendering system performs four major operations on each triangle. First, the system transforms the 3-D world coordinates of the vertices into the 2-D screen coordinates. Second, all pixels inside the triangle are determined using the 2-D screen coordinates of the vertices. The texture coordinates of these pixels are then calculated. Third, the system uses the texture coordinates to calculate the color of each pixel. Finally the image is stored in a frame buffer and displayed on a screen.

Figure 3.1 shows the overall architecture of the rendering system. It consists of four major components:

1. a world to screen space transformation (WSST) unit

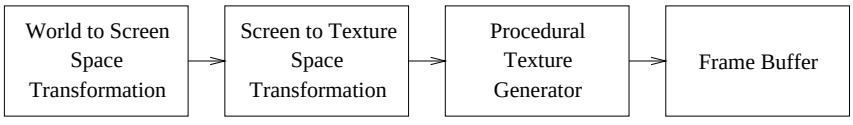

Figure 3.1: 3-D Rendering System Using Procedural Textures

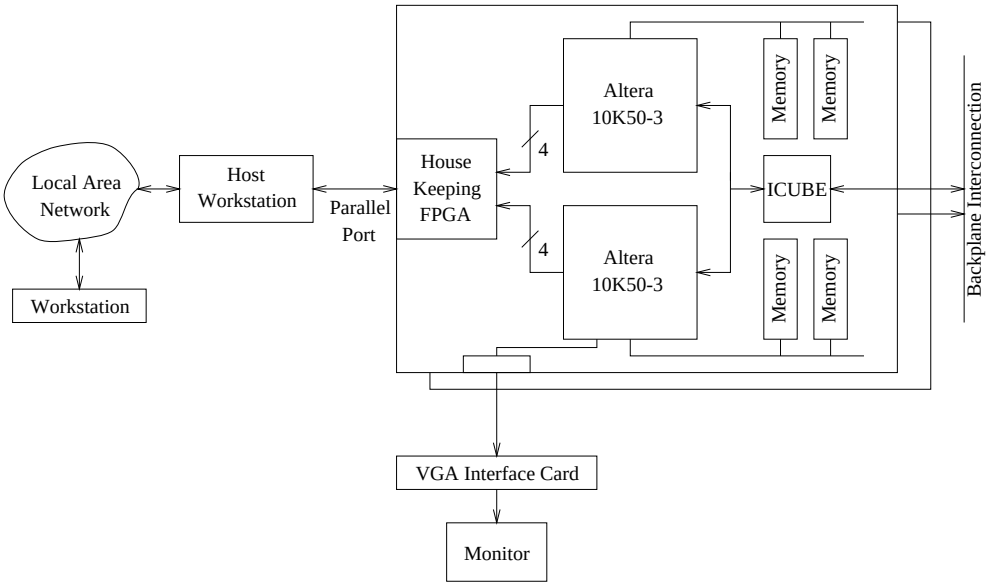

Figure 3.2: Experimental Setup

2. a screen to texture space transformation (STST) unit

3. a procedural texture generator

4. a frame buffer

Each component performs one of the operations listed in the previous paragraph. Conventionally, WSST functions are usually implemented in software; STST and the frame buffer are implemented in hardware; and textures are implemented using a RAM. This thesis proposes to implement textures in FPGAs as a procedural texture generator. A set of textures can be implemented by loading their algorithms into the FPGA based procedural texture generator. Although STST and the frame buffer should ideally be implemented in an ASIC, they were constructed in FPGAs on the prototype.

The prototype of the 3-D rendering system was implemented on the TM-2. As shown in Figure 3.2, the TM-2 consists of two boards. Each board contains two Altera 10K50 FPGAs and four banks of 64-bit wide SRAM. The TM-2 system can be connected to a local area network through a host workstation. Using the host, any workstation on the network can communicate with the TM-2.

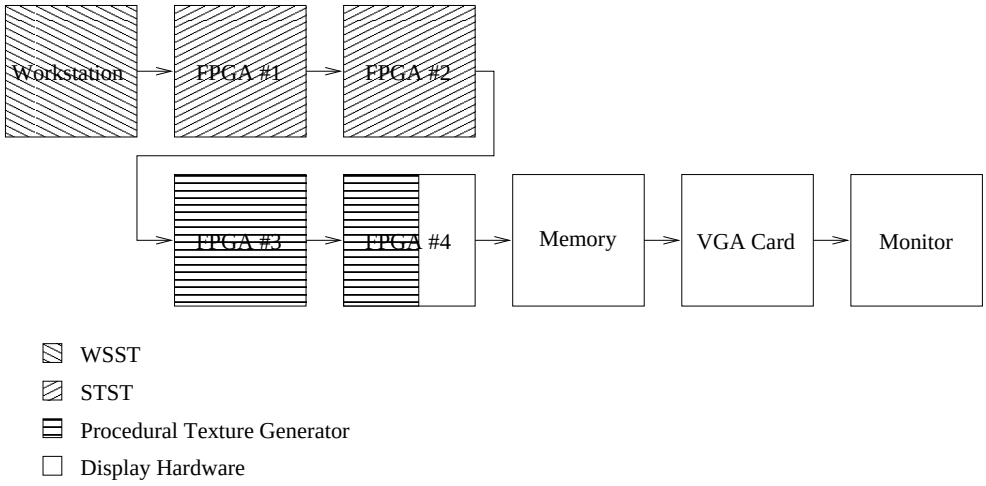

Figure 3.3: Partition of the Hardware Resources

The resources used in the implementation include one workstation, all four FPGAs on the TM-2, one bank of TM-2 SRAM, a VGA card, and a monitor. The workstation is connected to the TM-2 via the local area network. The partitioning of the rendering system among all hardware resources is shown in detail in Figure 3.3. Since there are only four FPGAs available, the entire rendering system cannot be implemented on the TM-2 system. The WSST calculations are performed once per triangle, while other units perform calculations once per pixel. Therefore, the WSST unit is implemented on the workstation, as commonly done in many graphic cards. Two FPGAs are allocated to the STST unit. One and a half FPGAs are allocated for the procedural texture generator. The frame buffer is implemented using the remaining resources. It uses one bank of TM-2 SRAM as a double frame buffer. It also controls the VGA card and the monitor.

All software is written in the C programming language. All hardware designs are done in the Altera Hardware Description Language (AHDL). The rendering system uses a screen space resolution of 512×512. The texture space resolution is $512 \times 512 \times 512$. Each color is represented using eight bits.

3.1 World to Screen Space Transformation

The WSST unit can be built using a variety of architectures. In low cost implementation, the functions of the unit can be performed by the CPU of a host system. In high performance implementation, on the other hand, the unit can be custom designed. The detail design of such a unit is beyond the scope of this thesis. Instead, this section describes the algorithms executed by the unit.

As mentioned earlier, the WSST unit transforms a list of triangles from the 3-D world space to the 2-D screen space. The major operations are clipping, perspective projection, solid texture parameter calculation and triangle-quad transformation.

3.1.1 Clipping Algorithm

Not all triangles within the initial triangle list can be displayed on the final screen. Only those triangles that are within a finite volume in the 3-D world space are actually displayed, the rest triangles are discarded from the list. The algorithms used to remove off-screen triangles from the triangle list are called the clipping algorithms [FHvD$^+$90]. The finite volume is called the view volume [FHvD$^+$90]. For efficient implementation of the 3-D graphic system, the clipping process is performed as early as possible in the rendering pipeline.

The clipping algorithm used in this thesis is similar to the Sutherland-Hodgman polygon-clipping algorithm [FHvD$^+$90, SH74]. The algorithm uses a perspective projection view volume as shown in Figure 3.4. The view volume is bounded by six clipping planes in the 3-D world space. The six clipping planes are $z = -1$, $z = -0.05$, $x = z$, $x = -z$, $y = z$, and $y = -z$. The front and back planes are $z = -0.05$, and $z = -1$, respectively. In 3-D world space, the clipping algorithm classifies all triangles in the triangle list into three classes, triangles that are completely inside the view volume, triangles that are completely outside the view volume, and triangles that are partially inside the view volume. Triangles that are completely inside the view volume are kept in the triangle list while triangles that are completely outside the view volume are removed from the triangle list.

Triangles that are partially inside the view volume are the triangles that have at least one vertex inside the view volume and at least one vertex outside the view volume. The algorithm partitions

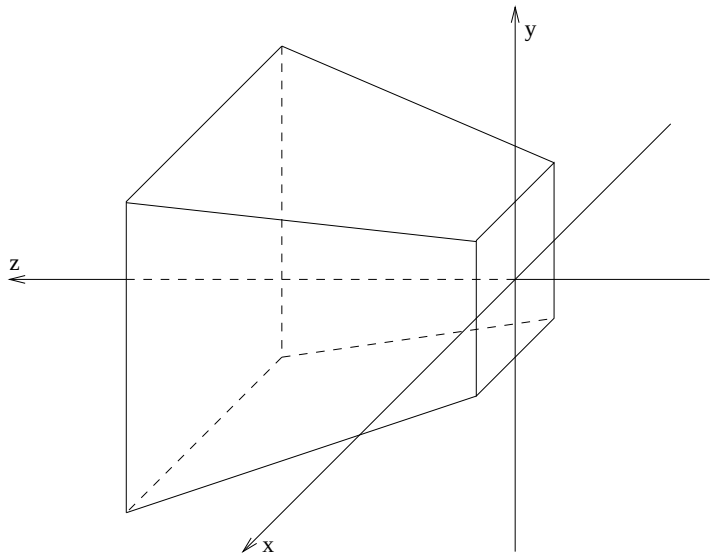

Figure 3.4: View Volume

each of these triangles into two polygons, one completely inside the view volume and the other completely outside the view volume. The polygon that is completely outside the view volume is discarded. The polygon that is completely inside the view volume is partitioned into a series of triangles which are then added back to the triangle list.

Figure 3.5 shows the algorithm used to clip a triangle partially inside the view volume. It is a general clipping algorithm for polygons, so at the start, the triangle is copied into a polygon structure. The algorithm clips the polygon against one clipping plane at a time. To clip a polygon against a clipping plane, all intersection points between the edges of the polygon and the plane are calculated using standard geometric equations. These intersection points and the vertices that are on the same side of the plane as the view volume are then used to create a new polygon. The vertices on the other side of the view volume are discarded. The new polygon then replaces the current polygon; and the next clipping plane replaces the current clipping plane. The same algorithm is repeated until all six clipping planes are clipped against. The final polygon is the portion of the initial triangle that is completely inside the view volume and is always convex.

Figure 3.6 illustrates the method used to partition the clipped polygon into a series of triangles. One vertex of the polygon is used in the creation of all triangles. This vertex and each neighboring pair of the remaining vertices are used to create one triangle. This algorithm works only for convex

```
function clip (t1)
-- INPUT DESCRIPTION: t1 is the triangle to be clipped
  let p1 be a polygon with three vertices;
  let p2 be a polygon with zero vertices;
  set polygon p1 to be identical to triangle t1;
  for each clipping plane, cp, of the view volume do
    for each vertex, v1, of the polygon p1 do
      if v1 and the view volume are on the same side of cp then
        increase the number of vertices of p2 by 1;
        let the new vertex of p2 be identical to v1;
      end if;
      let v2 be identical to the right neighboring
      vertex of v1 on polygon p1;
      let v3 be the intersection point between the
      edge (v1,v2) and the clipping plane cp;
      if v3 is between v1 and v2 on edge (v1,v2) then
        increase the number of vertices of p2 by 1;
        let the new vertex of p2 be identical to v3;
      end if;
    end for;
    set polygon p1 to be identical to polygon p2;
    set polygon p2 to be a polygon with zero vertices;
  end for;
  return p1;
end function;
```

Figure 3.5: Clipping Algorithm

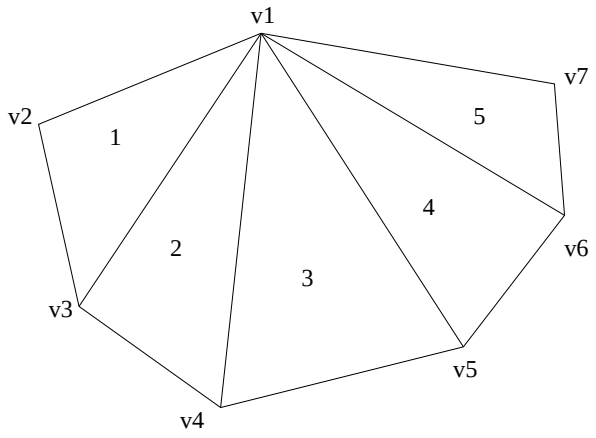

Figure 3.6: Decomposing a Polygon into Triangles

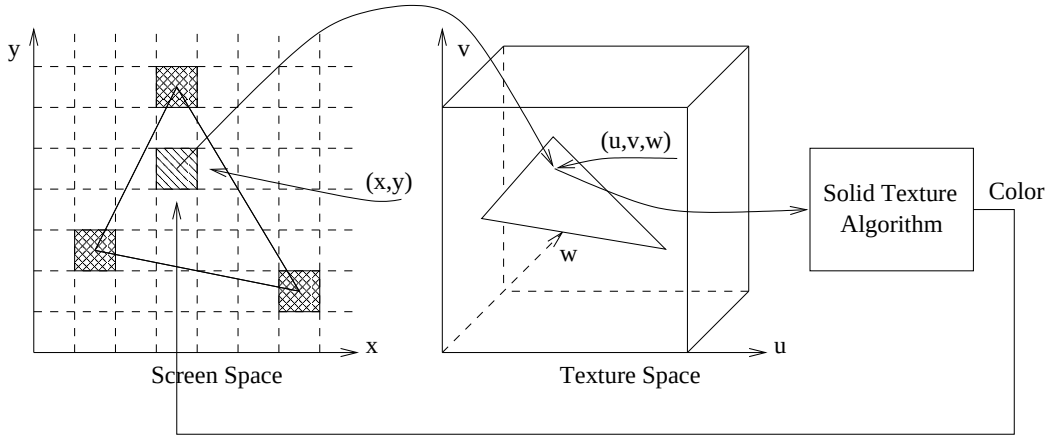

Figure 3.7: Screen Space to Texture Space Transformation

polygons and produces incorrect results for concave polygons. The inputs to the clipping algorithm are triangles. With triangular inputs, the algorithm produces only convex polygons; therefore, this division algorithm always works for the chosen clipping algorithm.

3.1.2 Perspective Projection Algorithm

After clipping, the triangles are projected into the 2-D screen space using perspective projection. The perspective projection is done once for every triangle vertex using the following equation:

$$x_{screen} = x_{world}/z_{world}$$
$$y_{screen} = y_{world}/z_{world}$$

x_{world}, y_{world}, and z_{world} represent the x, y, and z coordinates of the vertex in the world space, while x_{screen} and y_{screen} represent the x and y coordinates in the screen space.

3.1.3 Solid Texture Parameter Calculation Algorithm

The perspective projection algorithm discussed in the previous section does not project every point on a triangle into the screen space. Instead, only the three vertices of the triangle are projected. The STST unit then uses the screen coordinates of these three vertices to find all screen pixels inside the triangle.

As shown in Figure 3.7, to determine the color of a pixel, the screen coordinates of the pixel are used to calculate its texture coordinates. The texture coordinates are then used to calculate the color of the pixel based on the specified procedural texture algorithm. The transformation from the screen coordinates to the texture coordinates is performed by the STST unit. The texture calculation is done by the procedural texture generator.

The algorithm for screen to texture space transformation is not widely published. During this study, a method was independently derived to perform this transformation. In this method, the WSST unit assists the STST unit by generating fifteen parameters for the screen to texture space transformation. This section defines these fifteen parameters and discusses the algorithm used to calculate these parameters.

The solid texture parameters are calculated based on the assumption that for each triangle, there exists an affine transformation from the world space to the texture space and vice-versa. This affine transformation is specified by the following set of equations:

$$\begin{aligned} au + bv + cw + d &= X \\ eu + fv + gw + h &= Y \\ iu + jv + kw + l &= Z \end{aligned} \tag{3.1}$$

where a, b, c, d, e, f, g, h, i, j, k, l are affine transformation constants; u, v, w are the texture coordinates; and X, Y, Z are the world coordinates. In texture space, the plane in which the triangle exists can be described by the following equation:

$$ou + pv + qw + r = 0 \tag{3.2}$$

where o, p, q, r are constants. Finally the perspective projection from world space to screen space can be described by the following set of equations:

$$\begin{aligned} mx &= X \\ my &= Y \\ m &= Z \end{aligned} \tag{3.3}$$

where x, y are the screen coordinates.

Substituting Equation 3.3 in Equation 3.1 and Equation 3.2 for X, Y, Z results in the following

equations:

$$\begin{aligned} au + bv + cw + d &= mx \\ eu + fv + gw + h &= my \\ iu + jv + kw + l &= m \\ ou + pv + qw + r &= 0 \end{aligned} \tag{3.4}$$

Equation 3.4 describes the relationship between the screen coordinates, (x, y), and the texture coordinates, (u, v, w). At this stage, the problem becomes to re-arrange Equation 3.4, so u, v, w, m is written in terms of x and y. Since we have four linear equations in Equation 3.4, this problem can be solved in the following way.

Re-arranging Equation 3.4 results in the following set of equations:

$$\begin{aligned} au + bv + cw - mx + 0y + 0 &= -d \\ eu + fv + gw + 0x - my + 0 &= -h \\ iu + jv + kw + 0x + 0y - m &= -l \\ ou + pv + qw + 0x + 0y + 0 &= -r \end{aligned} \tag{3.5}$$

Equation 3.5 has the following matrix representation:

$$V_b = M \times V_a \tag{3.6}$$

where:

$$V_a = \begin{bmatrix} u \\ v \\ w \\ mx \\ my \\ m \end{bmatrix} \quad V_b = \begin{bmatrix} -d \\ -h \\ -l \\ -r \end{bmatrix} \quad M = \begin{bmatrix} a & b & c & -1 & 0 & 0 \\ e & f & g & 0 & -1 & 0 \\ i & j & k & 0 & 0 & -1 \\ o & p & q & 0 & 0 & 0 \end{bmatrix}$$

Use Gaussian elimination to numerically solve for X, such that

$$X = \begin{bmatrix} X_{00} & X_{01} & X_{02} & X_{03} \\ X_{10} & X_{11} & X_{12} & X_{13} \\ X_{20} & X_{21} & X_{22} & X_{23} \\ X_{30} & X_{31} & X_{32} & X_{33} \end{bmatrix}$$

and

$$X \times M = \begin{bmatrix} 1 & 0 & 0 & A & B & C \\ 0 & 1 & 0 & D & E & F \\ 0 & 0 & 1 & G & H & I \\ 0 & 0 & 0 & J & K & L \end{bmatrix}$$

where X_{00}, X_{01}, X_{02}, X_{03}, X_{10}, X_{11}, X_{12}, X_{13}, X_{20}, X_{21}, X_{22}, X_{23}, X_{30}, X_{31}, X_{32}, X_{33}, A, B, C, D, E, F, G, H, I, J, K, and L are all numerical constants, computed during the Gaussian elimination. Solving analytically for u, v, w, and m, we have:

$$\begin{aligned} u &= U_{init} + (a_{00} \times x + a_{01} \times y + Y_{0init})/(a_{30} \times x + a_{31} \times y + Y_{3init}) \\ v &= V_{init} + (a_{10} \times x + a_{11} \times y + Y_{1init})/(a_{30} \times x + a_{31} \times y + Y_{3init}) \\ w &= W_{init} + (a_{20} \times x + a_{21} \times y + Y_{2init})/(a_{30} \times x + a_{31} \times y + Y_{3init}) \end{aligned} \tag{3.7}$$

$$m = Z = -M_{init}/(a_{30} \times x + a_{31} \times y + Y_{3init}) \tag{3.8}$$

where the fifteen solid texture parameters are:

$$\begin{aligned} a_{00} &= A \times M_{init}, \quad a_{01} = B \times M_{init}, \quad Y_{0init} = C \times M_{init}, \\ a_{10} &= D \times M_{init}, \quad a_{11} = E \times M_{init}, \quad Y_{1init} = F \times M_{init}, \\ a_{20} &= G \times M_{init}, \quad a_{21} = H \times M_{init}, \quad Y_{2init} = I \times M_{init}, \\ a_{30} &= J, \qquad\qquad a_{31} = K, \qquad\qquad Y_{3init} = L, \\ U_{init} &= -dX_{00} - hX_{01} - lX_{02} - rX_{03}, \\ V_{init} &= -dX_{10} - hX_{11} - lX_{12} - rX_{13}, \\ W_{init} &= -dX_{20} - hX_{21} - lX_{22} - rX_{23} \end{aligned}$$

and

$$M_{init} = (d \times X_{30} + h \times X_{31} + l \times X_{32} + r \times X_{33})$$

Overall, the algorithm used by the WSST unit to calculate the fifteen parameters consists of three steps. First, for a given triangle, the affine transformation between its texture space representation and its world space representation is first calculated. These equations are listed as Equation 3.1. Second, the plane in which the triangle exists in texture space is calculated. This equation is listed as Equation 3.2. Finally, Equation 3.4 is constructed and Gaussian elimination is used to calculate the fifteen parameters. Once these fifteen parameters are derived, they are given to the STST unit. For each pixel in the given triangle, the STST unit transforms the screen coordinates of the pixel into the corresponding texture coordinates using Equation 3.7.

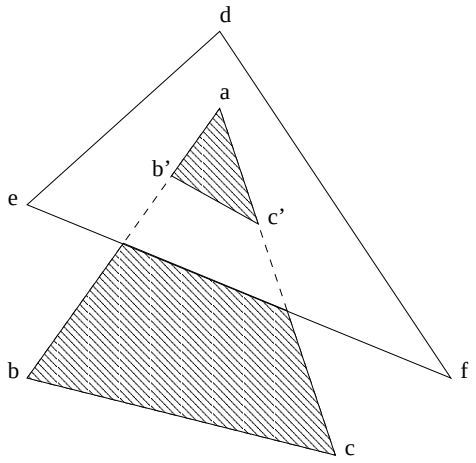

Figure 3.8: Two Intersecting Triangles

3.1.4 Triangle-Quad Transformation Algorithm

After perspective projection, some areas of the screen might be covered by more than one triangle. The pixel colors in these screen areas should be determined by the triangles that are nearest to the screen (the top triangle). Conventional graphic systems typically use the z-buffering [FHvD$^+$90] algorithm to identify the top triangle for each pixel in hardware. This method, however, would require a large amount of memory and logic resources that are not available on the TM-2 system. Instead for the prototype, the task of finding the top triangle for each area is performed by the WSST unit. A scan-line algorithm [FHvD$^+$90, WREE67, Bou70, BK70, Wat70] is used for the task. As other scan-line algorithms, this algorithm assumes that no triangles in the scene are intersecting each other. If one triangle intersects another as shown in Figure 3.8, these triangles have to be first divided into several non-intersecting triangles. This means, in Figure 3.8, triangle abc must be divided into polygon $bb'c'c$ and triangle $ab'c'$. Polygon $bb'c'c$ has to be further divided into triangle $bb'c'$ and triangle bcc'.

This scan-line algorithm differs from other algorithms in the final outputs. Regular scan-line algorithms output the final rendered images. This scan-line algorithm, on the other hand, divides each scan-line into sections. Sections from different scan-lines are then collected into special quadrilaterals called quads. These quads are the output of this scan-line algorithm. The quads, along with their solid texture parameters, are sent to the STST unit, where they are finally rendered

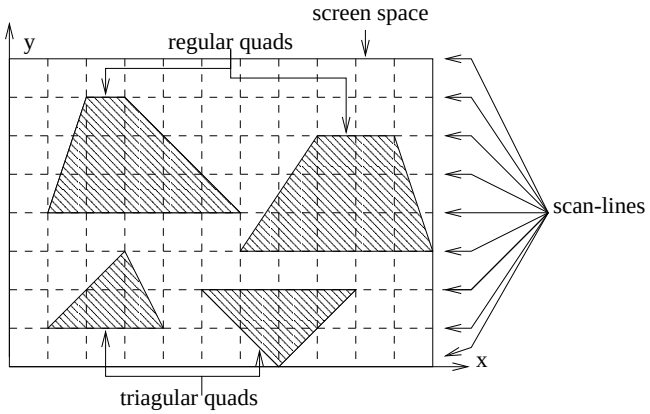

Figure 3.9: Quads

onto the computer screen. The rest of this section describes the scan-line algorithm in detail.

Quads [Gal96] are quadrilaterals whose top and bottom edges are parallel to the scan-lines in screen space. Figure 3.9 shows some examples of quads. As shown in the figure, triangles also can be quads. A triangle is a quad if one edge of the triangle is parallel to the scan-lines in screen space. Quads are chosen to be the WSST unit output and the STST unit input because they are easier to render in hardware than triangles. Using quads as the STST unit input simplifies the hardware design of the STST unit. In some other implementation, this dividing boundary between the WSST unit functionality and the STST unit functionality can be changed based on the relative complexity of the components.

To create quads, the list of triangles has first to be clipped, and perspectively projected into the screen space. If the triangles are texture mapped, their solid texture parameters also have to be calculated beforehand.

In screen space, the algorithm takes a list of triangles and generates a list of quads. The quads are generated by investigating the triangle edges one scan-line at a time. Using an active edge table, all edges that are on the current scan-line are recorded. These edges are called the active edges. In the table, the active edges are sorted based on their intersection points with the current scan-line. Edges with small x intersection coordinates are placed at the front of the table, while edges with large x intersection coordinates are placed at the back of the table.

Moving from one scan-line to a neighboring scan-line, two events might arise that will change

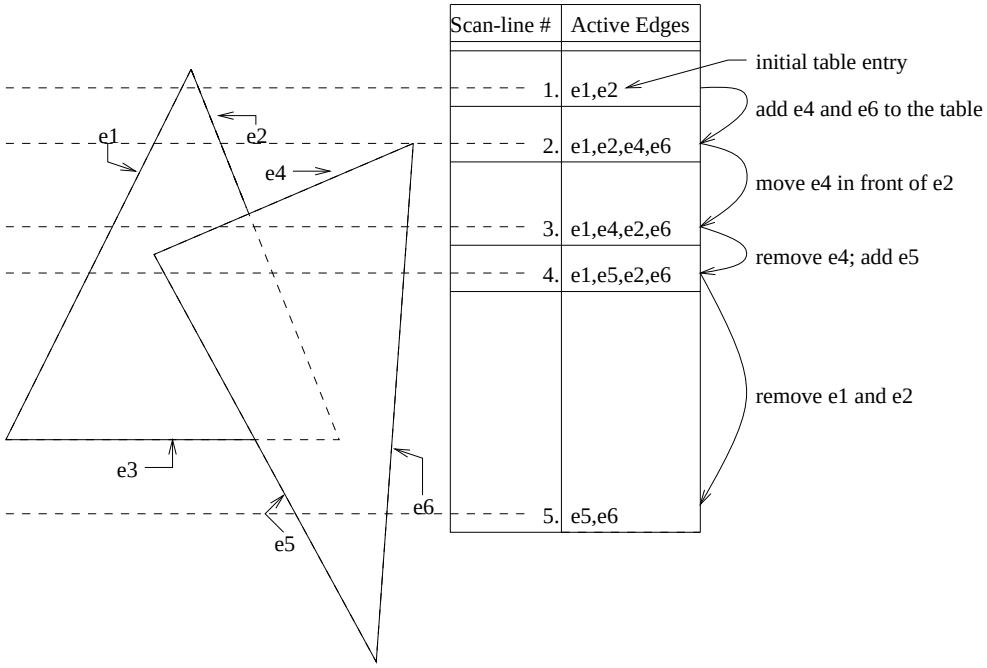

Figure 3.10: Scan-line Conversion from Triangles to Quads

the contents of the active edge table. First, active edges might be deleted or added to the table. An active edge on the previous scan-line might no longer be active on the current scan-line. This edge must be deleted from the active edge table. Also an edge that is not active on the previous scan-line might become active on the current scan-line. This edge must be added to the active edge table. Second, the order of active edges might change. The x intersection coordinates between the active edges and the scan-lines are different from one scan-line to the next. Whenever a new scan-line becomes the current scan-line, the active edge table has to be resorted based on the new intersection points. These two situations are illustrated in Figure 3.10.

The algorithm starts at the first scan-line on the screen and moves down the screen one scan-line at a time. For each scan-line, the algorithm calculates the current active edge table. When the contents of the active edge table is changed between two neighboring scan-lines, the top scan-line marks the end of a group of quads and the bottom scan-line marks the beginning of a new group of quads. When all scan-lines have been investigated, the entire screen is divided into a series of quads. An example is shown in Figure 3.11. There are two triangles in this example. Using the scan-line algorithm, the screen space is divided into seven quads. Three of them are covered by the

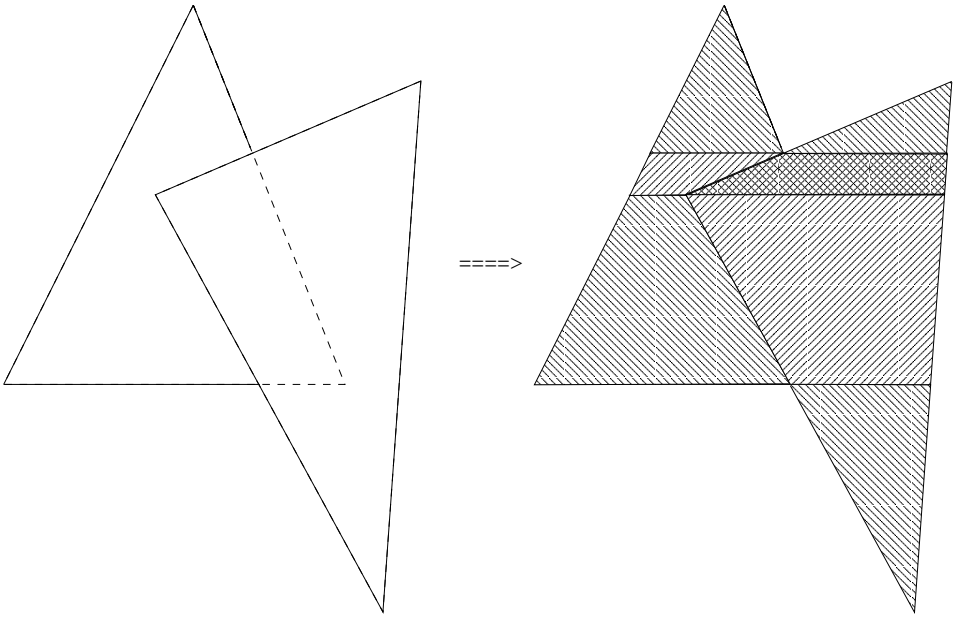

Figure 3.11: Triangle to Quad Transformation

left triangle; and the rest four are covered by the right triangle.

The following method is used to determine the correct triangle that covers each quad. For each quad, an interior point is used as a test point. In the screen space, each triangle that contains this test point covers the screen area occupied by the quad. Among these triangles, the triangle that is closest to the screen at the corresponding point in the world space is the closest triangle to the screen area occupied by the quad. This triangle is then determined using Equation 3.8 and its solid texture parameters are used to derive the solid texture parameters of the quad.

The geometry of a quad can be specified using six parameters [Gal96]:

- y_{init}:
  the top scan-line position

- y_{final}:
  the bottom scan-line position

- $x_{leftinit}$:
  the x coordinate of the left edge and the top scan-line intersection

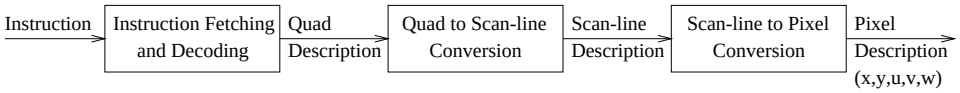

Figure 3.12: Major Functional Blocks of STST Unit

- $x_{leftinc}$:
  the slope of the left edge

- $x_{rightinit}$:
  the x coordinate of the right edge and the top scan-line intersection

- $x_{rightinc}$:
  the slope of the right scan-line

To draw a texture mapped quad on the screen, the WSST unit communicates these six parameters along with the fifteen solid texture parameters to the STST unit.

3.2 Screen to Texture Space Transformation

The WSST unit transforms a list of triangles in the 3-D world space into a list of quads in the 2-D screen space. The STST unit is then used to transform each quad into a list of pixel descriptions.

Overall, the STST unit performs three tasks. These three tasks are illustrated in Figure 3.12. First, the unit fetches and decodes instructions from the WSST unit. These instructions are transformed into quad descriptions. Second, once an entire quad description is received, the STST unit transforms the quad description into a series of scan-line descriptions. Third, these scan-line descriptions are then transformed into a series of pixel descriptions.

As shown in Figure 3.12, the output of the STST unit consists of sets of five parameters, x, y, u, v, w. Each set of these five parameters is used to describe a single pixel. Parameters x and y specify the location of the pixel on the screen. Parameters u, v, and w specify the solid texture coordinate associated with the given pixel. The solid texture coordinates are then sent to the procedural texture generator for calculating the final color of the pixels.

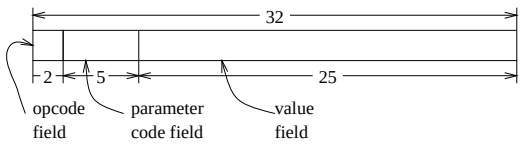

Figure 3.13: STST Unit Instruction Format

3.2.1 Instruction Fetching and Decoding Unit

The output of the WSST unit is a list of quads. Each quad in the list is specified by twenty-one parameters. These parameters are communicated to the STST unit using the instruction format shown in Figure 3.13. As shown in the figure, each instruction consists of thirty-two bits. These thirty-two bits are divided into three fields: an op-code field, a parameter code field, and a value field. The op-code field consists of two bits. The parameter code field consists of five bits; and the immediate value field consists of twenty-five bits.

There is one instruction for each quad parameter. When the instruction is used to specify a quad parameter, the op-code field is set to 0. The parameter code field is used to specify the type of quad parameter associated with the instruction; these values are shown in detail in Table 3.1. The value of the parameter is stored in the value field. The instruction fetching and decoding unit contains twenty-one 25-bit registers. Each of these registers is used to store the value of one quad parameter.

In addition to the quad parameter instructions, there are three more instructions in the instruction set. All these three instructions use only the op-code field. The parameter code field and the value field are ignored for these instructions. The start instruction instructs the hardware to start to render the quad based on the value currently stored in the quad parameter registers. When a double frame buffer is used, the switch frame buffer instruction instructs the hardware to switch to another frame buffer for rendering the quads that follows the instruction. The last instruction is the clear screen instruction. It simply instructs the hardware to clear the entire screen area to the background color. The formats of all instructions are shown in Table 3.1.

The basic structure of the instruction fetching and decoding unit is shown in Figure 3.14. The unit consists of three major components, the controller, the decoder, and the register file. The controller handles the communication between the unit and the rest of the system. As shown in the

Instruction Name	Opcode	Parameter Code	Description
quad parameter 0	0	0	y_{init}
quad parameter 1	0	1	y_{final}
quad parameter 2	0	2	$x_{leftinit}$
quad parameter 3	0	3	$x_{leftinc}$
quad parameter 4	0	4	$x_{rightinit}$
quad parameter 5	0	5	$x_{rightinc}$
quad parameter 6	0	6	$Y_{0init} + a_{00} \times x_{leftinit} + a_{01} \times y_{init}$
quad parameter 7	0	7	$Y_{1init} + a_{10} \times x_{leftinit} + a_{11} \times y_{init}$
quad parameter 8	0	8	$Y_{2init} + a_{20} \times x_{leftinit} + a_{21} \times y_{init}$
quad parameter 9	0	9	$Y_{3init} + a_{30} \times x_{leftinit} + a_{31} \times y_{init}$
quad parameter 10	0	10	a_{00}
quad parameter 11	0	11	a_{01}
quad parameter 12	0	12	a_{10}
quad parameter 13	0	13	a_{11}
quad parameter 14	0	14	a_{20}
quad parameter 15	0	15	a_{21}
quad parameter 16	0	16	a_{30}
quad parameter 17	0	17	a_{31}
quad parameter 18	0	18	U_{init}
quad parameter 19	0	19	V_{init}
quad parameter 20	0	20	W_{init}
start	1	X	start
switch frame buffer	2	X	switch frame buffer
clear screen	3	X	clear screen

Table 3.1: STST Unit Instructions

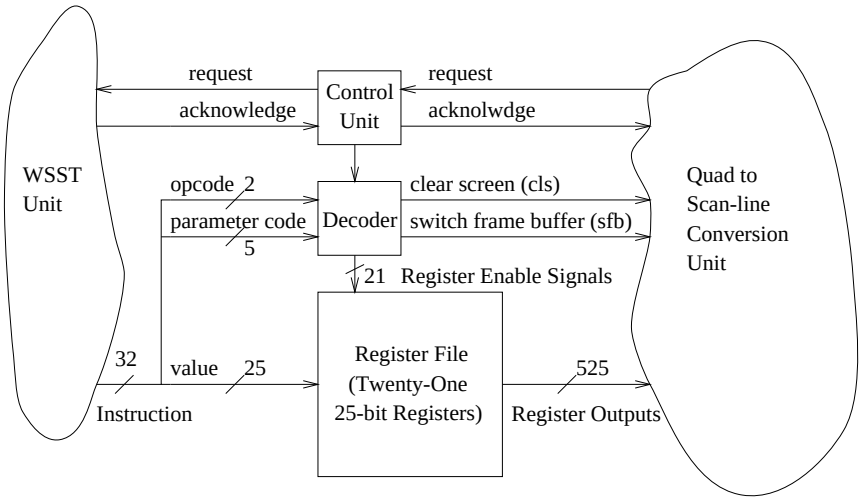

Figure 3.14: Instruction Fetching and Decoding Unit

figure, the communication protocol used is the two-stage hand shake protocol [Kat90]. Since this communication protocol is an asynchronous protocol, the instructions are processed independently from the execution speed of the rest of the system. This is desirable, since the time, needed by the WSST unit to perform its tasks, and the time, needed by the STST unit to render a quad, both are variable and input dependent.

The decoder is also controlled by the controller. It accepts the op-code and parameter code of the input instruction. When instructed by the controller, the decoder generates appropriate register enable signals, the clear screen signal, or the switch buffer signal. If the input instruction contains one of the quad parameters, the corresponding register in the register file is updated. All register outputs are available in parallel to the quad to scan-line conversion unit of the STST unit.

3.2.2 Quad to Scan-line Conversion Unit

As shown in Figure 3.15, a quad can be decomposed into $y_{final} - y_{init} + 1$ number of scan-lines. Each scan-line starts at point (x_{sleft}, y_s) and ends at point (x_{sright}, y_s). y_s, x_{sleft} and x_{sright} are

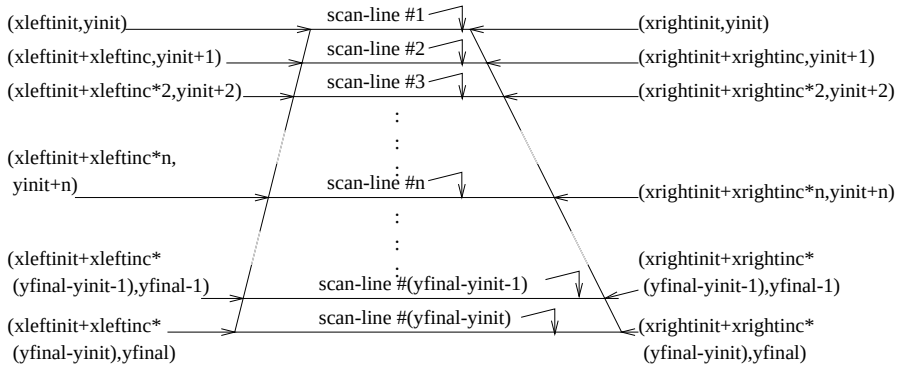

Figure 3.15: Decomposing a Quad into Scan-lines

defined by:

$$\begin{aligned} y_s &= y_{init} + n \\ x_{sleft} &= x_{leftinit} + x_{leftinc} \times n \\ x_{sright} &= x_{rightinit} + x_{rightinc} \times n \end{aligned} \tag{3.9}$$

where n is an integer between 0 and $y_{final} - y_{init}$.

To implement solid texture mapping, four more variables, Y_{0s}, Y_{1s}, Y_{2s}, Y_{3s}, are calculated by the quad to scan-line conversion unit for every scan-line. These values are defined by the following equations:

$$\begin{aligned} Y_{0s} &= Y_{0init} + a_{00} \times x_{leftinit} + a_{01} \times y_{init} + a_{00} \times x_{sdiff} \times n + a_{01} \times n \\ Y_{1s} &= Y_{1init} + a_{10} \times x_{leftinit} + a_{11} \times y_{init} + a_{10} \times x_{sdiff} \times n + a_{11} \times n \\ Y_{2s} &= Y_{2init} + a_{20} \times x_{leftinit} + a_{21} \times y_{init} + a_{20} \times x_{sdiff} \times n + a_{21} \times n \\ Y_{3s} &= Y_{3init} + a_{30} \times x_{leftinit} + a_{31} \times y_{init} + a_{30} \times x_{sdiff} \times n + a_{31} \times n \end{aligned} \tag{3.10}$$

where n is an integer and is between 0 and $y_{final} - y_{init}$; and $x_{sdiff} = \lfloor x_{leftinit} + x_{leftinc} \times n \rfloor - \lfloor x_{leftinit} + x_{leftinc} \times (n-1) \rfloor$.

In the actual hardware implementation, y_s, x_{sleft}, x_{sright}, Y_{0s}, Y_{1s}, Y_{2s} and Y_{3s} are calculated using the incremental algorithm shown in Figure 3.16. For every loop iteration, the algorithm outputs the parameters for one scan-line. At first, variables y_s, x_{sleft}, x_{sright}, Y_{0s}, Y_{1s}, Y_{2s} and Y_{3s} are initialized to contain the values associated with the top scan-line of the quad. For every loop iteration, the value of the above variables associated with the next scan-line down the screen is calculated. The algorithm terminates when the last scan-line of the quad is reached.

```
ys = yinit
xsleft  = xleftinit
xsright = xrightinit
// Assigning Y0s Y1s Y2s Y3s to parameter 6 7 8 9 respectively
Y0s = Y0init + a00 * xleftinit + a01 * yinit
Y1s = Y1init + a10 * xleftinit + a11 * yinit
Y2s = Y2init + a20 * xleftinit + a21 * yinit
Y3s = Y3init + a30 * xleftinit + a31 * yinit
while (ys <= yfinal) {
  ys = ys + 1;
  xsleftold = floor(xsleft)
  xsleft  = xsleft + xleftinc
  xsright = xsright + xrightinc
  xsdiff = floor(xsleft) - xsleftold
  Y0s = Y0s + a00 * xsdiff + a01
  Y1s = Y1s + a10 * xsdiff + a11
  Y2s = Y2s + a20 * xsdiff + a21
  Y3s = Y3s + a30 * xsdiff + a31
}
```

Figure 3.16: Incremental Algorithm for Quad to Scan-line Conversion

The quad to scan-line conversion process is executed in parallel with the scan-line to pixel conversion process. While the scan-line to pixel conversion unit is processing scan-line n, the quad to scan-line conversion unit is generating scan-line $n+1$. This parallel execution method exploits the inherent parallelism in the pixel rendering algorithm. It also provides the opportunities to further simplify the hardware design for the quad to scan-line conversion unit.

The STST unit is designed to have a throughput of close to one pixel per clock cycle. Since each scan-line typically consists of several pixels, more than one clock cycle can be used to generate one scan-line. One of the possible designs for the quad to scan-line unit is shown in Figure 3.17. Seven registers are used to contain the values of the variables y_s, x_{sleft}, x_{sright}, Y_{0s}, Y_{1s}, Y_{2s}, and Y_{3s}. Another eleven registers contain the corresponding increment values for these variables. Four of these, labeled a_{00}, a_{10}, a_{20}, a_{30}, are shift registers. Additional initialization logic is not shown in the figure. Only one adder is used in the design. The controller uses multiple clock cycles to calculate the seven parameters for each scan-line. If on average, there are many pixels per scan-line, the multiple clock cycles used per scan-line will not have a significant impact on the overall performance of the STST unit. However, if higher performance is desired, more adders should be

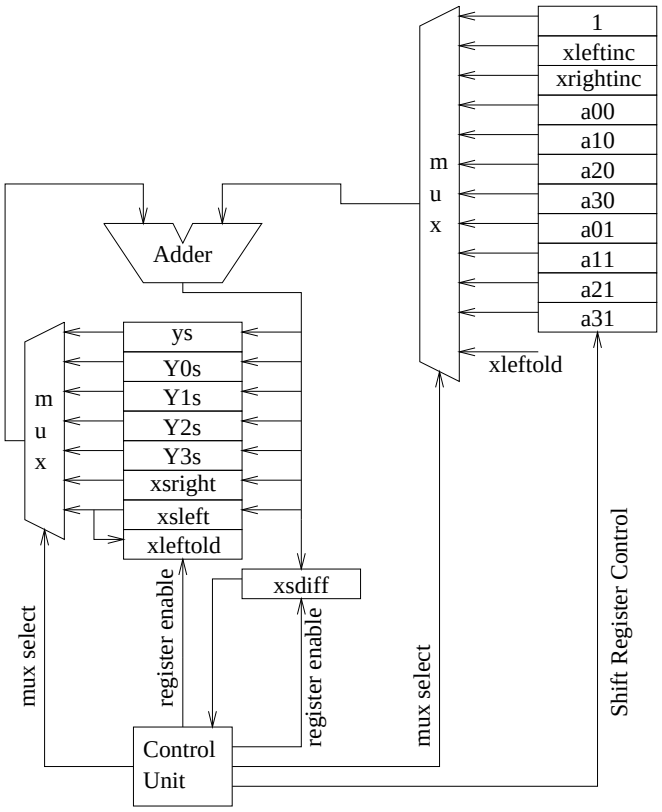

Figure 3.17: Datapath for Quad to Scan-line Conversion

used.

In the current TM-2 implementation, three adders are used. Adder one is used to compute ys, $xsleft$, $xsright$, and $xsdiff$. Adder two is used to computer $Y0s$ and $Y1s$. Adder three is used to computer $Y2s$ and $Y3s$. Each multiplications is decomposed into a sequence of additions completed over 10 clock cycles. Twenty-three cycles are needed to compute all the parameters for one scan-line.

3.2.3 Scan-line to Pixel Conversion Unit

The scan-line to pixel conversion unit decomposes a scan-line into its constituent pixels. For each pixel, five parameters x, y, u, v, w are calculated. x, y are coordinates of a pixel in the screen space. u, v, w are coordinates of a pixel in the solid texture space. The left most pixel on the scan-line is labeled number 0. To calculate the five parameters for the nth pixel on the scan-line, four variables, Y_{0p}, Y_{1p}, Y_{2p}, and Y_{3p}, must be calculated using the following equations:

$$\begin{aligned} Y_{0p} &= Y_{0s} + a_{01} \times n \\ Y_{1p} &= Y_{1s} + a_{11} \times n \\ Y_{2p} &= Y_{2s} + a_{21} \times n \\ Y_{3p} &= Y_{3s} + a_{31} \times n \end{aligned} \tag{3.11}$$

where n is an integer and $0 \leq n \leq (xsright - xsleft)$. Then u,v,w are calculated based on Y_{0p}, Y_{1p}, Y_{2p}, and Y_{3p} using the following equations:

$$\begin{aligned} u &= U_{init} + Y_{0p}/Y_{3p} \\ v &= V_{init} + Y_{1p}/Y_{3p} \\ w &= W_{init} + Y_{2p}/Y_{3p} \end{aligned} \tag{3.12}$$

x and y are calculated using the following equations:

$$\begin{aligned} x &= x_{sleft} + n \\ y &= y_s \end{aligned} \tag{3.13}$$

where n is an integer and $0 \leq n \leq (x_{sright} - x_{sleft})$.

For each scan-line, Equation 3.11, 3.12, and 3.13 can be calculated using the incremental algorithm shown in Figure 3.18. In this algorithm, x, y, Y_{0p}, Y_{1p}, Y_{2p}, and Y_{3p} are first initialized to

```
x = xsleft
y = ys
Y0p = Y0s
Y1p = Y1s
Y2p = Y2s
Y3p = Y3s
do {
  u = Uinit + Y0p / Y3p
  v = Vinit + Y1p / Y3p
  w = Winit + Y2p / Y3p
  x = x + 1
  Y0p = Y0p + a01
  Y1p = Y1p + a11
  Y2p = Y2p + a21
  Y3p = Y3p + a31
} while (x <= xsright)
```

Figure 3.18: Incremental Algorithm for Scan-line to Pixel Conversion

their corresponding values for the left most pixel on the scan-line. These initialization values are the outputs from the quad to scan-line conversion unit. Then the algorithm moves to the right one pixel at a time. For each loop iteration, x, y, Y_{0p}, Y_{1p}, Y_{2p}, and Y_{3p} are updated and the corresponding x, y, u, v, and w values are calculated. The loop ends when all pixels on the given scan-line are considered.

One of the possible data-path designs for the above algorithm is shown in Figure 3.19. The control logic and the initialization hardware are not shown in the figure. In this design, parameters, a_{01}, a_{11}, a_{21}, a_{31}, U_{init}, V_{init}, W_{init}, and y, are stored in registers and updated once per scan-line. Y_{0p}, Y_{1p}, Y_{2p}, Y_{3p}, and x are also stored in registers. The contents of these registers are initialized to Y_{0s}, Y_{1s}, Y_{2s}, Y_{3s}, and x_{sleft} respectively at the start of each scan-line calculation. For higher performance, there is one adder dedicated to increment each of these variables. Once Y_{0p}, Y_{1p}, Y_{2p}, and Y_{3p} are calculated for a given pixel, Y_{0p}/Y_{3p}, Y_{1p}/Y_{3p}, and Y_{2p}/Y_{3p} are evaluated to derive the values for u, v, and w. To achieve high performance, three pipelined dividers are used for the division operation. Since pipelined dividers are used, delay registers are needed to buffer the values of x and y.

Due to hardware resource limitations, the hardware design shown in Figure 3.19 is not implemented on TM-2. Instead, only one 4-stage pipelined divider is used in the current TM-2

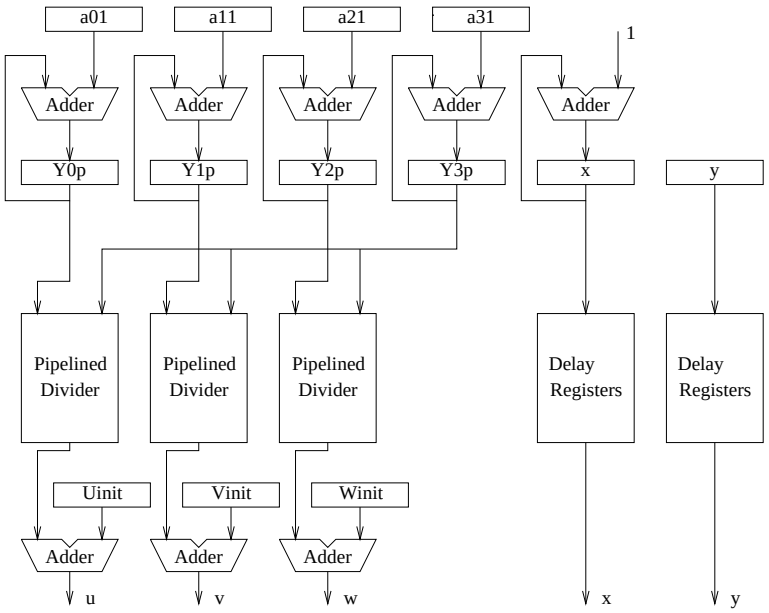

Figure 3.19: Datapath for Scan-line to Pixel Conversion

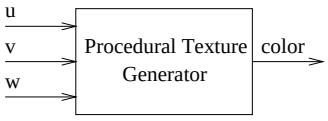

Figure 3.20: I/O Interface of the Procedural Texture Generator

implementation. The three divisions, Y_{0p}/Y_{3p}, Y_{1p}/Y_{3p}, and Y_{2p}/Y_{3p}, share the same divider in a time-multiplexed fashion.

3.3 Procedural Texture Generator

The procedural texture generator and its procedural texture algorithms are discussed in detail in chapter 4. In this section, the I/O interface of the generator is briefly discussed.

There is little difference between the I/O interface of the generator and the I/O interface of a regular texture memory. As shown in Figure 3.20, The reconfigurable solid texture generator takes in three bit-vectors, u, v, and w as its input. It generates a bit-vector representing the color as its

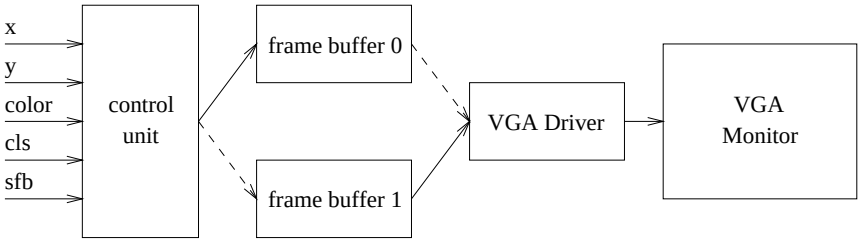

Figure 3.21: Frame Buffer

output.

3.4 Frame Buffer

The frame buffer hardware is shown in Figure 3.21. Overall, the system consists of five components, the control unit, two frame buffers, the VGA driver, and the monitor. The two frame buffers are used to implement a double buffering algorithm. While the contents of one frame buffer are being updated by the control unit, the VGA driver displays the contents of the other frame buffer on the monitor.

The input to the control unit is five bit-vectors, x, y, $color$, cls, and sfb. x and y indicate the location of the pixel to be rendered. $color$ indicates the color of the pixel to be rendered. The corresponding location in the frame buffer is updated by the control unit according to the values of x, y, and $color$. cls is the clear screen signal generated by the instruction fetching and decoding unit shown in Figure 3.14; and sfb is the switch frame buffer signal also generated by the instruction fetching and decoding unit. cls and sfb signals take precedence over the x, y, and $color$ signals. If signal cls is set to one, the control unit clears the entire content of the frame buffer to the background color. If signal sfb is set to one, the frame buffer being updated by the control unit becomes the frame buffer being read by the VGA driver, and the frame buffer being read by the VGA driver becomes the frame buffer being updated by the control unit.

Chapter 4

FPGA Implementations of Procedural Texture Algorithms

Six procedural texture algorithms have been implemented in FPGAs. Each of these algorithms takes three inputs, u, v, w. These three inputs specify a set of coordinates in a 3-D texture space. The substances that these textures model can be classified into two categories, solid and gaseous. Three textures model the coloring of solids including marble, brick, and wood. Another three model the coloring of gaseous substances including fog, fire, and cloud. Despite the difference in appearances, all six textures are fractal in nature — they all use the Perlin noise function to create fractal effects. In software, these algorithms are implemented in IEEE floating point arithmetic. Floating point hardware, however, is expensive to implement in FPGAs. Fixed point hardware is used, instead, for minimum precision implementations. Extensive pipelining is used to maximize the throughput of the algorithms.

4.1 Fractals and the Perlin Noise Function

This section describes the FPGA implementations of fractals and the Perlin noise function.

4.1.1 Fractals

In computer graphics, fractal functions are often synthesized by summing several versions of a base function at different scales and frequencies. Figure 4.1 shows this process in one dimension. There

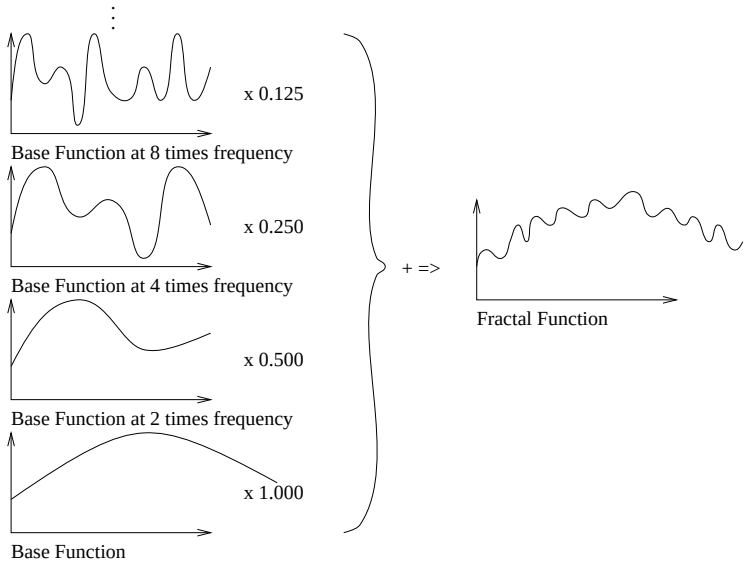

Figure 4.1: One Dimensional Fractal Function

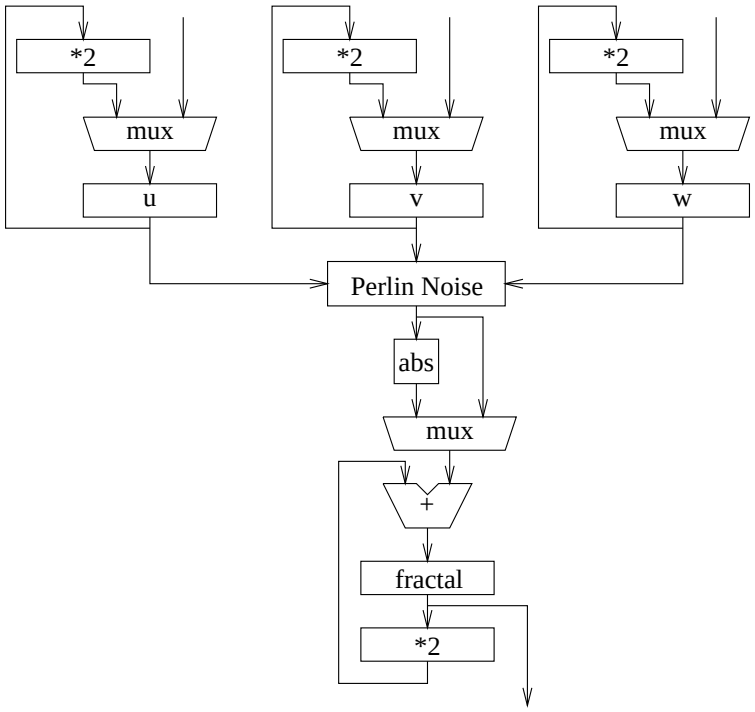

Figure 4.2: Fractal Function Hardware

are a series of functions at the left side of the figure. They are derived from the same base function by varying the frequency and the amplitude. More formally, if the base function is represented by the equation $y = P(u)$, then the base function at m times the frequency can be represented by the equation $y = P(m \times u)$. To create the fractal function, each version of the base function is scaled inversely proportional to its frequency; then all versions are summed together. Therefore, the fractal function becomes:

$$y = P(u) + \frac{1}{2} \times P(2 \times u) + \ldots + \frac{1}{m} \times P(m \times u)$$

For every new version of the base function created, the frequency is usually doubled and the scale factor is usually halved from the previous version. m is usually set to be between eight and sixty-four. A 3-D fractal function uses a base function of three variables, $P(u, v, w)$. All input variables of the 3-D base function are scaled.

Figure 4.2 shows the architecture of the fractal function in detail. In the figure, blocks u, v, w, and $fractal$ are all registers. The multiplexers and the registers are controlled by a control unit not shown in the figure. The hardware is used to implement two fractal functions, turbulence and fractalsum. These functions are defined by the following formula:

$$turbulence = \sum_{i=0}^{3} 2^{-i} P(2^i u, 2^i v, 2^i w)$$

$$fractalsum = \left| \sum_{i=0}^{3} 2^{-i} P(2^i u, 2^i v, 2^i w) \right|$$

where the function $P(u, v, w)$ represents the Perlin noise function, the actual 3-D base function used. The hardware implements the above two equations by scaling and accumulating either the value of the Perlin noise function or the absolute value of the Perlin noise function into the register labeled fractal. When the absolute value is used, the resulting fractal value is the turbulence. As the name implies, the turbulence function simulates the turbulence characteristics found in many fluids and solidified solids [EMP+94]. When the value of the Perlin noise function is directly used, the resulting function is the fractalsum function, which is often used to simulate gas formations [EMP+94]. In both cases, four cycles are needed to create one fractal value.

4.1.2 Perlin Noise Function

The Perlin noise function is one of the most computationally efficient base functions. In this thesis, a Perlin noise function of three-dimensional space was used. Informally, it can be described using

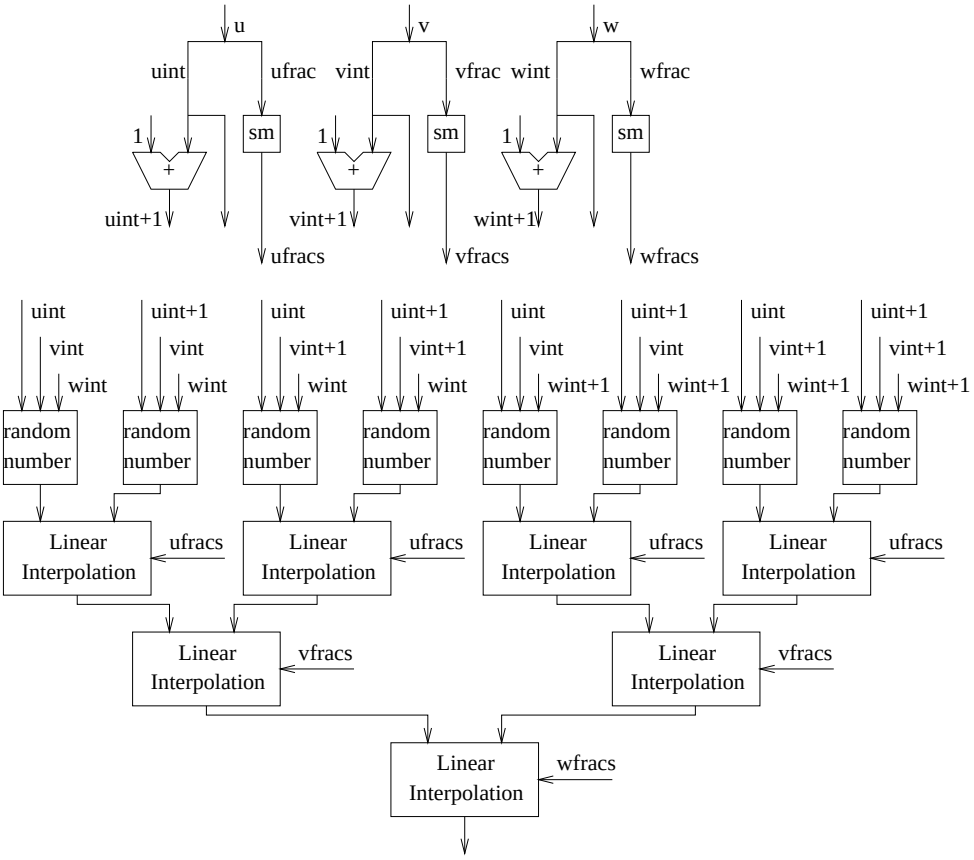

Figure 4.3: Perlin Noise Function Hardware

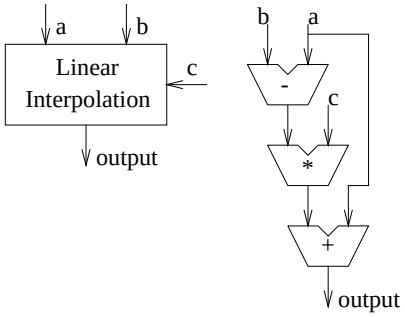

Figure 4.4: Linear Interpolation Unit

the following equation:

$$P(u,v,w) =$$
$$I(R(\lfloor u\rfloor,\lfloor v\rfloor,\lfloor w\rfloor),R(\lfloor u\rfloor,\lfloor v\rfloor,\lceil w\rceil),$$
$$R(\lfloor u\rfloor,\lceil v\rceil,\lfloor w\rfloor),R(\lfloor u\rfloor,\lceil v\rceil,\lceil w\rceil),$$
$$R(\lceil u\rceil,\lfloor v\rfloor,\lfloor w\rfloor),R(\lceil u\rceil,\lfloor v\rfloor,\lceil w\rceil),$$
$$R(\lceil u\rceil,\lceil v\rceil,\lfloor w\rfloor),R(\lceil u\rceil,\lceil v\rceil,\lceil w\rceil),$$
$$(u-\lfloor u\rfloor),(v-\lfloor v\rfloor),(w-\lfloor w\rfloor))$$

where $R(x_1,x_2,x_3)$ is a pseudo random function of its inputs; and $I(x_{000},x_{001},\ldots,x_{111},x_u,x_v,x_w)$ is an interpolation function in three dimensions. This calculates the function value on the 8 corners of a grid cell, and performs interpolation based on the associated values of the eight and the distance between the point in question and each of these grid points [Per85].

The original Perlin noise function, as actually proposed by Ken Perlin, implements the function $R(x_1,x_2,x_3)$, as three tables of 256 pre-generated pseudo random numbers stored in memory and two adders [EMP+94]. This method can consume quite large amounts of memory, since multiple copies of $R(x_1,x_2,x_3)$ are needed to fully exploit the parallelism available. A more efficient hardware method of generating pseudo random function values using xor tables [Rau91] is used in this study. This method provides significant saving in hardware.

The second improvement made to the original Perlin noise function for hardware implementation is to the interpolation method, $I(x_{000},x_{001},\ldots,x_{111},x_u,x_v,x_w)$. The original function uses an computationally expensive wavelet interpolation method [EMP+94]. This method has some superior statistical properties than the ordinary 3-D linear interpolation method; however, it is much more computationally expensive. In this study a smoothing function, $sm(x)=3x^2-2x^3$, is used to remove any second order discontinuities that might result from the linear interpolation process. The interpolation function $I(x_{000},x_{001},\ldots,x111,x_u,x_v,x_w)$ becomes
$L(x_{000},x_{001},\ldots,x_{111},sm(x_u),sm(x_v),sm(x_w))$, where $L(\ldots)$ is the linear interpolation function. By adding this smoothing function, the image quality of the 3-D linear interpolation is much improved. The hardware consumption is still much lower than the wavelet method.

Figure 4.3 shows the Perlin noise hardware. The inputs are u, v, w. The fraction, floor and ceiling values of each input are first calculated and are denoted by $ufrac$, $vfrac$, $wfrac$, $uint$, $vint$, $wint$, $uint+1$, $vint+1$, $wint+1$, respectively. The function, $R(x_1,x_2,x_3)$, is implemented by

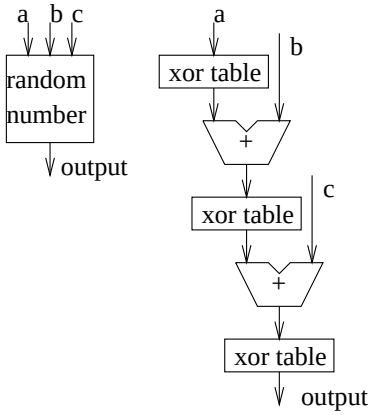

Figure 4.5: Random Number Generator

blocks, labeled *random number*. The function, $I(x_{000}, x_{001}, \ldots, x_{111}, x_u, x_v, x_w)$, is implemented by blocks, labeled *sm* and *linear interpolation*. $ufrac$, $vfrac$, and $wfrac$ are processed by the smoothing function, *sm*. The smoothing function implements the equation $sm(x) = 3x^2 - 2x^3$ in 10K50 EAB memory blocks [Alt96]. The outputs of the smoothing function are denoted by $ufracs$, $vfracs$, and $wfracs$.

The internal structure of the *linear interpolation* units is shown in Figure 4.4. Each unit implements the function $f(a, b, c) = a + c \times (b - a)$. This is a special case of the general linear interpolation formula, $g(x) = g(x_0) + \frac{g(x_1)-g(x_0)}{x_1-x_0}(x - x_0)$, where $g(x_0) = a$, $g(x_1) = b$, $x_1 - x_0 = 1$, and $x - x_0 = c$. The input, c, must be a positive fraction value between 0 and 1. a and b are real numbers.

The internal structure of the *random number* unit is shown in Figure 4.5. For a given set of inputs, the unit outputs a corresponding pseudo random number. The xor tables shown in Figure 4.5 execute the function:

$$y_0 = (x_0 \; and \; r_{00}) \; xor \; \ldots \; xor \; (x_n \; and \; r_{0n})$$
$$y_1 = (x_0 \; and \; r_{10}) \; xor \; \ldots \; xor \; (x_n \; and \; r_{1n})$$
$$\ldots$$
$$y_n = (x_0 \; and \; r_{n0}) \; xor \; \ldots \; xor \; (x_n \; and \; r_{nn})$$

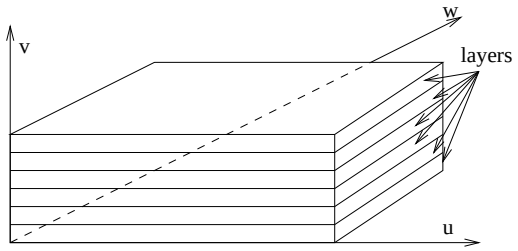

Figure 4.6: Marble Internal Structure

where $(y_n, y_{n-1}, \ldots, y_0)$ is the output bit vector, $(x_n, x_{n-1}, \ldots, x_0)$ is the input bit vector and

$$(r_{00}, r_{01}, \ldots, r_{0n})$$
$$(r_{10}, r_{11}, \ldots, r_{1n})$$
$$\ldots$$
$$(r_{n0}, r_{n1}, \ldots, r_{nn})$$

is a set of pre-generated constant bit vectors [Rau91]. Since r_{ij} is static, the entire *xor* table can be implemented in approximately eight 4-input LUTs. This is much less expensive than a 256×8 RAM. The *xor* table is used to scramble its input bits into a random value. This scrambling process is repeated three times to produce a random value for any point in space.

4.2 Perlin Noise Based 3-D Procedural Textures

4.2.1 Solid Type Procedural Textures

This section discusses the implementation of marble, wood and brick textures. All use the turbulence fractal function.

Marble

The marble algorithm models the internal coloring of marble. As illustrated in Figure 4.6, marble is formed by layers of colored rock deposits. Over time, different colored layers start to intermix with each other because of the extremely high pressure and the geological movements. This process generates the unique vein-like coloring inside the marble. This phenomenon is modeled by the

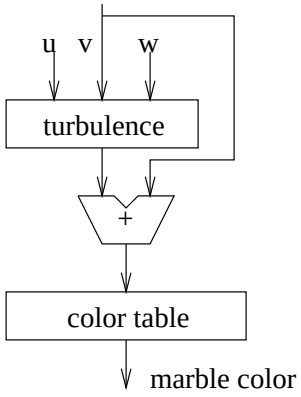

Figure 4.7: Procedural Texture Generator Configuration for the Marble Texture

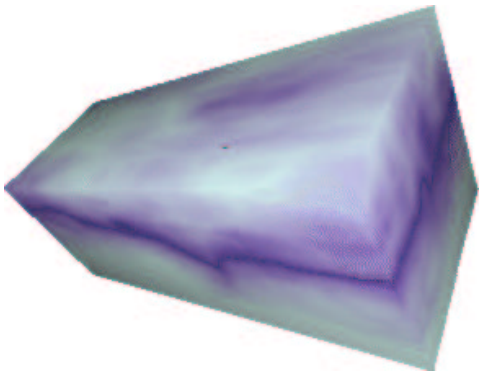

Figure 4.8: Marble Texture Mapped Cube

function:

$$M(u, v, w) = (turbulence(u, v, w) + v) \ mod \ 128$$

$M(u, v, w)$ is used to index into a color table of 128 entries. The color table is configured to store the color of the various rock layers. Each color table entry represents the color of one layer; and the address of the entry corresponds to the layer position. When the color table is accessed according to v, the resulting 3-D texture image corresponds to the unmixed layers of marble. To simulate the intermixing of layers over time, the turbulence value is added to v.

The hardware for generating the marble texture is shown in Figure 4.7. The final marble texture mapped onto a cube is shown in Figure 4.8.

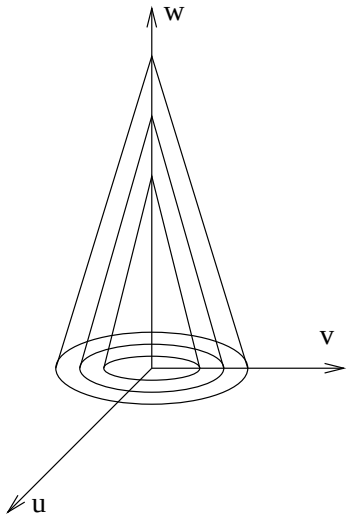

Figure 4.9: Wood Internal Structure

Wood

As illustrated by Figure 4.9, the internal structure of wood can be approximated by a series of cones that are have random perturbations. A tree grows one layer every year. The color within each layer varies with the seasons. The range of color within each layer is roughly the same from one layer to another.

Wood is modeled by the following function:

$$W(u,v,w) = ((u^2 + v^2 + \alpha w) + turbulence(u,v,w))\ mod\ 128$$

$W(u,v,w)$ is used to index into a color table of 128 entries. The range of color within a single layer is stored in the color table. The basic shape of each cone is modeled by function $u^2 + v^2 + \alpha w$, which is a hyperbolic function of u and v. The exact function for cones is $\sqrt{u^2 + v^2 + \alpha w}$. The hyperbolic function is less expensive to compute, and also models the non-uniform growth of trees, where young trees grow much faster than older ones. The *mod* operation creates the layering effect of cones. Adding turbulence to $W(u,v,w)$ simulates the irregularity of tree growth.

The hardware for calculating the wood solid texture is shown in Figure 4.10. A wood texture mapped cube is shown in Figure 4.11. Notice that the pattern is realistic and consistent across all faces of the cube. This is more clearly shown in full color prints.

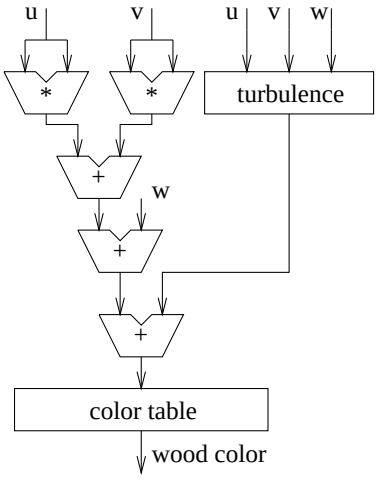

Figure 4.10: Procedural Texture Generator Configuration for the Wood Texture

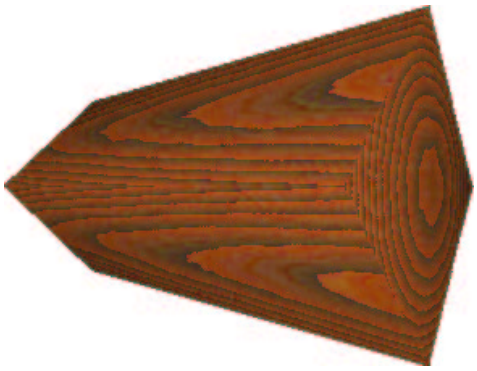

Figure 4.11: Wood Texture Mapped Cube

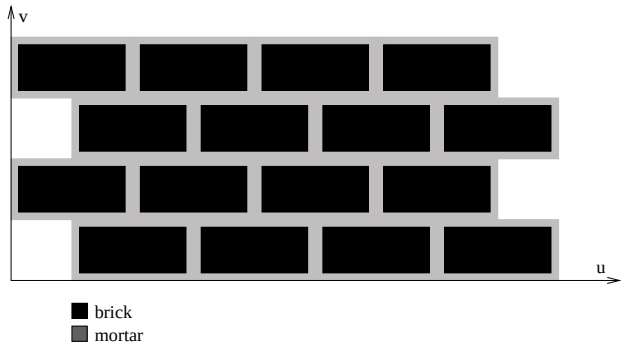

Figure 4.12: Brick Wall Pattern

Brick

The brick texture is a two-dimensional procedural texture. It uses the u, v components of the texture coordinates to determine the material at a given point in the brick wall. The brick pattern is illustrated by Figure 4.12. As shown by the figure, a brick wall is constructed by mortar and bricks. There is mortar around each brick; and there is a half brick shift from one row of bricks to the next.

The algorithm first determines the layer of the brick using the following equation:

$$\left\lfloor \frac{v}{H + 2 \times M_H} \right\rfloor$$

where H is the height of a brick, M_H is the height of each layer of mortar. If the division results in an even number, then the brick layer should be shifted to the left by half a brick width. The following equation is used to shift the brick layer:

$$u_{new} = u + \frac{W}{2} + M_W$$

where W is the width of a brick, M_W is the width of each column of mortar. If the division results in an odd number, then no shift is required and u_{new} is set to be u.

To determine if the point is in brick or mortar u_r and v_r are calculated using the following equations:

$$u_r = u_{new} \ mod \ (W + 2 \times M_W)$$
$$v_r = v \ mod \ (H + 2 \times M_H)$$

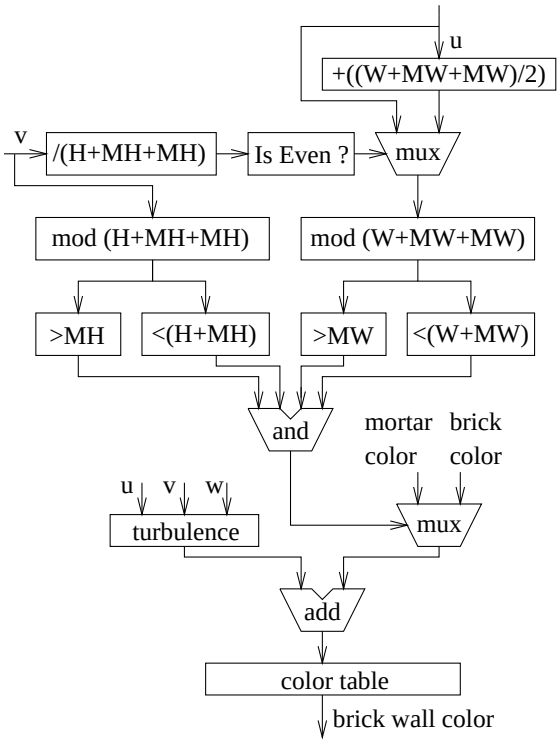

Figure 4.13: Procedural Texture Generator Configuration for the Brick Texture

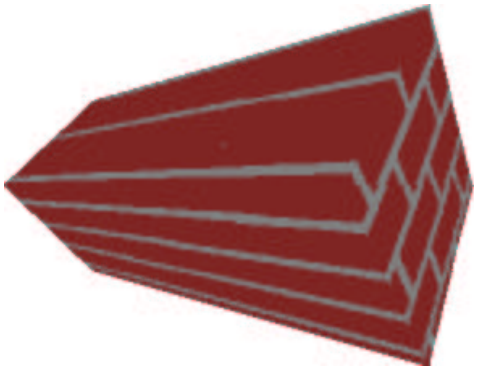

Figure 4.14: Brick Texture Mapped Cube

If $M_H < u_r < (H + M_H)$ and $M_W < v_r < (W + M_W)$, then the point is in a brick, otherwise the point is in mortar. If the point is in a brick, the turbulence value at the given point is used to index into a table of 128 pre-generated brick colors. If the point is in mortar, the same turbulence value is used to index into a table of 128 pre-generated mortar colors.

The hardware for calculating the brick texture is shown in Figure 4.13. A brick texture mapped cube is shown in Figure 4.14.

4.2.2 Gas Type Procedural Textures

The fractalsum function is often used as a base function for modeling gaseous substances. Gaseous procedural textures can be used as slides of transparencies in a rendering system. The slides can be rotated through the 3-D texture space, independent from the motion of the slides in the world space, to simulate the flow of gases [EMP+94, FHvD+90]. The key in simulating a variety of gases based on a single base function is the use of gas shaping functions [EMP+94]. The solid texture coordinates, (u, v, w), are passed through a gas shaping function to produce a new set of coordinates, (u_g, v_g, w_g). The shaped coordinates are then passed to the fractalsum function. The simplest gaseous procedural texture is the fog texture. Its shaping function is the following:

$$u_g = u$$
$$v_g = v$$
$$w_g = w$$

The r, g, b color components are then calculated using the following equations

$$r = g = b = |fractalsum(u_g, v_g, w_g)|$$

The hardware for calculating the fog texture is shown in Figure 4.15. A slide of gas texture is shown in Figure 4.16.

The cloud texture uses a shaping function of

$$u_g = u \times 2$$
$$v_g = v$$
$$w_g = w$$

The b color component is set to be 255 to simulate the color of the sky. The r, g color components are calculated based on the fractalsum function. First, $c = fractalsum(u_g, v_g, w_g)$ is calculated. If

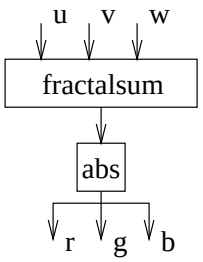

Figure 4.15: Procedural Texture Generator Configuration for the Fog Texture

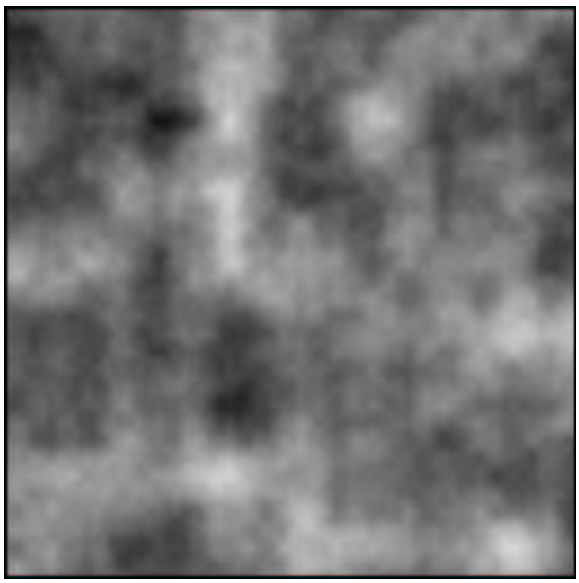

Figure 4.16: A Slide of Fog Texture

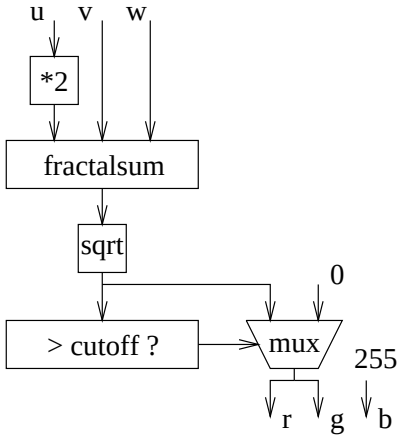

Figure 4.17: Procedural Texture Generator Configuration for the Cloud Texture

c is greater than a fixed *cutoff* value then

$$r = c$$

$$g = c$$

otherwise;

$$r = 0$$

$$g = 0$$

The *cutoff* value is used to control the density of the synthesized cloud. To be precise, the above algorithm models the stratus cloud. By varying the *cutoff* value and the shaping function, various other cloud types, including cirrus and cumulus, can also be simulated [EMP+94].

The hardware for calculating the cloud texture is shown in Figure 4.17. A slide of cloud texture is shown in Figure 4.18.

The fire texture uses a shaping function of

$$u_g = u$$

$$v_g = e^{-v}$$

$$w_g = w$$

The $fractalsum(u_g, v_g, w_g)$ is used to index into a color table of 128 entries, which stores a spectrum of flame colors.

The hardware for calculating the fire texture is shown in Figure 4.19. A slide of fire texture is shown in Figure 4.20.

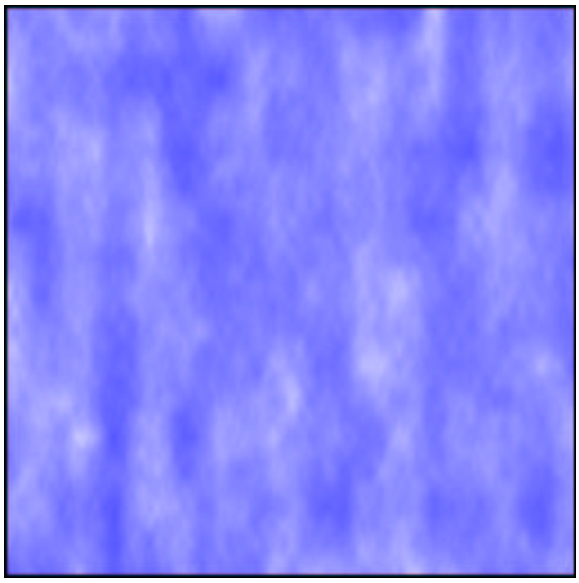

Figure 4.18: A Slide of Cloud Texture

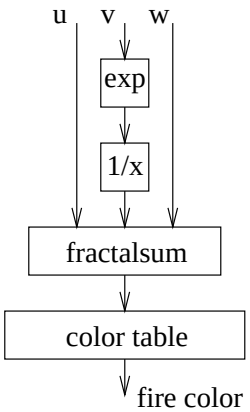

Figure 4.19: Procedural Texture Generator Configuration for Fire Texture

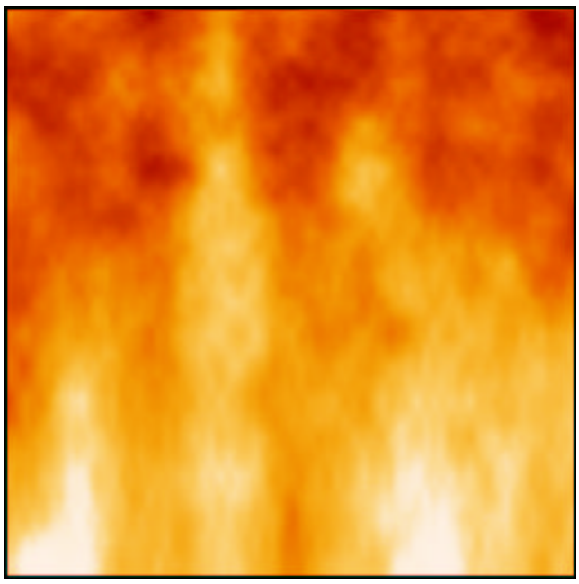

Figure 4.20: A Slide of Fire Texture

Chapter 5

Performance and Hardware Cost

The previous two chapters complete the architectural description of the rendering system. This chapter presents the performance and hardware cost data collected from the TM-2 implementation. The chapter is divided into three sections. Section one lists the performance data of the rendering system. Section two presents the hardware cost of the procedural texture generator. Section three proposes an ASIC+FPGA implementation of the procedural texture generator. In this approach, common procedural texture functions like the Perlin noise function is implemented in an ASIC while FPGAs are used to implement application specific functions for each texture.

5.1 Performance

The portion of the rendering system implemented on the TM-2 uses two clock signals. The frame buffer uses a clock frequency of 25.0 MHz. This speed is mandated by the VGA monitor that the frame buffer controls. The rest of the system uses a clock frequency of 12.5 MHz. Under the 12.5 MHz clock, the system is able to produce one pixel for every four clock cycles. The WSST software is executed on a 296MHz UltraSPARC-II CPU. The software is able to keep up with the performance of the hardware.

The performance bottleneck for the rendering system is the STST unit. When implemented on its own, the procedural texture generator can be clocked at a much higher clock frequency. When measured in isolation from the rest of the system, the execution speed of all six textures is determined by the fractal function unit. On the TM-2, the generator can run at a maximum

Textures	Look-Up Tables	Memory	Area	Area as % of 1 Gb of DRAM Area
Marble	2839	1152 bits	$47mm^2$	4.1%
Wood	3428	1152 bits	$57mm^2$	5.0%
Brick	2870	1152 bits	$47mm^2$	4.1%
Fog	2700	1152 bits	$45mm^2$	3.9%
Cloud	3006	1152 bits	$50mm^2$	4.4%
Fire	3152	5760 bits	$52mm^2$	4.5%

Table 5.1: Area Cost of Implementing Procedural Texture Generator

clock frequency of 28 MHz for all six textures, but is limited to 12.5 MHz by rest of the system. As designed, it can produce one pixel of texture for every four clock cycles. This performance is equivalent to 7 Million Pixels Per Second (MPPS). The system can fill 230K pixels per frame at 30 Hz frame rate. This performance is about 1400 times of the software performance of a turbulence function using the original Perlin noise function [EMP+94] and executed on a 296MHz UltraSPARC-II CPU. Also in [EMP+94], Ebert describes a fast version of the Perlin noise function which uses linear interpolation and direct table look-up for pseudo random number generation. The hardware still achieves 17 times speedup comparing to a turbulence function using Ebert's Perlin noise function and executed on a 296 MHz UltraSPARC-II CPU.

5.2 Hardware Cost in Comparison to Memory Based Texture Mapping

In memory based texture mapping, large amounts of memory are required to store three-dimensional texture images. In this study, 3-D procedural textures are synthesized with a resolution of $512 \times 512 \times 512$. Eight bits are used to represent the color of each pixel. Without any form of compression, each of these three-dimensional images requires 1 Gbit of storage memory. On the other hand, less than one and half 10K50 FPGAs are required to implement each texture. This section compares these two approaches to procedural texture mapping using silicon area as a yard stick.

Current state of the art technologies can package 256 Mbits of DRAM onto a $286mm^2$ die area using a $0.25\mu m$ process [WWK+96]. Using the same DRAM technologies, 1 Gbit of memory would require $1144mm^2$ of die. Altera 10k100 FPGAs are the latest implementation of the 10K50

architecture. Scaled to the same $0.25\mu m$ process, each logic array block of the 10k100 FPGAs consumes $132,000\mu m^2$ of silicon [Bet98]. This area not only includes the area consumed by the look-up tables, but also the associated routing resources for each logic array block. Since each logic array block contains eight look-up tables, each look-up table consumes approximately $16,000\mu m^2$ of silicon. Besides logic array blocks, embedded memory blocks are also used in texture synthesis. Each embedded memory block contains 2048 memory bits; and one embedded memory block is approximately the same size as one logic array block.

The amount of FPGA resources consumed by each procedural texture is shown in column two and column three of Table 5.1. Two types of resources are consumed, the look-up tables and the embedded memory blocks. The total silicon area consumed by these programmable logic resources are shown in column four. The fifth column of Table 5.1 shows the programmable logic area as a percentage of the area consumed by 1 Gbit of DRAM. For the texture algorithms investigated, the programmable logic implementations use 3.9% to 5.0% of the area required by the texture memory storing uncompressed textures of the same resolution. The FPGAs can achieve even higher area efficiency for algorithms with more input variables and larger texture spaces.

5.3 Single-Chip Graphic Accelerator with On-Chip Support for Perlin Noise based Procedural Texture Mapping

The experimental data and the wide spread use of Perlin noise function also suggest the possibility of synthesizing procedural textures in a mixture of ASIC and FPGA hardware. The combined ASIC+FPGA approach have the potential of synthesizing Perlin noise base textures at higher speed and with smaller silicon area cost. The ASIC+FPGA procedural texture generator might be small enough to fit on a single chip with the rest of the graphic accelerator. The possible floor plan for such an single-chip design is shown in Figure 5.1.

The difference between this approach and the pure FPGA approach is that the Perlin noise would be directly implemented in ASIC hardware, which has higher performance and higher logic density. Some other commonly used procedural texture functions might also be directly implemented in ASIC along with the Perlin noise. Only the remaining functions in procedural texture algorithms are required to be implemented in FPGAs. Table 5.2 shows the possible performance

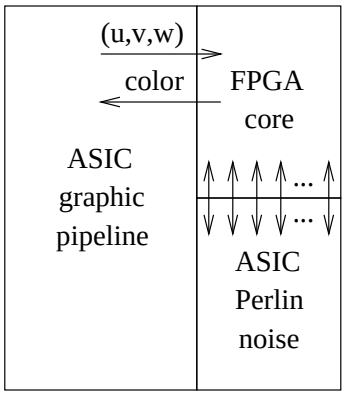

Figure 5.1: ASIC+FPGA Procedural Texture Mapping Organization

Textures	Max. Clock Freq.	MPPS	Frames Per Second
Marble	125 MHz	125	476
Wood	74 MHz	74	282
Brick	47 MHz	47	179
Fog	wiring delay	Limited by ASIC	Limited by ASIC
Cloud	43 MHz	43	164
Fire	50 MHz	50	190

Table 5.2: ASIC+FPGA Performance

figure for the ASIC+FPGA implementation for six textures investigated. Table 5.3 shows the possible area consumption by the six textures. These data are measured by removing the Perlin noise function from these six textures and measuring the speed and hardware costs of the remaining FPGA circuits. It is assumed that the ASIC implementation of the Perlin noise function is able to keep up with the performance of the FPGA circuits.

Textures	Look-Up Tables	Memory	FPGA Area
Marble	147	0 bits	$2.4mm^2$
Wood	736	0 bits	$13mm^2$
Brick	178	0 bits	$2.9mm^2$
Fog	29	0 bits	$0.47mm^2$
Cloud	335	0 bits	$5.5mm^2$
Fire	481	4608 bits	$8.2mm^2$

Table 5.3: Area cost of FPGA Hardware in ASIC+FPGA Approach

Chapter 6

Conclusions and Future Work

This thesis has presented the architecture of a 3-D computer graphic rendering system which synthesizes 3-D procedural textures in FPGA hardware. The rendering system is implemented on the TM-2 digital prototype system. The prototype system executes at a speed of 12.5 MHz and can produce pixels at a rate of 3.125 MPPS. On the TM-2 system, only 3.9% to 5.0% of the silicon area that would be consumed by the texture memory is consumed by FPGAs implementing the procedural texture generator. The implementation also shown that the procedural texture generator can achieve high performance required by the animation applications.

There are three main areas of future work. First, it is a time-consuming job to manually translate procedural texture algorithms into hardware, especially when fixed point representation is used. CAD tools need to be developed to automate most of this translation process. Second, procedural texture algorithms contain many arithmetic computations. New programmable logic architectures can be developed to target at arithmetic applications, so procedural texture algorithms can be implemented in smaller and faster programmable hardware. Third, to make the concept of synthesizing procedural texture in FPGA hardware practical, more procedural texture algorithms need to be developed. More importantly these algorithms need to be efficiently implemented in programmable logic.

Bibliography

[Alt96] Altera, *Altera 10K FPGA Databook*, 1996.

[Ath90] Peter Mark Athanas, *An Adaptive Machine Architecture and Compiler for Dynamic Processor Reconfiguration*, PhD thesis, Brown University, 1990.

[BAC96] Andrew C. Beers, Manneesh Agrawala, and Navin Chaddha, "Rendering from compressed Textures", *Computer Graphics (SIGGRAPH '96 Proceedings)*, 1996.

[BAW96] Duncan A. Buell, Jeffrey M. Arnold, and J. Water, *Splash 2: FPGAs in a custom computing machine*, IEEE Computer Society Press, Los Alamos, CA, 1996.

[Bet98] Vaughn Betz, *Architecture and CAD for Speed and Area Optimization of FPGAs*, PhD thesis, University of Toronto, 1998.

[BK70] W.J. Bouknight and K.C. Kelly, An Algorithm for Producing Half-Tone Computer Graphics Presentations with Shadows and Movable Light Sources, *SJCC*, 1970.

[BN76] J.F. Blinn and M.E. Newell, Texture and Reflection in Computer Generated Images, *CACM*, October 1976.

[Bou70] W.J. Bouknight, A Procedure for Generation of Three-Dimensional Half-Toned Computer Graphics Presentations, *CACM*, September 1970.

[BRRV92] Stephen D. Brown, J. Francis Robert, Jonathan Rose, and Zvonko G. Vranesic, *Field-Programmable Gate Arrays*, Kluwer Academic Publisher, 1992.

[BRV89] Patrice Bertin, Didier Roncin, and Jean Vuillemin, "Introduction to Programmable Active Memories", Technical report, Digital Equipment Corporation, June 1989.

[Cat74] E. Catmull, *A Subdivision Algorithm for Computer Display of Curved Surfaces*, PhD thesis, University of Utah, December 1974.

[CL96] Don Cherepacha and David Lewis, "DP-FPGA: An FPGA Architecture Optimized for Datapaths", *VLSI Design*, V4-4, pp 329–343, 1996.

[EMP+94] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Worley Steven, *Texturing and Modeling: A Procedural Approach*, AP Professional, Boston, 1994.

[FHvD+90] James D. Foley, John Hughes, van Dam, Feiner, and Hughs, *Computer Graphics: Principles and Practice*, Addison-Wesley, Reading, Mass, second edition, 1990.

[Gal96] David Galloway, "2-D Texture Mapping on TM-2", Technical report, University of Toronto, 1996.

[Gle87] James Gleick, *Chaos: Making a New Science*, Penguin Books, New York, 1987.

[HE89] D. R. Godel Hofstadter and Bach Escher, *An Eternal Golden Braid*, Vintage Books, New York, 1989.

[Hog] John Hoggard, "Fractal Geometry", http://www.math.vt.edu/people/hoggard/FracGeomReport.

[Kat90] Randy H. Katz, *Contemporary Logic Design*, Addison-Wesley Pub Co, 1990.

[LGI+98] David M. Lewis, David R. Galloway, Marcus van Ierssel, Jonathan Rose, and Paul Chow, "The Transmogrifier-2: A 1 Million Gate Rapid Prototyping System", *IEEE Trans. VLSI Systems*, pp 188–198, June 1998.

[PB92] Jean Vuillemin Patrice Bertin, Didier Roncin, "Programmable Active Memories: A Performance Assessment", Technical report, Digital Equipment Corporation, February 1992.

[Pea85] D.R. Peachey, "Solid Texturing of Complex Surfaces", *Computer Graphics (SIGGRAPH '85 Proceedings)*, pp 279–286, 1985.

[Per85] Ken Perlin, "An Image Synthesizer", *Computer Graphics (SIGGRAPH '85 Proceedings)*, V19, pp 287–296, July 1985.

[PS88] Heinz-Otto Peitgen Peitgen and Dietmar Saupe, *The Science of Fractal Images*, Springer Verlag, 1988.

[R94] Jeschke R, "An FPGA-Based Reconfigurable Coprocessor for the IBM PC", M.A.Sc, University of Toronto, 1994.

[Rau91] B. Ramakrishna Rau, "Pseudo-Randomly Interleaved Memory", *ACM*, 1991.

[Raz94] Rahul Razdan, *PRISC: Programmable Reduced Instruction Set Computers*, PhD thesis, Harvard University, May 1994.

[RBS94] Rahul Razdan, Karl Brace, and Michael D Smith, "PRISC Software Acceleration Techniques", Technical report, Digital Equipment Corporation and Harvard University, 1994.

[RS94] Rahul Razdan and Michael D Smith, "A High-Performance Microarchitecture with Hardware-Programmable Functional Units", In *MICRO-27*, Novermber 1994.

[RV92] Sriram Rajamani and Pramod Viswanath Viswanath, "Accelerating the RISC processor using Programmable Logic", Technical report, University of Berkely, 1992.

[SH74] I.E. Sutherland and G.W. Hodgman, Reentrant Polygon Clipping, *CACM*, January 1974.

[Wat70] G.S. Watkins, *A Real Time Visible Surface Algorithm*, PhD thesis, University of Utah, June 1970.

[Waw] John Wawrzynek, "BRASS Research Group Home Page", http://www.cs.berkeley.edu/Research/Projects/brass/.

[Wei98] Eric W. Weisstein, *CRC Concise Encyclopedia of Mathmatics*, CRC Press, 1998.

[Wit95] Ralph D. Witting, "OneChip: an FPGA Processor with Reconfigurable Logic", M.A.Sc, University of Toronto, 1995.

[WREE67] C. Wylie, G.W. Romney, D.C. Evans, and A.C. Erdahl, Halftone Perspective Drawings by Computer, *FJCC*, 1967.

[WWK+96] Yohji Watanabe, Hing Wong, Toshiaki Kirihata, Daisuke Kato, John K. DeBrosse, Takahiko Hara, Munehiro Yoshida, Hideo Mukai, Khandker N. Quader, Takeshi Nagai, Peter Poechmueller, Peter Pfefferl, Matthew R. Wordeman, and Shuso Fujii, "A $286mm^2$ 256Mb DRAM with x 32 Both-Ends DQ", *IEEE Journal of Solid-State Circuits*, V31, pp 567, April 1996.