TRANSFORMATION OF RENDERING ALGORITHMS FOR HARDWARE IMPLEMENTATION

Ph.D. Thesis by
Ali Mohamed Ali Abbas

scientific supervisor
Professor Dr. Szirmay-Kalos László

Faculty of Electrical Engineering and Informatics
Budapest University of Technology and Economics

Budapest, 2002.

Contents

List of Figures

Common abbreviations and notations

BRDF (f_r)	**B**i-Directional **R**eflection and **R**efraction **F**unction
CAD	**C**omputer **A**ided **D**esign
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
VHDL	**H**ardware **D**escription **L**anguage
x, y, z	Local or world coordinates of x, y and z respectively
X, Y, Z	Screen coordinates of X, Y and Z respectively
R, G, B	Tristimulus color values, stands for red, green and blue respectively
$\vec{L}$	Lighting direction
$\vec{N}$	Surface normal
$\vec{R}$	Reflection direction of $\vec{L}$ onto $\vec{N}$
$\vec{V}$	Viewing direction
$\vec{H}$	Halfway vector between $\vec{L}$ and $\vec{V}$
$\vec{x}$	Surface point
I^a	Ambient intensity
$I^e(\vec{x}, \vec{V})$	Surface self-emission
$I(\vec{x}, \vec{V})$	Radiance function of point $\vec{x}$ into direction $\vec{V}$
I_l^{in}	Incoming radiance generated by light source l
I^{out}	Outgoing radiance
$h(\vec{x}, \vec{L})$	Visibility function
$\mathcal{T}$	Light transport operator
Ω	Directional hemisphere
$d\omega_{\vec{L}}$	Differential solid angle at $\vec{L}$
$\theta_{\vec{L}}$	The angle between $\vec{L}$ and $\vec{N}$
δ	The angle between $\vec{N}$ and $\vec{H}$
ψ	The angle between $\vec{R}$ and $\vec{V}$
k_a	Ambient reflection parameter
k_d	Diffuse reflection parameter
k_s	Specular reflection parameter
t	Running variable

Preface

Computer graphics basically aims at rendering complex virtual world models and presenting images on a computer screen. To obtain an image of a virtual world, surfaces visible in pixels should be determined, and the rendering equation is used to calculate the color values of the pixels. The rendering equation, even in its simplified form, contains a lot of complex operations, including the computation of the vectors, their normalization and the evaluation of the output radiance, which makes the process rather resource demanding. A real-time animation system has to generate at least 15 images per second to provide the illusion of continuous motion. Suppose that the number of pixels on the screen is about 10^6 (advanced systems usually have $1280 * 1024$ resolution). Thus the maximum average time to manipulate a single pixel, which might include visibility and rendering calculations, cannot exceed the following limit: $1/(15 * 10^6) \approx 66\ nsec$. Since this value is comparable to a few commercial memory read or write cycles, processors which execute programs by reading the instructions and the data from memories are far too slow for this task, thus special solutions are needed. One alternative is the hardware realization, i.e. the design of a special digital network.

Hardware realization requires the original algorithms to be transformed to use only simple operations that are supported by the hardware elements. The idea behind this is to carry out the expensive computations just for a few points or pixels, and the rest can be approximated from these representative points by much simpler expressions using incremental evaluation. One way of doing this is the tessellation of the original surfaces to polygon meshes and using the vertices of the polygons as representative points. These techniques are based on linear (or in the extreme case, constant) interpolation. These methods are particularly efficient if the geometric properties can also be determined in a similar way, connecting incremental shading to the incremental visibility calculations of polygon mesh models.

Of course, when speeding up the algorithms, we cannot allow significant decrease of the realism. For example, the jaggies, which are common in all raster graphics systems should be reduced, which is called the anti-aliasing. The surfaces usually do not have constant material properties, but patterns or textures may appear. Such phenomena should also be handled by the graphics system, which is the area of texture mapping. Sometimes it is not enough to compute only the direct reflection of the light, but multiple reflections should also be taken into account. The family of algorithms that are capable of doing this is called global illumination.

This thesis contributes to the state of the art of rendering by proposing new rendering algorithms that overcome the drawbacks of linear interpolations and are comparable in image quality with the already known sophisticated techniques but allow for simple hardware implementation. These algorithms include the filtered line drawing, Phong shading, texture mapping

and ray-bundle based global illumination.

The thesis is organized as follows:

- *Chapter 1* is an introductory part to image synthesis process.
- *Chapter 2* is a survey of various functions on scan-lines and triangles.
- *Chapter 3* focuses on some line drawing algorithms, application of some filtering techniques and introduces a new approach called "*incremental cone-filtering lines"*.
- *Chapter 4* is a survey of linear interpolation on triangles, visibility calculations based on the z-buffer algorithm and an overview of constant and Gouraud shading methods.
- *Chapter 5* is an overview of Phong shading and its related methods, such as normals shading, dot product, angular shading, etc.
- *Chapter 6* introduces a new shading algorithm based on *"spherical interpolation"* as an alternative to Phong shading.
- *Chapter 7* introduces a new method called *"quadratic shading"*.
- *Chapter 8* is an overview of texture mapping and includes the application of our new method *"quadratic interpolation"* for this task.
- *Chapter 9* focuses on ray-bundle global illumination and its hardware implementation.
- *Chapter 10* contains the conclusions and the summary of the new results.

Acknowledgments

In the name of God, I wish to express my sincere gratitude to all who contributed their time, talent, knowledge, support and encouragement, for the completion of this work, in particular to:

My advisor and scientific supervisor, *Professor Dr. Szirmay-Kalos László*, for his advice, support and encouragement.

Dr. Horváth Tamás and Mr. Fóris Tibor, for providing the valuable knowledge on programming and simulation environments.

Professor Dr. Arató Péter (Head of the Department and Dean of the Faculty), for his encouragement and offering the pleasant research atmosphere.

Academic, technical and administrative staff at the Department of Control Engineering and Information Technology, for their assistance during my study period.

Members in charge at the Faculty: *Professor Dr. Selényi Endre* (Vice-Dean of scientific affairs), *Dr. Harangozó, József* (Ph.D. course director) and *Dr. Zoltai József* (Vice-Dean of student affairs), for their excellent cooperation during my study period.

Members in charge at the International Education Center: *Dr. Gyula Csopaki* (Director), *Dr. Lógó János* (Libyan students affairs coordinator) and *Mrs. Nagy Margit* (administrative coordinator), for their excellent cooperation during my study period.

My parents, my wife and my five children, for their encouragement, support and patience.

Libyan society, for the financial support.

Budapest, 2002. Ali Mohamed Ali Abbas

Chapter 1

Introduction

The objective of **image synthesis** or **rendering** is to provide the user with the illusion of watching real objects on the computer screen. The image is generated from an internal model that is called the **virtual world**. To provide the illusion of watching the real world, the color sensation of an observer looking at the artificial image generated by the graphics system must be similar to the color perception which would be obtained in the real world (Figure 1.1).

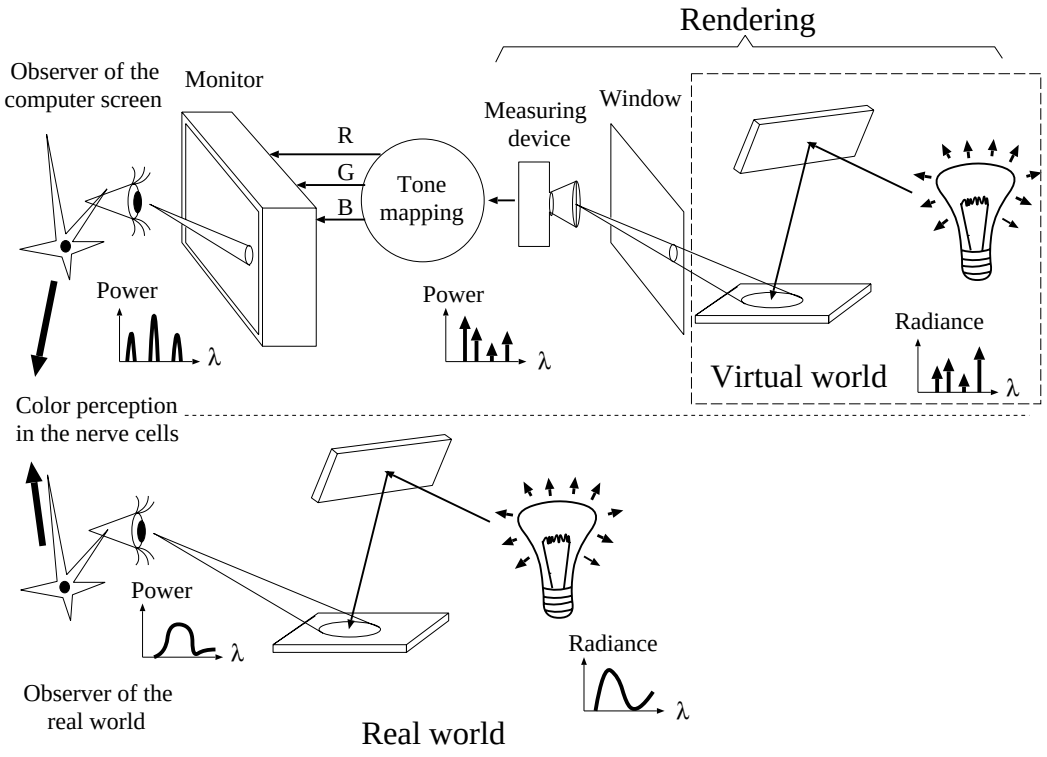


Figure 1.1: Tasks of rendering

The color perception of humans depends on the **light power** reaching the eye from a given

direction. The power, in turn, is determined from the **radiance** [SK95] of the visible points. The radiance depends on the shape and optical properties of the objects and on the intensity of the light sources.

The image synthesis uses an internal model consisting of the **geometry of the virtual world**, **optical material properties** and the description of the **lighting** in the scene. From these, applying the laws of physics (e.g. rendering equation) the real world optical phenomena can be simulated to find the light distribution in the scene.

The **rendering equation** [Kaj86] describes the light-material interaction on a single wavelength and has the following form:

$$I(\vec{x},\vec{V}) = I^e(\vec{x},\vec{V}) + (\mathcal{T}I)(\vec{x},\vec{V}), \tag{1.1}$$

where $I(\vec{x},\vec{V})$ is the radiance function at surface point $\vec{x}$ when looking from viewing direction $\vec{V}$, $I^e(\vec{x},\vec{V})$ is the self-emission and $\mathcal{T}$ is an integral operator called light transport operator that is responsible for calculating a single reflection of the light. This equation expresses the radiance of a surface as a sum of its own emission $I^e(\vec{x},\vec{V})$ and the reflection of the radiances of those points that are visible from here $(\mathcal{T}I)(\vec{x},\vec{V})$. To find the possible visible points, all incoming directions should be considered and the other contribution of the directions should be summed, which is done by the light transport operator:

$$(\mathcal{T}I)(\vec{x},\vec{V}) = \int_{\Omega} I(h(\vec{x},\vec{L}),-\vec{L}) \cdot f_r(\vec{L},\vec{x},\vec{V}) \cdot \cos\theta_{\vec{L}}\, d\omega_{\vec{L}}, \tag{1.2}$$

where Ω is the directional hemisphere, $h(\vec{x},\vec{L})$ is the visibility function defining the point that is visible from point $\vec{x}$ at illumination or lighting direction $\vec{L}$, $f_r(\vec{L},\vec{x},\vec{V})$ is the bi-directional reflection/refraction function (BRDF for short), $\theta_{\vec{L}}$ is the angle between direction vector $\vec{L}$ the surface normal $\vec{N}$, and $d\omega_{\vec{L}}$ is the differential solid angle at direction $\vec{L}$ (Figure 1.2).

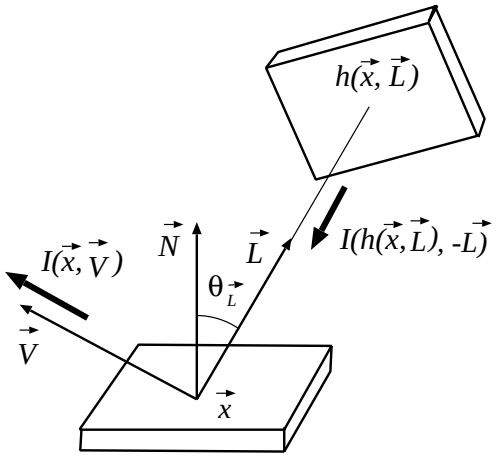


Figure 1.2: Geometry of the rendering equation

BRDFs define the optical material properties of the surfaces. Some materials are dull and reflect light dispersely and about equally in all directions (diffuse reflections); others are shiny and reflect light only in certain directions relative to the viewer and light source (specular reflections).

First of all, consider diffuse — optically very rough — surfaces reflecting a portion of the incoming light with radiance uniformly distributed in all directions. Looking at the wall, sand, etc. the perception is the same regardless of the viewing direction (Figure 1.3). If the BRDF is independent of the viewing direction, it must also be independent of the light direction because of the Helmholtz-symmetry [Min41], thus the BRDF of these **diffuse surfaces** is constant on a single wavelength:

$$f_{r,\text{diffuse}}(\vec{L}, \vec{V}) = k_d, \tag{1.3}$$

where k_d is the diffuse reflection parameter, $\vec{L}$ is the direction of the incident light, and $\vec{V}$ is the viewing direction.

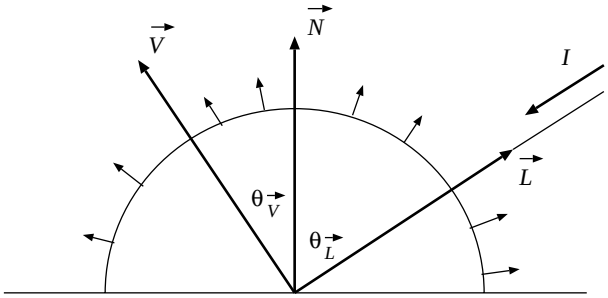


Figure 1.3: Diffuse reflection

Specular surfaces reflect most of the incoming light around the ideal reflection direction $\vec{R}$, which is the mirror direction of lighting direction $\vec{L}$ onto surface normal $\vec{N}$, thus the BRDF should be maximum at this direction and should decrease sharply (Figure 1.4).

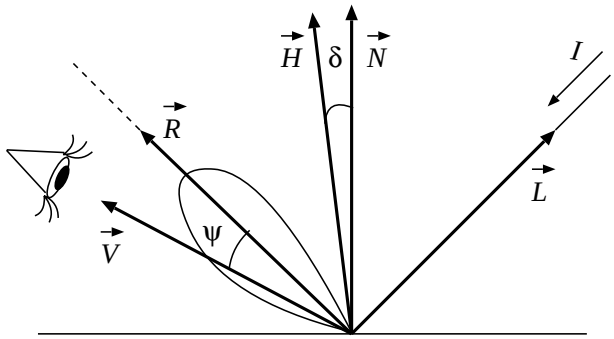


Figure 1.4: Specular reflection

The **Phong BRDF** [Pho75] was the first model proposed for specular materials, which uses the $\cos^n$ function for this purpose, thus the BRDF is the following:

$$f_{r,\mathrm{Phong}}(\vec{L},\vec{N},\vec{V}) = k_s \cdot \frac{\cos^n \psi}{\cos\theta_{\vec{L}}} = k_s \cdot \frac{(\vec{R}\cdot\vec{V})^n}{(\vec{N}\cdot\vec{L})}, \tag{1.4}$$

where k_s is the specular reflection parameter, $\vec{R}$ is the mirror direction of $\vec{L}$ onto the surface normal $\vec{N}$, n is the shininess parameter, and $\vec{L}$, $\vec{N}$, $\vec{R}$ and $\vec{V}$ are supposed to be unit vectors.

Blinn [Bli77] proposed an alternative to this BRDF, which has the following form:

$$f_{r,\mathrm{Blinn}}(\vec{L},\vec{N},\vec{V}) = k_s \cdot \frac{\cos^n \delta}{\cos\theta_{\vec{L}}} = k_s \cdot \frac{(\vec{N}\cdot\vec{H})^n}{(\vec{N}\cdot\vec{L})}, \tag{1.5}$$

where $\vec{H}$ is the halfway unit vector between $\vec{L}$ and $\vec{V}$ defined as:

$$\vec{H} = \frac{\vec{L}+\vec{V}}{|\vec{L}+\vec{V}|}. \tag{1.6}$$

Unlike Phong and Blinn models, which are only empirical constructions, Cook-Torrance BRDF [CT81] is derived from physical laws and from the statistical analysis of the microfacet structure of the surface and results in the following formula:

$$\begin{aligned} f_{r,\mathrm{Cook}}(\vec{L},\vec{N},\vec{V}) &= \frac{P(\vec{H})\cdot F(\lambda,\vec{H}\cdot\vec{L})}{4\cdot(\vec{N}\cdot\vec{L})\cdot(\vec{N}\cdot\vec{V})} \\ &\cdot \min\left\{2\cdot\frac{(\vec{N}\cdot\vec{H})\cdot(\vec{N}\cdot\vec{V})}{(\vec{V}\cdot\vec{H})}, 2\cdot\frac{(\vec{N}\cdot\vec{H})\cdot(\vec{N}\cdot\vec{L})}{(\vec{L}\cdot\vec{H})}, 1\right\}, \end{aligned} \tag{1.7}$$

where P is the probability density of the microfacet normals, and F is the wavelength (λ) dependent Fresnel function computed from the refraction index and the extinction coefficient of the material [SK95].

Examining these BRDF models, we can come to the conclusion that the reflected radiance formulae are relatively simple functions of dot products (i.e. cosine angles) of the pairs of unit vectors, including, for example, the light vector $\vec{L}$, viewing vector $\vec{V}$, halfway vector $\vec{H}$, normal vector $\vec{N}$, etc.

1.1 Tasks of image synthesis

Image synthesis is basically a transformation of objects from modeling space to the color distribution of the display defined by the digital image (Figure 1.6). Its techniques mostly depend on the space where the geometry of the internal model is represented. The photo is taken of the model by a “software camera”. The position and direction of the camera are determined by the user, and the generated image is displayed on the computer screen (Figure 1.5). The transformation involves the following characteristic steps:

- **Primitive decomposition**: The first step of image generation is the decomposition of objects used for modeling into points, lines, or polygons suitable for the image synthesis algorithms [Kun93]. In order to allow geometric transformations, the allowed type of objects should be limited. Suppose, for example, that the scene consists of spheres. Unfortunately, the transformation of a sphere, even if we use linear transformations, is not necessarily a sphere, which changes the type of the object and makes the calculation process complicated. In order to overcome this problem, the tessellation finds the sets of geometric objects whose type is invariant under homogeneous linear transformation. These types are the point, the line segment and the polygon. Tessellation approximates all surface types by points, line segments and polygons.

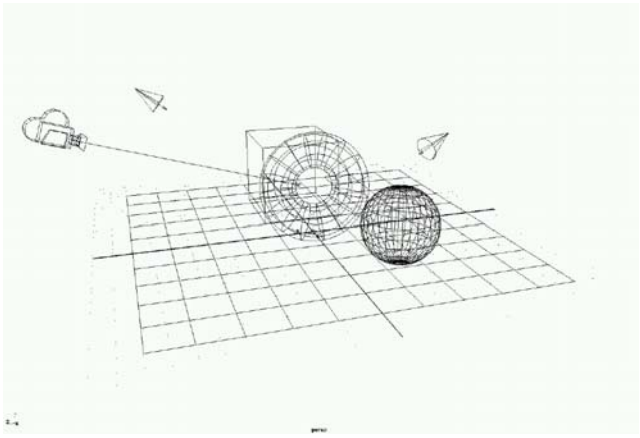
- **Transformation and clipping**: Objects are defined in a variety of local coordinate systems. However, the generated image is required in a coordinate system of the screen since eventually the color distribution of the screen has to be determined. This requires geometric transformation. On the other hand, it is obvious that the photo will only reproduce those portions of the model, which lie in the finite pyramid defined by the camera as the apex, and the sides of the $3D$ window. The process of removing those invisible parts that fall outside the pyramid is called clipping [Kuz95].

- **Rasterization, visibility computations and shading**: In the screen coordinate system the pixels that cover the projection of the objects should be identified, which is called the rasterization. Whenever the visible object is identified, its color needs to be computed using the approximated rendering equation.

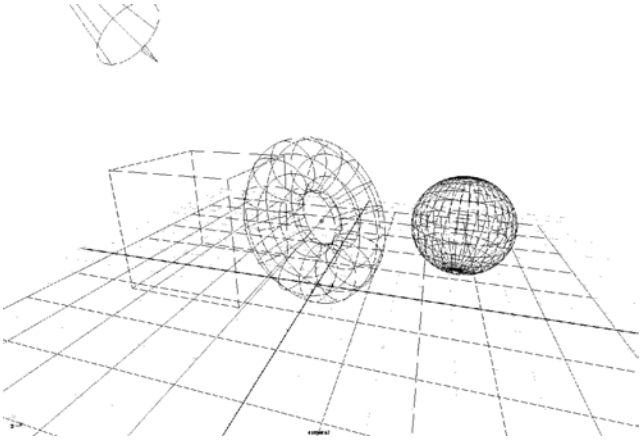
- **Tone mapping and display**: The result of the solution of the rendering equation is the radiance function sampled at different wavelengths and at different pixels. Computer screens can produce controllable electromagnetic waves, or colored light, mixed from three separate wavelengths for their observers. Thus image synthesis should compute the R, G, B intensities that can be produced by the color monitor. This step is generally referred to as tone mapping. In order to simplify this process, the rendering equation is solved just only for three wavelengths, which directly correspond to the wavelengths of the red, green and blue phosphors. We have to note that this is only an approximation, but due to the fact that it can eliminate the tone mapping operation, became popular in real-time graphics systems.

Note that transformation and clipping handle geometric primitives such as points, lines or polygons, while in visibility and shading computation — if it is done in image space — the primary object is the pixel. Since the number of pixels is far more than the number of primitives, the last step is critical for real-time rendering.

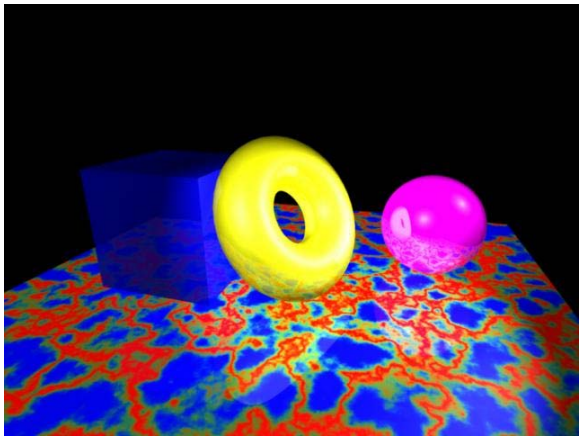
For the sake of simplicity and without loss of generality, in this thesis we assume that the polygon mesh consists of triangles only (this assumption has the important advantages that three points are always on a plane and the triangle formed by the points is convex).



The model in world coordinates



The transformed model in screen coordinates



The rendered image

Figure 1.5: The evolution of the image

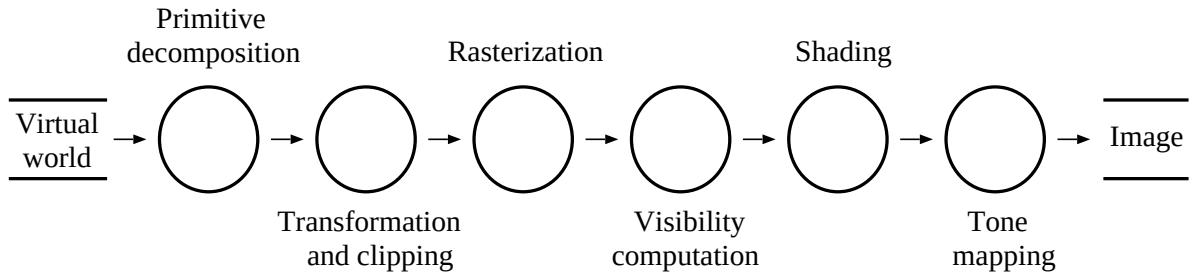


Figure 1.6: Dataflow of image synthesis

1.2 Incremental shading techniques

Incremental shading models take a very drastic approach to simplifying the rendering equation, namely eliminating all the factors which can cause multiple interdependence of the radiant intensities of different surfaces. To achieve this, they allow only non-refracting transparency (where the refraction index is 1), and reflection of the light from point, directional and ambient light sources, while ignoring the multiple reflections, i.e. the light coming from other surfaces.

The reflection of the light from light sources can be evaluated without the intensity of other surfaces, so the dependence between them has been eliminated. In fact, non-refracting transmission is the only feature left which can introduce dependence, but only in one way, since only those objects can alter the image of a given object which are behind it, looking at the scene from the camera.

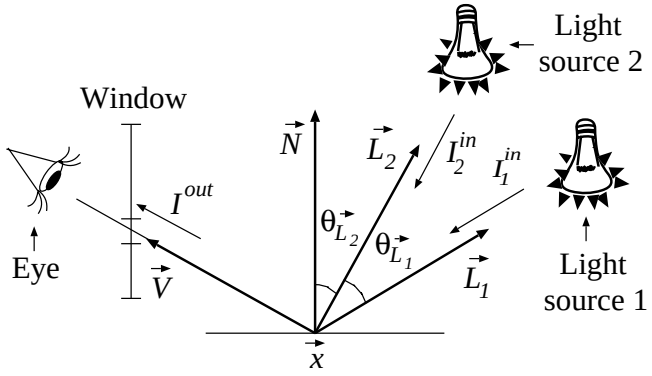


Figure 1.7: Radiance calculation in local illumination methods

If the indirect illumination coming from other surfaces is ignored and only directional and positional light sources are present (Figure 1.7), I^{in} is a Dirac-delta type function which simplifies the integral of the rendering equation to a discrete sum:

$$I^{out}(\vec{x},\vec{V}) = I^e + k_a \cdot I^a + \sum_l I_l^{in}(\vec{x},\vec{L}_l) \cdot f_r(\vec{L}_l,\vec{x},\vec{V}) \cdot \cos\theta_{\vec{L}_l} \tag{1.8}$$

where I^{out} is the outgoing radiance, I^e is the self emission, k_a and I^a are the ambient reflection parameter and the ambient intensity respectively, and I_l^{in} is the incoming radiance generated by light source l.

The radiance values are needed for each pixel, which, in turn, require the rendering equation to be solved for the visible surface. The rendering equation, even in its simplified form, contains a lot of complex operations, including the computation of the vectors, their normalization and the evaluation of the output radiance, which makes the process rather resource demanding.

1.2.1 Rasterization

The image consists of pixels, thus rasterization approximates all objects by sets of pixels. Recall that thanks to tessellation, we have to consider only point, line segment and triangle rasterization. During rasterization, we also have to take into account that many different objects may be projected onto the same pixel, thus they would be approximated by the same pixel. It must be found out which object is used to determine the color of the pixel. This step is generally referred as the **visibility computation** [Kau93].

Wire frame rendering

Wire frame rendering draws only the edges of the triangles approximating complex surfaces. Since the intersections of these edges on the scene do not significantly modify the perception of the image, the visibility computation can be ignored. Wire frame rendering is very fast, however, the images are difficult to perceive, because parts, that otherwise should be invisible, also show up. On the other hand, it is difficult to find out which parts are in front. To guide the human perception, pixels that represent points close to the observer are drawn with intensive colors. This technique, that can be interpreted as using fog in the scene, is called **depth cueing**.

Shaded rendering

Shaded rendering draws tessellated triangles including their interior not just their edges. For each pixel belonging to the projection of a triangle, the visibility problem should be solved and the color of the visible point should be computed. For the solution of the visibility problem, the z-buffer algorithm has become the most popular. This algorithm recognizes that from those patches that can be projected onto a given pixel that patch is really visible which is the closest to the eye. In order to find the patch with the minimum distance, a separate buffer is maintained which stores these distance values and patches are compared and inserted into the buffer during rendering.

The calculation of the color of the visible surfaces is called the shading. In light-surface interaction the surface illuminated by an incident beam may reflect a portion of the incoming energy in various directions or it may absorb the rest. To find the color of the surfaces, we

have to determine the incident illumination and to compute the light reflected towards the eye according to the optical properties of the surface.

This can be rather time consuming if we used it independently for each pixel. However, the distance of two close points from the camera, or their colors are usually quite similar. This makes interpolation techniques worthwhile. Interpolation can be used to speed up the rendering of the triangle mesh, where the expensive computations take place just at the vertices and the data of the internal points are interpolated. A simple interpolation scheme would compute the color and linearly interpolate it inside the triangle (Gouraud shading [Gou71]). However, specular reflections may introduce strong non-linearity, thus linear interpolation can introduce severe artifacts (left of Figure 1.9). The core of the problem is that the color may be a strongly non-linear function of the pixel coordinates, especially if specular highlights occur on the triangle, and this non-linear function can hardly be well approximated by a linear function (Figure 1.8).

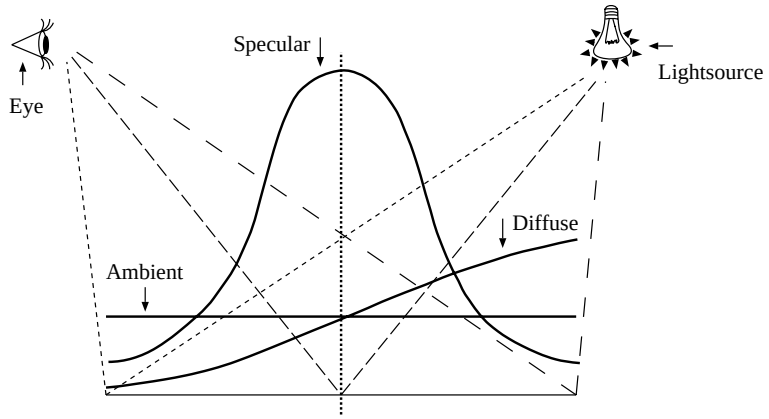


Figure 1.8: Ambient, diffuse, and specular reflections

The artifacts of Gouraud shading can be eliminated by a non-linear interpolation called Phong shading [Pho75] (right of Figure 1.9). In Phong shading, vectors used by the BRDFs in the rendering equation are interpolated from the real vectors at the vertices of the approximating triangle; the interpolated vectors are normalized and the rendering equation is evaluated at every pixel for diffuse and specular reflections and for each lightsource, which is rather time consuming. The main problem of Phong shading is that it requires complex operations on the pixel level, thus its hardware implementation is not suitable for real-time rendering.

Texture mapping

So far we have assumed the optical material properties are constant on the surfaces. This assumption does not hold in practice, but the BRDF itself is a function of the surface point. To define such a function, the surface is mapped to the unit square, called **texture space**, where the BRDF data are stored. This means that during rendering each pixel should be transformed to the texture space where the optical data are available. This method is called **texture mapping** (bottom of Figure 1.5).

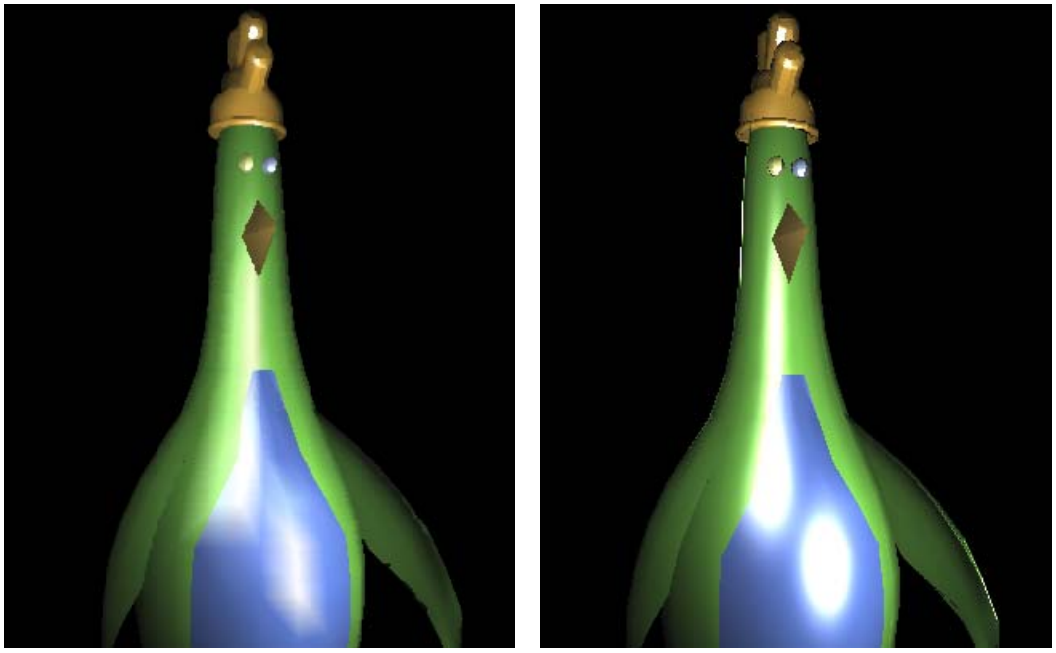


Figure 1.9: Comparison of linear interpolation i.e. Gouraud shading (left) and non-linear interpolation by Phong shading (right)

1.3 The objectives of this thesis

This thesis contributes to the state of the art of rendering by proposing new image synthesis algorithms that overcome the drawbacks of linear interpolations and are comparable in image quality with the already known, sophisticated techniques but allow for simple hardware implementation. These algorithms cover filtered line drawing, Phong shading, texture mapping and ray-bundle based global illumination.

In order to achieve these goals, the following methodology was used. First, sophisticated rendering algorithms providing the required visual quality were analyzed. Since these are too intensive computationally, their hardware realization is not feasible. Based on a general framework of using the incremental concept, the algorithms have been transformed to an equivalent or approximately equivalent form with the aim of direct hardware support [AVJ01] [Abb95].

The new algorithms have been implemented in software and their functional properties have been evaluated. Then, the algorithms also have been specified in VHDL [Ash90] [Per91] [WWD+95], which is a popular computer hardware description language, assuming the delay times according to Xilinx Synthesis Technology real FPGA device [Xil01]. The conversion allowed the direct simulation in Model-Technology environments [Inc94], and the demonstrations show that the algorithms can really provide real-time rendering.

Chapter 2

Hardware implementation of rendering functions

In incremental image synthesis all operations are applied to linear objects, including points, lines and triangles. These linear objects are the results of the tessellation process. The objects are then transformed to the screen coordinate system and clipped. Note that the reason of selecting these linear objects is that their type is invariant to these operations, i.e. a transformed and clipped line segment is also a line segment, while a transformed and clipped triangle list is also a triangle list. Suppose that these objects are already in the screen coordinate system.

Since these linear objects correspond to linear or in special cases constant functions, the rendering of these objects requires the computation of linear or constant expressions over the pixel grid. This chapter discusses how it can be realized efficiently by synchronous digital networks.

2.1 Functions on scan-lines

This section reviews the implementation strategies of simple functions that shade the pixels in a single scan-line.

2.1.1 One-variate constant functions

In this case we have to generate the sequence of X values from X_{start} to X_{end} and with each X value we have to give a constant color I. For the generation of an X series, a counter is needed that is initialized by X_{start} and a comparator to stop the counter when $X = X_{end}$.

The resulting algorithm is:

for $X = X_{start}$ **to** $X = X_{end}$ **do**
    Store I in the raster memory at X, Y;
endfor

The hardware implementation of one-variate constant functions is shown in Figure 2.1.

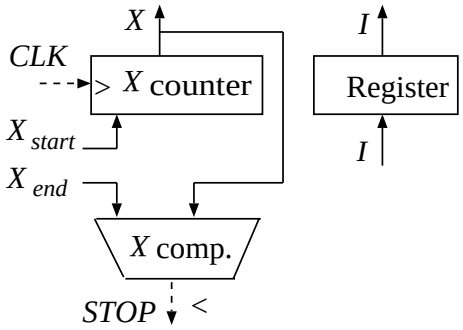


Figure 2.1: Hardware implementation of one-variate constant functions

2.1.2 One-variate linear functions

Let us consider the implementation of the following linear function:

$$I(X) = T_1 X + T_0, \tag{2.1}$$

for X from X_{start} to X_{end}.

If we implemented this directly, the hardware should compute a floating point multiplication and an addition for each value, which is rather demanding. To eliminate the multiplication, we introduce the incremental concept which computes $I(X+1)$ from the previous value $I(X)$ instead of $(X+1)$, using the following formula:

$$I(X+1) = T_1(X+1) + T_0 = T_1 X + T_0 + T_1 = I(X) + T_1.$$

Note that in this way the computation requires just a single addition.

The sequence of $I(X)$ can be generated by the following algorithm:

$I = T_1 X_{start} + T_0$;
for $X = X_{start}$ **to** X_{end} **do**
    Store I in the raster memory at X, Y;
    I += T_1;
endfor

The hardware implementation of one-variate linear functions is shown in Figure 2.2.

Note that the incremental concept traced back the computation of the multiplication to a single addition. However, function I and the parameters are not necessarily integers, and it is not possible to ignore the fractional part, since the incremental formula will accumulate the error to an unacceptable degree. The realization of floating point addition is not at all simple. Non-integers, fortunately, can also be represented in fixed point form where the low b_I bits of the code word represent the fractional part, and the high b_I bits store the integer part. From a different point of view, a code word having binary code C represents the real number $C \cdot 2^{-b_I}$.

Let us determine the length of the register needed to store I. Concerning the integer part, if I can have values from 0 to N, then $b_I > \log_2 N$ bits are needed. The number of bits in the fractional part has to be set to avoid incorrect I calculations due to the cumulative error in I. Since the maximum length of the iteration is M, $(M = \max\{X_{end} - X_{start}\})$, and the maximum error introduced by a single step of the iteration is less than 2^{-b_I}, the cumulative error is maximum $M2^{-b_I}$. Incorrect calculations of I is avoided if the cumulative error is less than 1:

$$M2^{-b_I} < 1 \quad \Longrightarrow \quad b_I > \log_2 M. \tag{2.2}$$

Since the results are expected in integer form, they must be converted to integers at the final stage of the calculation. The Round function finding the nearest integer for a real number, however, has high combinational complexity. Fortunately, the Round function can be replaced by the Trunc function generating the integer part of a real number if 0.5 is added to the number to be converted. The implementation of the Trunc function poses no problem for fixed point representation, since just the bits corresponding to the fractional part must be neglected. This trick can generally be used if we want to get rid of the Round function.

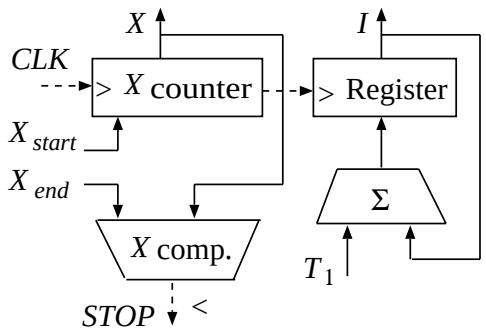


Figure 2.2: Hardware implementation of one-variate linear functions

2.1.3 One-variate quadratic functions

Let us consider the implementation of the following quadratic function:

$$I(X) = T_2X^2 + T_1X + T_0, \tag{2.3}$$

for X from X_{start} to X_{end}.

To eliminate the multiplication, we apply the incremental concept two times. First the quadratic expression is reduced to a linear one:

$$I(X+1) = T_2(X+1)^2 + T_1(X+1) + T_0 = I(X) + \Delta I(X),$$

where $\Delta I(X) = 2T_2X + T_2 + T_1$.

Then we apply the incremental concept once more for the linear function $\Delta I(X)$:

$$\Delta I(X+1) = \Delta I(X) + 2T_2.$$

The resulting algorithm is:

$I = T_2 X_{start}^2 + T_1 X_{start} + T_0$;
$\Delta I = 2T_2 X_{start} + T_2 + T_1$;
for $X = X_{start}$ **to** X_{end} **do**
    Store I in the raster memory at X, Y;
    I += ΔI;
    ΔI += $2T_2$;
endfor

The hardware implementation of one-variate quadratic functions is shown in Figure 2.3.

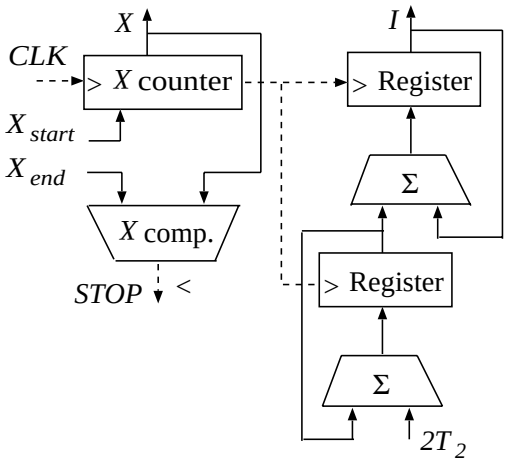


Figure 2.3: Hardware implementation of one-variate quadratic functions

2.2 Functions on triangles

This section reviews the implementation strategies of simple functions on scan-lines that are used to fill image space triangles. A complete triangle is rendered by generating those scan-lines and the pixels in these scan-lines which cover this triangle. For each scan-line, the start and end points should be identified and the interpolation parameters need to be initialized, then the scan-line interpolation can be initiated. As the hardware algorithm considers only horizontal sided triangles — if not so — the image space triangle should be divided at Y_2 coordinate to two horizontal sided parts a lower and an upper (Figure 2.4). In this section we will consider only lower horizontal sided triangle. The upper part can be handled similarly. Note that the color

is two-variate function $I(X,Y)$. If only a scan-line is considered then Y is constant and there $I(X,Y)$ will be represented by a one-variate function $I(X)$.

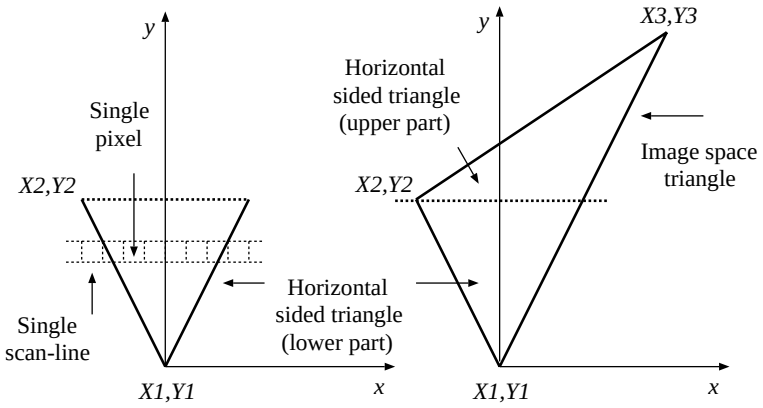


Figure 2.4: Image space triangle and horizontal sided triangle

2.2.1 Two-variate constant functions

In this case we have to generate the sequence of (X,Y) integer values called pixels that are inside a horizontal sided triangle. The algorithm generates the pixels on a scan-line by scan-line basis. In a single scan-line the Y coordinate is constant.

When we step onto the next scan-line, Y is incremented, and $X_{start}(Y)$ and $X_{end}(Y)$ coordinates should be determined by the following equations and are illustrated in a raster grid by a lower part horizontal sided triangle (Figure 2.5):

$$\begin{aligned} X_{start}(Y) &= \frac{Y-Y_1}{Y_2-Y_1}\cdot(X_2-X_1)+X_1, \\ X_{end}(Y) &= \frac{Y-Y_1}{Y_3-Y_1}\cdot(X_3-X_1)+X_1. \end{aligned} \tag{2.4}$$

Since $X_{start}(Y)$ and $X_{end}(Y)$ are linear functions, they can be simplified by applying the incremental concept:

$$\begin{aligned} X_{start}(Y+1) &= X_{start}(Y)+A_{start}, \\ X_{end}(Y+1) &= X_{end}(Y)+A_{end}, \end{aligned} \tag{2.5}$$

where

$$A_{start}=\frac{X_2-X_1}{Y_2-Y_1}, \quad A_{end}=\frac{X_3-X_1}{Y_3-Y_1}. \tag{2.6}$$

The resulting algorithm is:

$X_{start} = X_1$;
$X_{end} = X_1$;
for Y = Y_1 **to** Y_2 **do**
    for X = X_{start} **to** X_{end} **do**
        Store I in the raster memory at X, Y;
    endfor
    X_{start} += A_{start};
    X_{end} += A_{end};
endfor

The hardware implementation of two-variate constant functions is shown in Figure 2.5 .

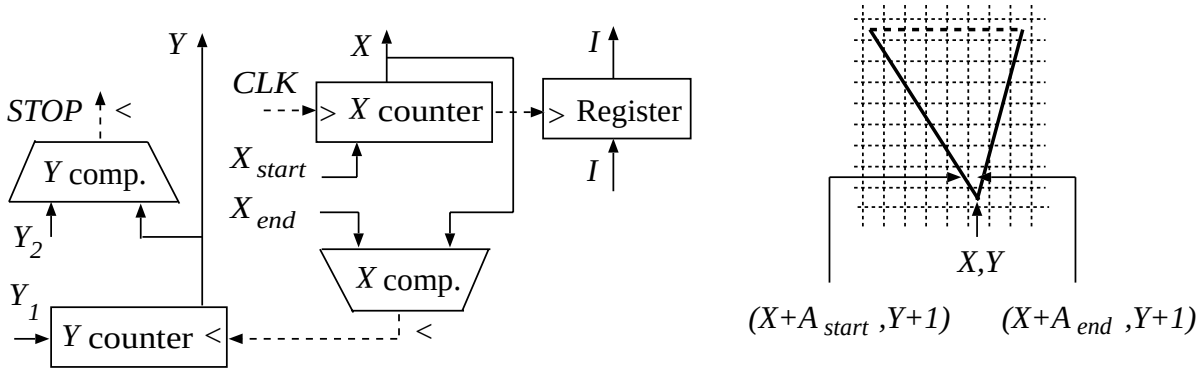


Figure 2.5: Hardware implementation of two-variate constant functions (left) and a raster grid (right)

2.2.2 Two-variate linear functions

Let us consider the implementation of the following linear function:

$$I(X, Y) = T_2 X + T_1 Y + T_0, \tag{2.7}$$

for (X, Y) pairs that are inside a horizontal sided triangle. The (X, Y) pairs are generated scan-line by scan-line as discussed in the previous section.

To eliminate the multiplication, we introduce the incremental concept of each scan-line and for their start edges:

$$\begin{aligned} I(X+1, Y) &= I(X, Y) + T_2. \\ I(X + A_{start}, Y+1) &= I(X, Y) + \Delta I(X, Y), \end{aligned}$$

where $\Delta I(X, Y) = T_2 A_{start} + T_1$.

The resulting algorithm is:

$X_{start} = X_1;$
$X_{end} = X_1;$
$I_{start} = T_2 X_{start} + T_1 Y_{start} + T_0;$
$\Delta I = T_2 A_{start} + T_1;$
for $Y = Y_1$ **to** Y_2 **do**
    $I = I_{start};$
    for $X = X_{start}$ **to** X_{end} **do**
        Store I in the raster memory at X, Y;
        I += T_2;
    endfor
    I_{start} += ΔI;
    X_{start} += A_{start};
    X_{end} += A_{end};
endfor

The hardware implementation of two-variate linear functions is shown in Figure 2.6.

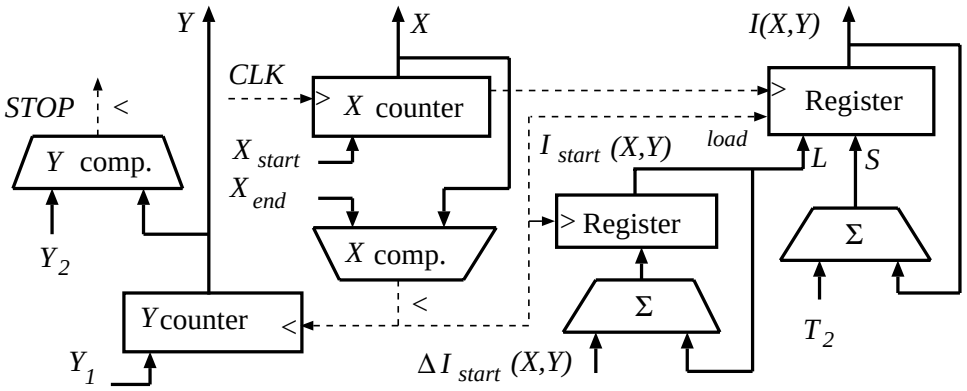


Figure 2.6: Hardware implementation of two-variate linear functions

The registers usually have two data inputs L and S. L is the input to the register when the *load* signal is active, and S is the input to the register for each clock. The clock signal of the subsystem responsible for the internal pixels of the scan-lines is the system clock. However, the clock signal controlling the elements that compute the interpolation at the start edge is the output of the comparator detecting the end of the scan-line.

2.2.3 Two-variate quadratic functions

Let us consider the implementation of the following quadratic function:

$$I(X,Y) = T_5X^2 + T_4XY + T_3Y^2 + T_2X + T_1Y + T_0. \tag{2.8}$$

To simplify Equation 2.8, we introduce the incremental concept for the scan-lines and for their start edges. First, the quadratic function is reduced to a linear one for the scan-lines:

$$I(X+1,Y) = I(X,Y) + \Delta I(X,Y),$$

where $\Delta I(X,Y) = 2T_5X + T_4Y + T_5 + T_2$.

Applying the incremental concept once more for the linear function $\Delta I(X,Y)$, we obtain the incremental value inside the scan-lines:

$$\Delta I(X+1,Y) = \Delta I(X,Y) + 2T_5.$$

When we step onto the next scan-line, Y is incremented, and the start X_{start} and the end X_{end} coordinates should be determined by Equation 2.6.

Now let us consider the computation on the start edge. The quadratic function $I(X,Y)$ will be reduced to a linear one:

$$I(X + A_{start}, Y+1) = I(X,Y) + \Delta I_{start}(X,Y),$$

where

$$\Delta I_{start}(X,Y) = T_5A_{start}^2 + (2T_5X + T_4Y + T_4 + T_2)A_{start} + T_4X + 2T_3Y + T_3 + T_1.$$

Applying the incremental concept once more for the linear function $\Delta I_{start}(X,Y)$, we obtain the incremental value on the start edges:

$$\Delta I_{start}(X + A_{start}, Y+1) = \Delta I_{start}(X,Y) + 2(T_5A_{start}^2 + T_4A_{start} + T_3).$$

To obtain the incremental value for the scan-lines at the start edges we should apply the incremental concept once more for the linear function $\Delta I(X,Y)$:

$$\Delta I(X + A_{start}, Y+1) = \Delta I(X,Y) + 2T_5A_{start} + T_4.$$

Let us group the above formulation in the following algorithm:

$X_{start} = X_1; \quad X_{end} = X_1;$
$I_{start}(X,Y) = T_5X_{start}^2 + T_4X_{start}Y_{start} + T_3Y_{start}^2 + T_2X_{start} + T_1Y_{start} + T_0;$
$\Delta I_{start}(X,Y) = T_5A_{start}^2 + (2T_5X_{start} + T_4Y_{start} + T_4 + T_2)A_{start}$
$\quad + T_4X_{start} + 2T_3Y_{start} + T_3 + T_1;$
$\Delta I(X_{start},Y) = 2T_5X_{start} + T_4Y_{start} + T_5 + T_2;$
for $Y = Y_1$ **to** Y_2 **do**
$\quad I(X,Y) = I_{start}(X,Y);$
$\quad \Delta I(X,Y) = \Delta I(X_{start},Y);$
$\quad$ **for** $X = X_{start}$ **to** X_{end} **do**
$\quad\quad$ Store $I(X,Y)$ in the raster memory at X,Y;
$\quad\quad I(X,Y)$ += $\Delta I(X,Y); \quad \Delta I(X,Y)$ += $2T_5;$
$\quad$ **endfor**
$\quad \Delta I(X_{start},Y)$ += $2T_5A_{start} + T_4;$
$\quad I_{start}(X,Y)$ += $\Delta I_{start}(X,Y);$
$\quad \Delta I_{start}(X,Y)$ += $2(T_5A_{start}^2 + T_4A_{start} + T_3);$
$\quad X_{start}$ += $A_{start}; \quad X_{end}$ += $A_{end};$
endfor

The hardware implementation of two-variate quadratic functions is shown in Figure 2.7.

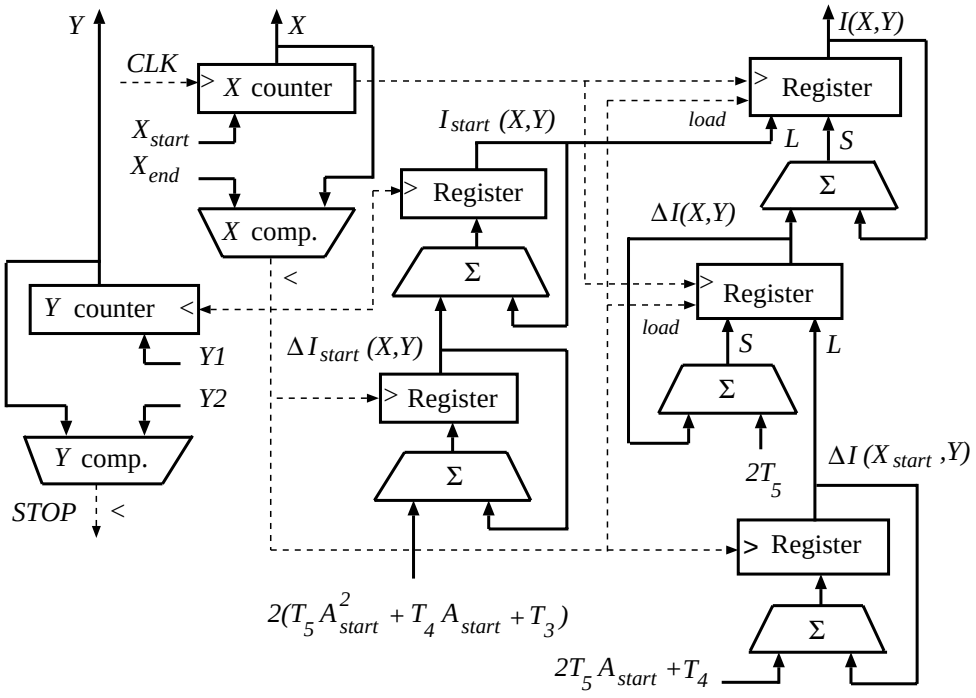


Figure 2.7: Hardware implementation of two-variate quadratic functions

As we have already discussed it in Section 2.1.2, function I and the parameters are not integers, thus we have to use fixed point representation. However, now the number of fractional bits should be determined differently. In order to obtain $I(X, Y)$ for some X, Y pixel,

$$M_Y \leq \max\{Y_2 - Y_1\}$$

iteration steps are executed on the start edge and

$$M_X \leq \max\{X_3 - X_2\}$$

steps on the horizontal span. A single iteration step involves the calculation of increments ΔI or ΔI_{start} as an addition with a constant, then the increase of I or I_{start} by the current increment values. The maximum error introduced by an addition with a constant is 2^{-b_I}, thus after m steps, the cumulative error of the increment is less than $m \cdot 2^{-b_I}$. Consequently, the cumulative error in values I and I_{start} after M steps is less than:

$$\sum_{m=1}^{M} m \cdot 2^{-b_I} = M(M-1) \cdot 2^{-(b_I+1)}. \tag{2.9}$$

Incorrect calculations of I is avoided if the cumulative error is less than 1. Since a single value requires at most M_Y steps on the start edge and M_X steps on the horizontal span, we obtain:

$$(M_X(M_X-1)+M_Y(M_Y-1)) \cdot 2^{-(b_I+1)} < 1 \implies b_I > \log_2(M_X(M_X-1)+M_Y(M_Y-1))-1. \tag{2.10}$$

If the horizontal and vertical sizes of the largest allowed triangle are 512 pixels, then this formula results in the requirement of 20 fractional bits.

Note that quadratic interpolation roughly doubles the number of fractional bits compared with linear interpolation.

Chapter 3

Drawing lines

Line drawing algorithms take very important part in the design of computer graphics software and hardware, where many images are mostly composed of line segments. The task is to identify the set of those pixels that approximate the appearance of $2D$ or $3D$ lines [Gar75]. Simple sampling algorithms would make it too obvious that the approximation consists of a set of small rectangles, creating jagged or stair-cased images. These jaggies can be eliminated by sophisticated filtering, which is called the anti-aliasing.

A scan-conversion algorithm for lines computes the coordinates of the pixels that lie on or near to an ideal, infinitely thin straight line imposed on a $2D$ raster grid. In principle, we would like the sequence of pixels to lie as close to the ideal line as possible and to be as straight as possible. If we consider one-pixel-thick approximation to an ideal line, the properties will change according to its slope. For lines with slopes between -1 and 1 inclusive, exactly one-pixel should be illuminated in each column, but for lines with slopes outside this range, exactly one-pixel should be illuminated in each row. All lines should be drawn with constant brightness, independently of its length and orientation, and as rapidly as possible [RBX90] [SKM94] [BB99].

Let the two end points of the line segment be (X_1, Y_1) and (X_2, Y_2) respectively. Then the slope of the line is:

$$m = \frac{Y_2 - Y_1}{X_2 - X_1} = \frac{\Delta Y}{\Delta X}. \tag{3.1}$$

Since we will consider only the cases of slope $0 \leq m \leq 1$, [Abb98] we have:

$$0 < Y_2 - Y_1 \leq X_2 - X_1.$$

3.1 Bresenham algorithm

In the incremental concept m must be real or fractional binary because the slope is a fraction. Bresenham developed a classic algorithm, which uses only integer arithmetic. The choice of pixels is made by testing the sign of a Discriminator based on the Midpoint principle [FvDFH90] [Che97]. The Discriminator obeys a simple recursive strategy where the chosen pixel will be the closest to the true line.

We assume that the slope of the line is between 0 and 1, where (X_1, Y_1) represents the lower-left endpoint and (X_2, Y_2) represents the upper-right endpoint.

Consider the line in Figure 3.1 where the previously selected pixel appears as black circle and the two pixels from which to choose at the next stage are shown as unfilled circles. Assume that we have just selected the pixel P at (X_P, Y_P) and now must choose between the pixel one increment to right (called the east pixel, E) or the pixel one increment to right and one increment up (called the north-east pixel, NE). Let Q be the intersection point of the line being scan-converted with the grid line $X = X_P + 1$. In Bresenham's formulation, the difference between the vertical distances from E and NE to Q is computed, and the sign of the difference is used to select the pixel whose distance from Q is smaller as the best approximation to the line. In the Midpoint formulation, we observe on which side of the line the Midpoint M lies. If M lies above the line, pixel E is closer to line, and if M lies below the line, pixel NE is closer to the line. The line may pass exactly between E and NE, or both pixels may lie on one side of the line. Also the error which is the vertical distance between the chosen pixel and the actual line is always less than a half.

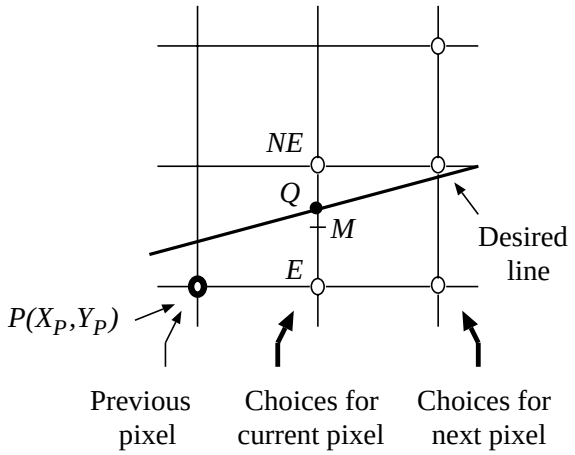


Figure 3.1: Pixel grid for Bresenham's Midpoint based line generator

Now all we need is a way to calculate on which side of the line M lies. Let us represent the line by an implicit function with coefficients a, b, and c:

$$F(X, Y) = a \cdot X + b \cdot Y + c = 0. \tag{3.2}$$

If $\Delta Y = Y_2 - Y_1$, and $\Delta X = X_2 - X_1$, the slope-intercept form can be written as:

$$Y = \frac{\Delta Y}{\Delta X} \cdot X + B,$$

therefore:

$$F(X, Y) = \Delta Y \cdot X - \Delta X \cdot Y + B \cdot \Delta X = 0, \tag{3.3}$$

here $a = \Delta Y,\ b = -\Delta X$ and $c = B \cdot \Delta X$.

It can easily be verified that $F(X, Y)$ is zero on the line, positive for points below the line, and negative for points above the line. To apply the Midpoint criterion, we need only to compute $F(M) = F(X_P + 1, Y_P + 0.5)$, and to test its sign. Because our decision is based on the value of the function at $(X_P + 1, Y_P + 0.5)$, we define a *decision variable* $dv = F(X_P + 1, Y_P + 0.5)$. By definition, $dv = a \cdot (X_P + 1) + b \cdot (Y_P + 0.5) + c$.

If $dv \leq 0$, then pixel E is selected, M is incremented by one in X direction, and the next position we need to consider is $(X_P + 2, Y_P + 0.5)$. Here we have:

$$dv(E) = F(X_P + 2, Y_P + 0.5) = a \cdot (X_P + 1) + b \cdot (Y_P + 0.5) + c + a = dv + a, \quad (3.4)$$

where we call the increment to add $\Delta E^- = a = \Delta Y$.

If $dv > 0$, then pixel NE is selected, M is incremented by one step in both X and Y coordinates, and the next position we need to consider is $(X_P + 2, Y_P + 1.5)$. Here we have:

$$dv(NE) = F(X_P + 2, Y_P + 1.5) = a \cdot (X_P + 1) + b \cdot (Y_P + 0.5) + c + a + b = dv + a + b, \quad (3.5)$$

where we call the increment to add $\Delta E^+ = a + b = \Delta Y - \Delta X$.

Since (X_1, Y_1) is on the line, $F(X_1, Y_1) = 0$, so we can directly calculate the initial value of dv for choosing between E and NE. The first midpoint is at $(X_1 + 1, Y_1 + 0.5)$, and:

$$F(X_1 + 1, Y_1 + 0.5) = F(X_1, Y_1) + a + \frac{b}{2} = F(X_1, Y_1) + \Delta Y - \frac{\Delta X}{2} = F(X_1, Y_1) + dv_{start}. \quad (3.6)$$

Using dv_{start}, we choose the second pixel, and so on. To eliminate the fraction in dv_{start}, we multiply the original function $F(X, Y)$ (Equation 3.2) by 2:

$$F(X, Y) = 2 \cdot (a \cdot X + b \cdot Y + c).$$

This also multiplies the constants ΔE^- and ΔE^+ and the decision variable dv_{start} without affecting its sign.

Bresenham summarized the above formulation in to the following algorithm (note that we renamed the decision variable dv to E):

BresenhamLineGenerator(X_1, Y_1, X_2, Y_2, I)
  ΔX=X_2-X_1; ΔY=Y_2-Y_1;
  E= - ΔX; ΔE^- = $2 \cdot \Delta Y$; ΔE^+ = $2(\Delta Y$ - $\Delta X)$;
  $Y = Y_1$;
  for $X = X_1$ **to** $X2$
    if $E \leq 0$ **then** E += ΔE^-;
    else E += ΔE^+; Y++;
    endif
    Add Frame Buffer (X, Y, I);
  endfor
end

3.1.1 Hardware implementation of Bresenham's line-drawing algorithm

The hardware implementation of Bresenham's line-drawing algorithm is straightforward. Note that the selection of the pixel coordinates X, Y is evaluated at every clock cycle, where the X coordinate is always incremented by 1, and the increment of the Y coordinate depends on the sign of the *decision variable* E. If the sign of E is zero or negative, there is no increment in the Y coordinate and, if the sign of E is positive, the Y coordinate is incremented by 1. Note that the generated line is "jagged" as shown in bottom of Figure 3.11 because we did not apply any filtering techniques. The block scheme of this hardware is shown in Figure 3.2. We can follow the operation of this hardware through its timing sequence shown in Figure 3.3. The control sequence of the hardware is given by the following behavioral model:

```
ARCHITECTURE Behavior OF BresenhamLG IS
BEGIN                          -- The line runs from (0,0) to (7,5)
  PROCESS ( ClkX )
  VARIABLE Xi,Yi,Ev: bit_vector_16;
  VARIABLE Running: BOOLEAN := FALSE;
  BEGIN
    IF ( Running = FALSE ) THEN
      IF ( Start = '1' ) THEN
        Stop <= '0';
        Running := TRUE;
        Ev := E;
        Xi := X1;
        Yi := Y1;
      END IF;
    ELSE
      IF( ClkX = '0' ) THEN
        IF ( Xi > X2 ) THEN
          Running := FALSE;
          Stop <= '1';
        ELSE
          IF ( Ev < 0 ) THEN
            Ev := (Ev + Edec);
          ELSE
            ClkY <= '1', '0' AFTER Delay;
            Ev := (Ev + Einc);
            Yi := (Yi + 1);
          END IF;
        END IF;
      ELSE                     -- Add-Frame-Buffer and inrement X
        X  <= Xi;
        Y  <= Yi;
        Xi := (Xi + 1);
      END IF;
    END IF;
  END PROCESS;
END Behavior;
```

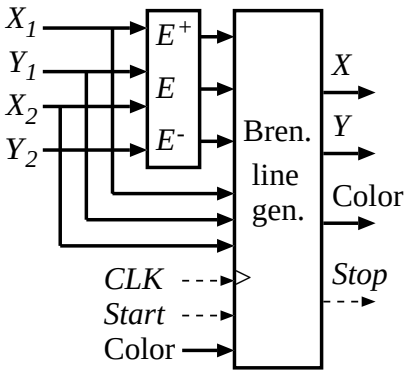


Figure 3.2: Hardware implementation of Bresenhame's line-drawing algorithm

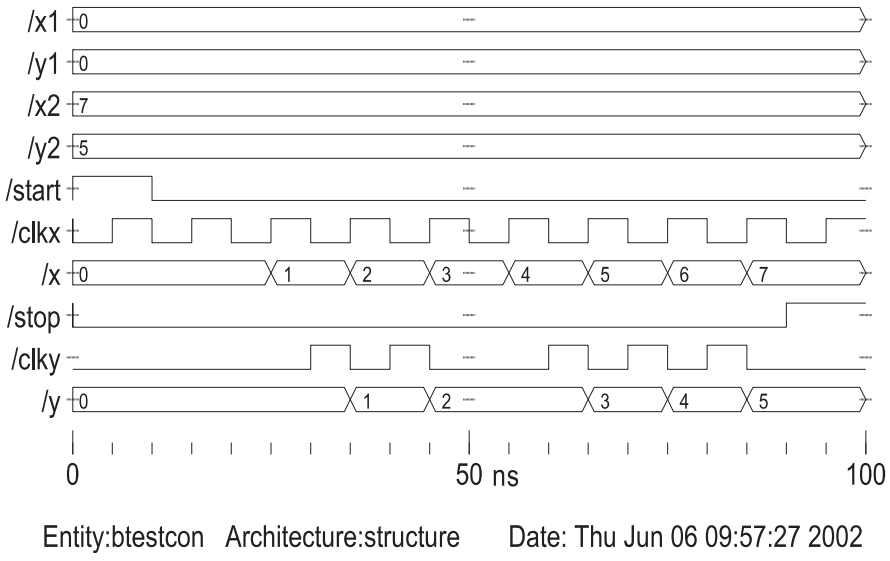


Figure 3.3: Time sequence of the hardware implementation of Bresenhame's line-drawing algorithm

3.2 Anti-aliasing lines

Line anti-aliasing methods are important techniques to handle the jagged lines [CW99] [Lui94] [Cro81]. These jaggies are the result of the improper sampling, which can be reduced by filtering. In order to realize this filtering, anti-aliasing line drawing algorithms have to calculate an integral over the intersection of one-pixel wide line and support of the filter kernel centered around the pixel concerned. The support and the shape of the filter depend on the selected filter type.

An ideal low pass filter would require the $sinc$ function to be convolved with the line, but this is quite intensive computationally, has infinite support, and may result in negative colors, which cannot be displayed. Thus approximations of the $sinc$ function are used in practice. The simplest and the most popular approximation is the box filter, which is 1 inside a pixel area and 0 otherwise.

3.2.1 Box-filtering lines

For box filtering, the intersection of the one-pixel wide line segment and the pixel concerned has to be calculated. The color of the pixel will be a sum of the line color and the background color, weighted by the intersection area and that area of the pixel which is not inside the intersection, respectively.

Looking at Figure 3.4, we can see that a maximum of three pixels may intersect a pixel rectangle in each column if the slope is between 0 and 45 degrees. Let the vertical distance of the three closest pixels to the center of the line be s, t and r respectively, and suppose $s < t < r$. By geometric considerations $s, t < 1$, $s + t = 1$ and $r \geq 1$ should also hold.

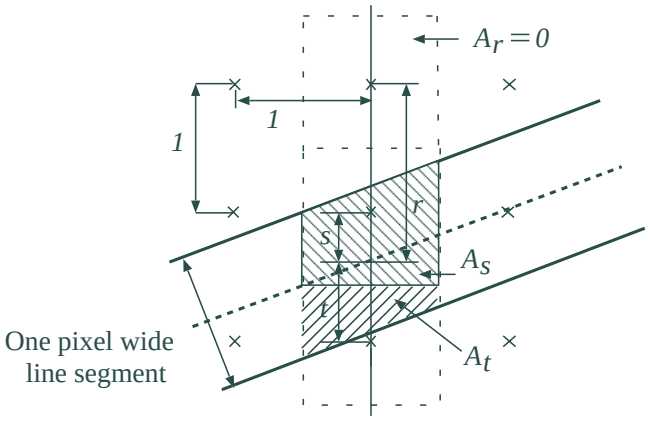


Figure 3.4: Box filtering of a line segment

Unfortunately, the areas of intersection, A_s, A_t and A_r, depend not only on s, t and r, but also on the slope of the line segment. This dependence, however, can be ignored by using the following approximation:

$$A_s \approx (1 - s), \quad A_t \approx (1 - t), \quad A_r \approx 0. \tag{3.7}$$

These equations are accurate only if the line segment is horizontal, but can be accepted as a fair approximation for lines with a slope from 0 to 45 degrees. Variables s and t are calculated for a line $Y = m \cdot X + b$:

$$s = m \cdot X + b - Round(m \cdot X + b) = Error(X), \;\; t = 1 - s, \tag{3.8}$$

where $Error(X)$ is, in fact, the accuracy of the digital approximation of the line for vertical coordinate Y. The color contribution of the two closest pixels in this pixel column is:

$$I_s = I \cdot (1 - Error(X)), \;\; I_t = I \cdot Error(X). \tag{3.9}$$

These formulae are also primary candidates for incremental evaluation since if the closest pixel has the same Y coordinate for an $X + 1$ as for X:

$$I_s(X + 1) = I_s(X) - I \cdot m, \;\; I_t(X + 1) = I_t(X) + I \cdot m.$$

If the Y coordinate has been incremented when stepping from X to $X + 1$, then:

$$I_s(X + 1) = I_s(X) - I \cdot m + I, \;\; I_t(X + 1) = I_t(X) + I \cdot m - I.$$

The incremental algorithm of Bresenham's line generator using Box filter is:

AntiAliasedBresenhamLine(X_1, Y_1, X_2, Y_2, I)
  ΔX=X_2-X_1; ΔY=Y_2-Y_1;
  E= - 2 $\cdot$ ΔX; ΔE^- = 2 $\cdot$ ΔY; ΔE^+ = 2(ΔY - ΔX);
  ΔI^-= ΔY / ΔX; ΔI^+ = I - ΔI^-;
  I_s = I + ΔI^-; I_t = - ΔI^-;
  $Y = Y_1$;
  for $X = X_1$ **to** $X2$
    if $E \leq 0$ **then**
      E += ΔE^-; I_s -= ΔI^-; I_t += ΔI^-;
    else
      E += ΔE^+; I_s += ΔI^+; I_t -= ΔI^+; Y++;
    endif
    Add Frame Buffer (X, Y, I_s); Add Frame Buffer $(X, Y$+1$, I_t)$;
  endfor
end

This algorithm assumes that the frame buffer is initialized such that each pixel has the color derived without taking this new line into account, and thus the new contribution can simply be added to it. This is true only if the frame buffer is initialized to the color of background and lines do not cross each other. The artifact resulting from crossed lines is usually negligible.

In the general case I must rather be regarded as a weight value determining the portions of the new line color and the color already stored in the frame buffer, which corresponds to the color of objects behind the new line. This requires program line "Add Frame Buffer (X, Y, I)" to be replaced by the following statements:

old-color = frame-buffer $[X, Y]$;
frame-buffer $[X, Y]$ = line-color $\cdot I +$ old-color $\cdot (1 - I)$;

3.2.2 Incremental cone-filtering lines

Box filter is a piece-wise constant approximation of the $sinc$ function. An even better approximation is provided by a piece-wise linear function, which is called the cone filter. For cone filtering, the volume of the intersection between the one-pixel wide line segment and the one-pixel radius cone centered around the pixel concerned has to be calculated. The height of the cone must be selected to guarantee that the volume of the cone is 1. Looking at Figure 3.5, we can see that a maximum of three pixels may have intersection with a base circle of the cone in each column if the line slope is between 0 and 45 degrees.

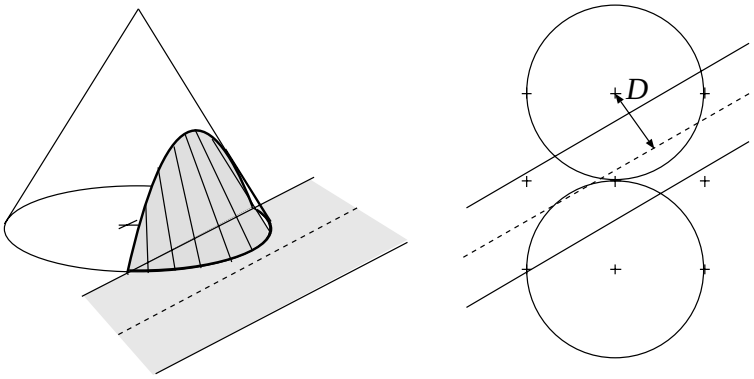


Figure 3.5: Cone-filtering of a line segment

Let the distance between the pixel center and the center of the line be D. For possible intersection, D must be in the range of $[-1.5 \ldots 1.5]$. For a pixel center (X, Y), the convolution integral — that is the volume of the cone segment above a pixel — depends only on the value of D, thus it can be computed for discrete D values and stored in a lookup table $V(D)$ during the design of the algorithm. The number of table entries depends on the number of intensity levels available to render lines, which in turn determines the necessary precision of the representation of D. Since $8 \ldots 16$ intensity levels are enough to eliminate the aliasing, the lookup table is defined here for three and four fractional bits. Since function $V(D)$ is obviously symmetrical, the number of necessary table entries for three and four fractional bits is $1.5 \cdot 2^3 = 12$ and $1.5 \cdot 2^4 = 24$ respectively. The precomputed $V(D)$ tables for three and four fractional bits, are shown in Figure 3.6.

Now the generation of D and the subsequent pixel coordinates must be discussed. Gupta and Sproull [GSS81] proposed a modification of the Bresenham algorithm to produce the pixel address and introduce an incremental scheme to generate the subsequent D distances. However, it required floating point multiplications on the pixel level, which are hard to realize in hard-

D_3	0	1	2	3	4	5	6	7	8	9	10	11
$V(D_3)$	7	6	6	5	4	3	2	1	1	0	0	0

D_4	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
$V(D_4)$	14	14	13	13	12	12	11	10	9	8	7	6	5	4	3	3	2	2	1	1	0	0	0	0

Figure 3.6: Precomputed $V(D)$ weight tables

ware. Thus we propose a new algorithm that has a similar approach, but applies just fixed point additions [Abb01].

Let the slope of the line be ϕ, and the vertical distance between the center of the line and the closest pixel be d (note that, for the sake of simplicity only lines slopes in the range of $[0 \ldots 45]$ are considered).

For geometric reasons, as illustrated by Figure 3.7, the D values for the three vertically arranged pixels are:

$$\begin{aligned} D &= d \cdot \cos\phi = d \cdot \frac{\Delta X}{\sqrt{(\Delta X)^2 + (\Delta Y)^2}} = d \cdot \Delta D, \\ D_H &= (1-d) \cdot \cos\phi = -D + \frac{\Delta X}{\sqrt{(\Delta X)^2 + (\Delta Y)^2}} = -D + \Delta D, \\ D_L &= (1+d) \cdot \cos\phi = D + \frac{\Delta X}{\sqrt{(\Delta X)^2 + (\Delta Y)^2}} = D + \Delta D. \end{aligned} \tag{3.10}$$

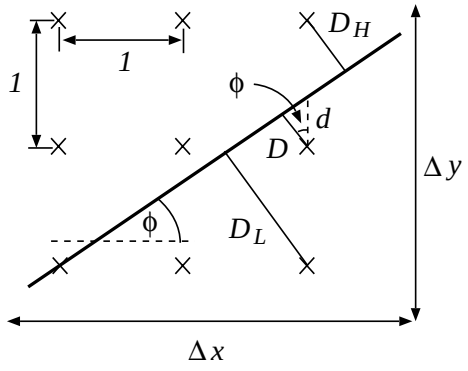


Figure 3.7: Incremental calculation of distance D

Let us realize that these formulae can also be simplified by the incremental concept. Assume

first that the Bresenham algorithm does not increment the Y coordinate:

$$\begin{aligned} d(X+1) &= d(X) + \frac{\Delta Y}{\Delta X}, \\ D(X+1) &= D(X) + \frac{\Delta Y}{\sqrt{(\Delta X)^2 + \Delta Y)^2}} = D(X) + \Delta D^-. \end{aligned} \tag{3.11}$$

Now we consider the case when the Bresenham algorithm increments the Y coordinate:

$$\begin{aligned} d(X+1) &= 1 - \left(d(X) - \frac{\Delta Y}{\Delta X}\right), \\ D(X+1) &= -D(X) + \left(1 - \frac{\Delta Y}{\Delta X}\right) \cdot \frac{\Delta X}{\sqrt{(\Delta X)^2 + \Delta Y)^2}} \\ &= -D(X) + \frac{\Delta X - \Delta Y}{\sqrt{(\Delta X)^2 + \Delta Y)^2}} = -D(X) + \Delta D^+. \end{aligned} \tag{3.12}$$

The complicated operations including divisions and square root in Equations 3.10, 3.11 and 3.12 should be executed once for the whole line, thus the pixel level algorithms contain just simple instructions and a single addition not counting the averaging with the colors already stored in the frame buffer.

In the subsequent program expressions that are difficult to calculate are evaluated at the beginning, and stored in the following variables:

$$denom = \frac{1}{\sqrt{(\Delta X)^2 + (\Delta Y)^2}}, \quad \Delta D = \Delta X \cdot denom. \tag{3.13}$$

Summarizing the new incremental cone-filtering line-drawing algorithm is:

IncrementalConeFilteringLine (X_1, Y_1, X_2, Y_2, I)
  ΔX = X_2-X_1; $\Delta Y = Y_2 - Y_1$;
  E = - ΔX; ΔE^- =2 $\cdot \Delta Y$; ΔE^+ = 2(ΔY - ΔX);
  $denom = 1/\sqrt{(\Delta X)^2 + (\Delta Y)^2}$; ΔD = $\Delta X \cdot denom$;
  $Y = Y_1$;
  for $X = X_1$ **to** X_2 **do**
    if $E \leq 0$ **then**
      E += ΔE^-; D += ΔD^-;
    else
      E += ΔE^+; D += ΔD^+; Y++;
    endif
    D_L = D + ΔD; D_H = - D + ΔD;
    Add Frame Buffer $(X, Y, V(D))$;
    Add Frame Buffer $(X, Y$+1$, V(D_H))$;
    Add Frame Buffer $(X, Y$-1$, V(D_L))$;
  endfor
end

3.2.3 Hardware implementation of incremental cone-filtering lines

The hardware implementation of incremental cone-filtering lines algorithm is based on Bresenham line generator algorithm to enhance the generated line by removing the jaggies, where three adjacent pixels are intensified in the Y coordinate for every step on the X coordinate. The three pixels are $(X, Y), (X, Y-1)$ and $(X, Y+1)$. The hardware is composed of two stages, stage one and stage two. Stage one loads the initial values for all registers and counters, calculates the X, Y coordinates of the center line, and the values for all the variables used. Stage two computes the three pixels coordinates, fetches their color values from a precomputed color table according to their relevant indexed values stored in the registers D, D_L, and D_H, respectively. In stage one, each of the working registers E, D, D_L, and D_H requires one clock cycle to modify its value, so four clock cycles are needed to complete stage one operation. At the working registers D and E the sign of the *decision variable* E will be checked, if the sign of E is zero or negative X is incremented by 1, E is incremented by ΔE^-, and D is incremented by ΔD^-. If the sign of E is positive, both X and Y are incremented by 1, E is incremented by ΔE^+, and D is incremented by ΔD^+. When this process is over, the $SS2$ (start stage two signal) is activated. The working registers D_H and D_L are modified as follows, $D_L = D + \Delta D$ and $D_H = -D + \Delta D$. Stage two needs only three clock cycles to compute the three pixels coordinates and stors their colors in the raster memory. Because Stage two does not depend on D_L and D_H, so the two stages can overlap each other by two clock cycles (Figure 3.10). Stage two has no operation for the forth clock cycle, because it waits for stage one to complete its operation. At the initialization process D is initialized to ΔD^-, and D_H and D_L are initialized to zero. To have visible time sequence, the line runs from $(3, 2)$ to $(8, 5)$. The block scheme of this hardware is shown in Figure 3.8 and its generated timing sequence is given by Figure 3.9.

The initialization process and the control of the working registers are given by the following behavioral model:

```
ARCHITECTURE Behavior OF Control1 IS
BEGIN
  PROCESS ( CLK )
  VARIABLE State: INTEGER RANGE 0 to 15 := 0;
  BEGIN
    IF ( CLK = '1' ) THEN
      CASE State IS                    --  0 to 11; are the loading states
        WHEN 12 =>                     -- 12 to 15; are the working states
          IF ( StopSS2 = '1' ) THEN
            StartS2 <= '0';  State := 0;
          ELSE
            Sel1Out <= Sel_D;  Sel1In  <= Sel_D;  PM <= '1';
            IF ( EOut = '1' ) THEN     -- E is negative or zero
              Sel2 <= Sel_dDm;         -- Delta D mince
            ELSE                       -- E is positive
              Sel2 <= Sel_dDp;         -- Delta D plus
            END IF;
            ClkMpxR <= '1' AFTER Delay1, '0' AFTER Delay2;
            State := 13;
          END IF;
```

```
        WHEN 13 =>
          Sel1In  <= Sel_E;  Sel1Out <= Sel_E;  PM <= '1';
          IF ( EOut = '1' ) THEN
            Sel2 <= Sel_dEm;           -- Delta E mince
          ELSE
            Sel2 <= Sel_dEp;           -- Delta E plus
            ClkY <= '1', '0' AFTER Delay;
          END IF;
          StartS2 <= '1' AFTER Delay1, '0' AFTER Delay2;
          ClkMpxR <= '1' AFTER Delay1, '0' AFTER Delay2;
          State := 14;
        WHEN 14 =>                     -- DL = Delta low, --dD = Delta D
          Sel1Out <= Sel_D;  Sel1In <= Sel_DL;  Sel2 <= Sel_dD;  PM <= '1';
          ClkMpxR <= '1' AFTER Delay1, '0' AFTER Delay2;
          State := 15;
        WHEN 15 =>                     -- DH = Delta high, --dD = Delta D
          Sel1Out <= Sel_D;  Sel1In <= Sel_DH;  Sel2 <= Sel_dD;  PM <= '0';
          ClkMpxR <= '1' AFTER Delay1, '0' AFTER Delay2;
          State := 12;
      END CASE;
    END IF;
  END PROCESS;
END Behavior;
```

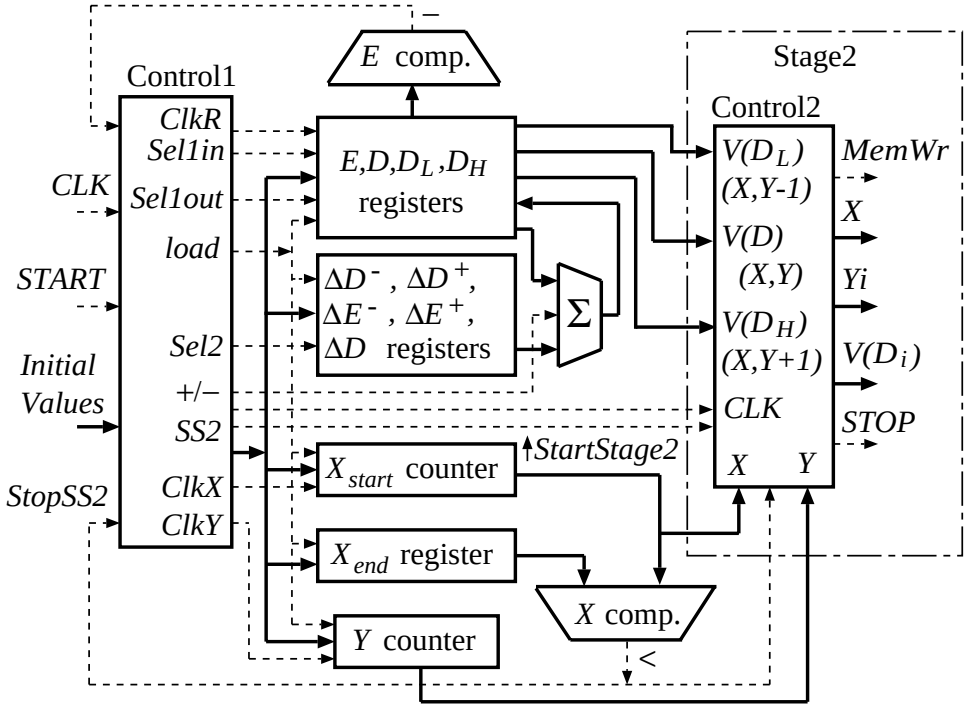


Figure 3.8: Hardware implementation of incremental cone-filtering lines

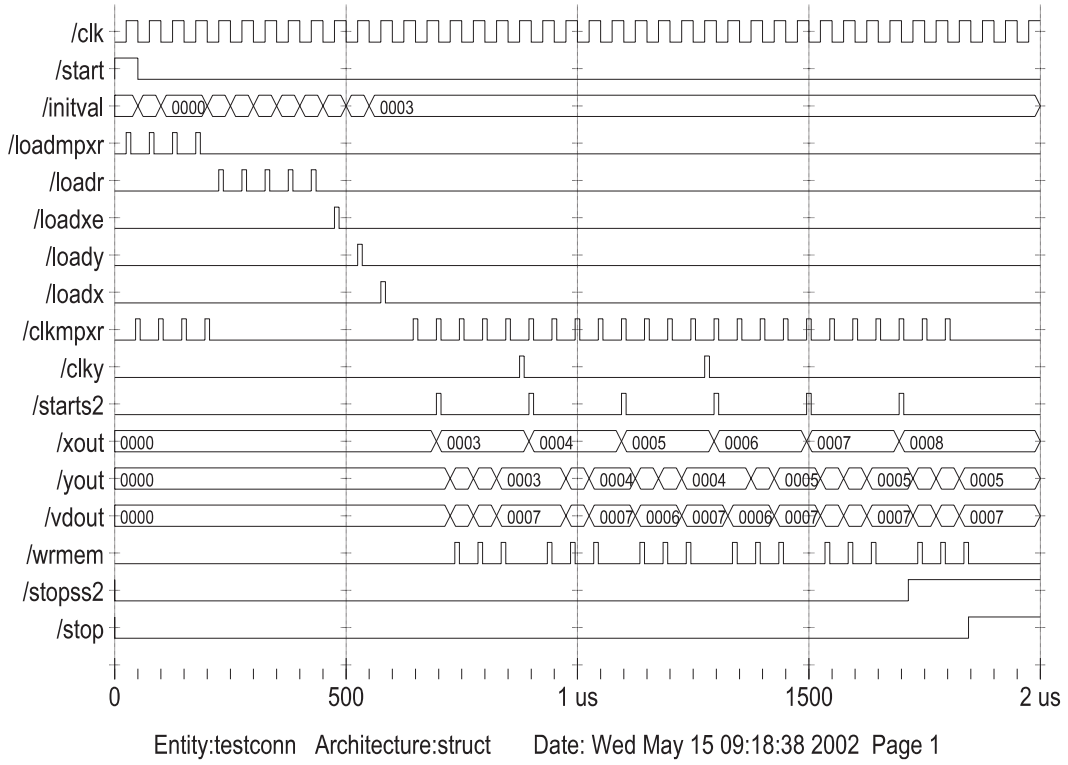


Figure 3.9: The time sequences of incremental cone-filtering lines

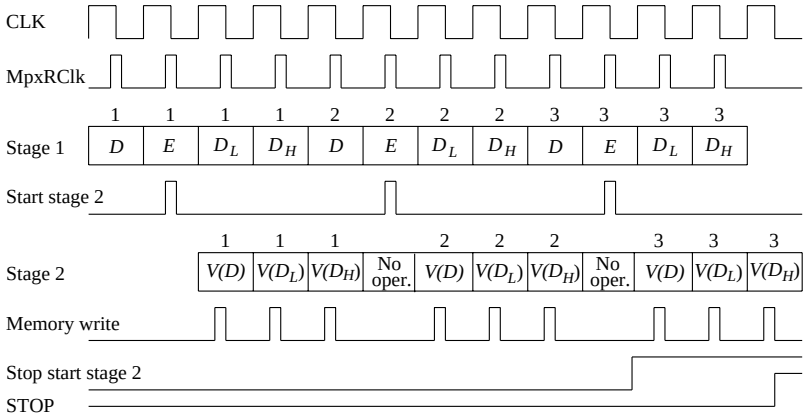


Figure 3.10: The overlapped operations in the hardware of incremental cone-filtering lines

Figures 3.11 and 3.12 illustrate the results of Bresenham line-drawing algorithm, and the Bresenham's line-drawing with box-filtering and incremental cone-filtering algorithms.

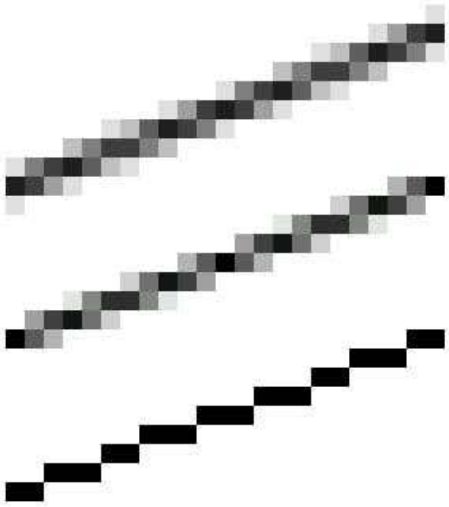


Figure 3.11: Comparison of lines drawn by Bresenham's algorithm (bottom), box-filtering (middle) and the incremental cone-filtering (top)

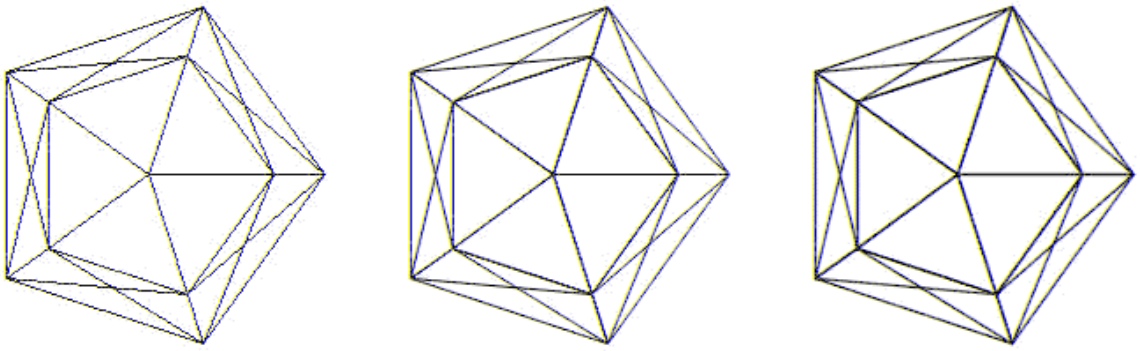


Figure 3.12: Comparison of coarsely tessellated wire-frame spheres (40 triangles) drawn by Bresenham's algorithm (left), box-filtering (middle) and the incremental cone-filtering (right)

3.3 Depth cueing

To enhance the appearance of $3D$ wire-frame images, a depth cueing procedure is applied, which uses more intensive colors for rendering those pixels which are closer to the eye position and the pixels become darker as the line gets further into the background, so the line seems to fade into the distance (Figure 3.13). A basic line drawing algorithm can generate the pixel address of a $2D$ digital line, therefore it must be extended to produce the color intensities by an incremental algorithm. In order to derive an incremental formula, the increment of intensity I is determined. Let the $3D$ screen space coordinates of the two end points of the line be (X_1, Y_1, Z_1) and (X_2, Y_2, Z_2) respectively and suppose that the Z values are in the range $(0 \ldots Z_{max})$. Assume that the intensity factor of depth cueing is C_{max} for $Z = 0$ and C_{min} for Z_{max}. The number of pixels composing this digital line is $N = max(|X_2 - X_1|, |Y_2 - Y_1|)$. The perceived color, taking into account the effect of depth cueing, is:

$$I(Z) = I_0 \cdot C(Z) = I_0 \cdot \left(C_{max} - \frac{C_{max} - C_{min}}{Z_{max}} \cdot Z \right). \tag{3.14}$$

The difference in the color of the two pixel centers is $\Delta I = (I(Z_2) - I(Z_1))/N$. This constant value should be added to the pixel color, which can be realized by a simple hardware similar to that of Figure 2.6.

Figure 3.13 also demonstrates hidden surface removal (visibility calculation), which finds the surface points that are visible through the given pixels, that are the nearest from the eye towards the center of the concerned pixels. For this, the z-buffer method is used.

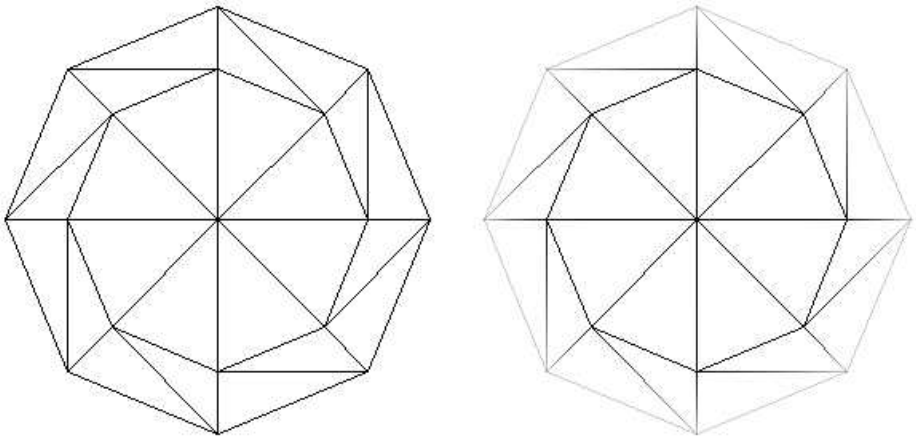


Figure 3.13: Comparison of coarsely tessellated wire-frame spheres (48 triangles) hidden surface removed, without depth-cueing (left) and with depth-cueing (right)

Chapter 4

Shaded surface rendering with linear interpolation

As mentioned in Section 1.2, the rendering equation, even in its simplified form, contains a lot of complex operations, including the computation of the vectors, their normalization and the evaluation of the output radiance, which makes the process rather resource demanding.

The speed of rendering could be significantly increased if it were possible to carry out the expensive computations just for a few points or pixels, and the rest could be approximated from these representative points by much simpler expressions. One way of obtaining this is the tessellation of the original surfaces to triangle meshes and using the vertices of the triangles as representative points. These techniques are based on linear (or in the extreme case, constant) interpolation requiring a value of the function to be approximated at the representative points, which leads to the incremental concept. These methods are particularly efficient if the geometric properties can also be determined in a similar way, connecting incremental shading to the incremental visibility calculations of triangle mesh models. Only triangle mesh models are considered, thus the geometry should be approximated by a triangle mesh before the algorithms can be used. It is assumed that the geometry has been transformed to the screen coordinate system suitable for visibility calculations and projection. In the screen coordinate system the X, Y coordinates of a point are equal to the corresponding coordinates of that pixel in which this point can be seen, and the Z coordinate increases with the distance from the viewer, thus it is the basis of visibility calculations (Figure 4.1). Note, on the other hand, that the vectors used by the rendering equation are not transformed, because the viewing transformation is not angle preserving, thus the transformation would distort the angles between them.

4.1 Rasterizing an image space triangle

Having transformed the triangles to the screen space, those pixels that cover the projection of the triangle should be identified. This is a triangle filling algorithm (Section 2.2).

In the following sections a visibility algorithm and incremental shading algorithms are discussed, which are capable to work parallel to the triangle rasterization algorithm [YR97].

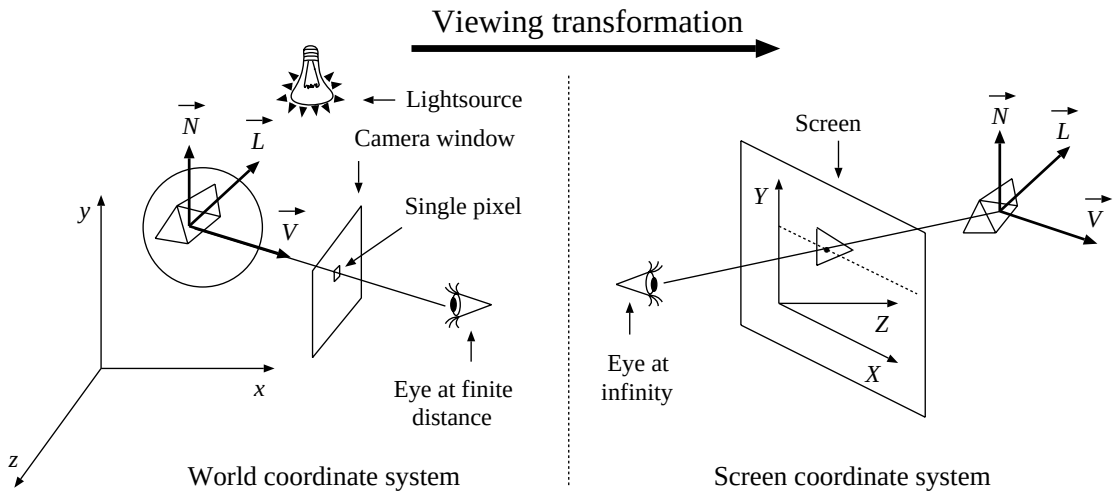


Figure 4.1: Transformation to the screen coordinate system

4.2 Linear interpolation on a triangle

In image synthesis functions should be computed on image space triangles (Figure 2.4). The most important family contains linear functions.

Suppose that the function to be interpolated is $\xi(X, Y)$ where ξ can be either vector or scalar and X, Y are the pixel coordinates (Figure 4.2).

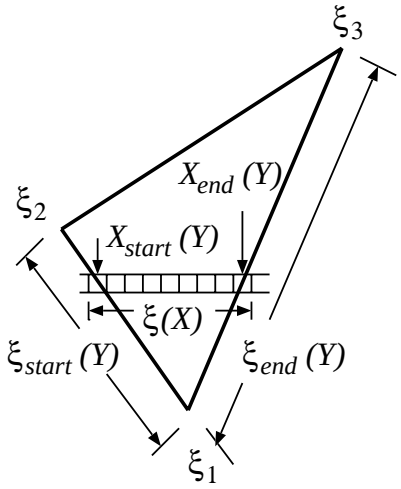


Figure 4.2: Linear interpolation on a triangle

The function is defined by values at the vertices of the triangle, i.e. we have:

$$\xi_1 = \xi(X_1, Y_1), \quad \xi_2 = \xi(X_2, Y_2), \quad \xi_3 = \xi(X_3, Y_3).$$

Linearity means that:

$$\xi(X, Y) = a \cdot X + b \cdot Y + c. \tag{4.1}$$

The question is whether or not these parameters a, b and c are worth computing and using in the interpolation scheme or their computation requires too much overhead.

In this section three linear interpolation methods are discussed and compared.

4.2.1 2D linear interpolation

This alternative computes the a, b and c parameters of the interpolation scheme and uses the standard form of the $2D$ function during interpolation. These parameters can be computed from the constraints at the vertices:

$$\begin{aligned} \xi_1 &= a \cdot X_1 + b \cdot Y_1 + c, \\ \xi_2 &= a \cdot X_2 + b \cdot Y_2 + c, \\ \xi_3 &= a \cdot X_3 + b \cdot Y_3 + c. \end{aligned} \tag{4.2}$$

This is a system of linear equations for unknown parameters a, b and c.

When the parameters are available, function ξ for a given X, Y requires the evaluation of $a \cdot X + b \cdot Y + c$, which, in turn, needs two multiplications and two additions. This step can also be speeded up by the incremental concept since:

$$\xi(X + 1, Y) = \xi(X, Y) + a,$$

thus a new function value requires just a single addition.

4.2.2 Using a sequence of 1D linear interpolations

It is usually simpler to replace the two-variate interpolation scheme by two one-variate schemes, one running on the edges of the triangle and the other running inside horizontal spans called scan-lines.

Thus it is enough to consider a one-variate interpolation either on the edge of the triangle or inside the scan-line. This alternative uses the values of the coordinates and the vectors at the start and end edges of the triangle at each scan-line, which are evaluated from the coordinates and the vectors at the three vertices, and applies a running variable t between the start and end edges of each scan-line. That is $t = 0$ at the start edge and $t = 1$ at the end edge, where

$$t = \frac{X - X_{start}}{X_{end} - X_{start}}. \tag{4.3}$$

The scheme of interpolation on the scan-line is as follows:

$$\xi(t) = (1-t)\cdot\xi_{start} + t\cdot\xi_{end}. \tag{4.4}$$

Substituting t in Equation 4.4 for the X coordinate of a single scan-line, we obtain:

$$\xi(X) = \frac{X_{end}-X}{X_{end}-X_{start}}\cdot\xi_{start} + \frac{X-X_{start}}{X_{end}-X_{start}}\cdot\xi_{end}. \tag{4.5}$$

Applying the incremental concept to Equation 4.5 yields to:

$$\xi(X+1) = \xi(X) + \frac{\xi_{end}-\xi_{start}}{X_{end}-X_{start}} \tag{4.6}$$

4.2.3 Interpolation with blending functions

Originally, the interpolating function is linear. Let us express ξ with 3 blending functions:

$$\xi(X,Y) = a_1(X,Y)\cdot\xi_1 + a_2(X,Y)\cdot\xi_2 + a_3(X,Y)\cdot\xi_3 \tag{4.7}$$

where

$$a_i(X,Y) = a_{ix}X + a_{iy}Y + a_{i0}$$

$(i = 1,2,3)$ is a linear weighting function. The interpolation criterion requires that $a_i(X,Y) = 1$ at vertex i and 0 in the other two vertices. From this criterion, the parameters (a_{ix}, a_{iy}, a_{i0}) of each weighting function can be determined.

4.3 An image space hidden surface elimination algorithm: the z-buffer algorithm

One of the simplest visible-surface algorithms to implement in either software or hardware, is the z-buffer algorithm. In addition to a frame-buffer (Fp) in which color values are stored, a z-buffer (Zp) is designed with the same number of entries, in which a z-value is stored for each pixel. The z-buffer is initialized to infinity, representing the z-value at the back-clipping plane, and the frame buffer is initialized to the background color. The smallest value that can be stored in the z-buffer represents the Z coordinate of the front clipping plane. Triangles are scan-converted into the frame buffer in arbitrary order. During the scan-conversion process, if the triangle point being scan-converted at (X,Y) is closer to the viewer than the point whose color and depth are currently in the buffers, then the new point's color and depth replace the old values.

The basic form of the z-buffer algorithm is:

```
Initialize frame-buffer to background color;
Initialize each cell of z-buffer[] to ∞;
for each object O do
    for each pixel P covered by the projection of O do
        if z-coordinate of the surface point < z-buffer [P] then
            color of P = color of surface point;
            z-buffer[P] = depth of surface point;
        endif
    endfor
endfor
```

No presorting is necessary and no object-object comparisons are required. The entire process is no more than a search over each set of pairs $Zp_i(X,Y), Fp_i(X,Y)$ for fixed X and Y, to find the smallest Z_i. The z-buffer and the frame-buffer record the information associated with the smallest Z for each (X,Y).

4.3.1 Hardware implementation of z-buffer algorithm

Having approximated the surface by a triangle mesh, the surface is given by the set of mesh vertices, which should have been transformed to screen coordinate system. The visibility calculation of the surface is thus a series of visibility computations for screen coordinate system triangles. This allows us to consider only the problem of scan conversion of single triangle. Let the vertices of the triangle in screen coordinates be $\vec{r}_1 = (X_1, Y_1, Z_1)$, $\vec{r}_2 = (X_2, Y_2, Z_2)$ and $\vec{r}_3 = (X_3, Y_3, Z_3)$, respectively. The scan-conversion algorithm should determine the X, Y pixel addresses and the corresponding Z coordinates of those pixels which belong to this triangle (Figure 4.3). If the X, Y pixel addresses are already available, then the calculation of the corresponding Z coordinate can exploit the fact that the triangle is on a plane, thus the Z coordinate is some linear function of the X, Y coordinates. This linear function can be derived from the equation of the plane, using the notation $\vec{n}$ and $\vec{r}$ to represent the normal vector and the points of the plane respectively:

$$\vec{n} \cdot \vec{r} = \vec{n} \cdot \vec{r}_1 \tag{4.8}$$

where

$$\vec{n} = (\vec{r}_2 - \vec{r}_1) \times (\vec{r}_3 - \vec{r}_1).$$

Let us denote the constant $\vec{n} \cdot \vec{r}_1$ by C, and express the equation in scalar form, substituting the coordinates of the vertices $\vec{r} = (X, Y, Z(X,Y))$ and the normal of the plane $\vec{n} = (n_X, n_Y, n_Z)$. The function of $Z(X,Y)$ is then:

$$Z(X,Y) = \frac{C - n_X \cdot X - n_Y \cdot Y}{n_Z}. \tag{4.9}$$

This linear function must be evaluated for those pixels which cover the pixel space triangle defined by the vertices (X_1, Y_1), (X_2, Y_2) and (X_3, Y_3). Equation 4.9 is suitable for the application

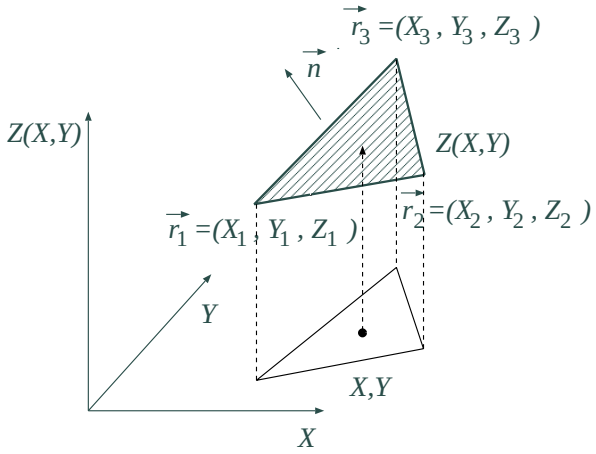


Figure 4.3: Screen space triangle

of the incremental concept. In order to make the boundary curve differentiable and simple to compute, the triangle is split into two parts by a horizontal line at the position of the vertex which is in between the two vertices in the Y direction (horizontal sided triangle).

The computational burden for the evaluation of the linear expression of the Z coordinate and for the calculation of the starting and ending coordinates of the horizontal spans of pixels covering the triangle can be significantly reduced by the incremental concept (Figure 4.4).

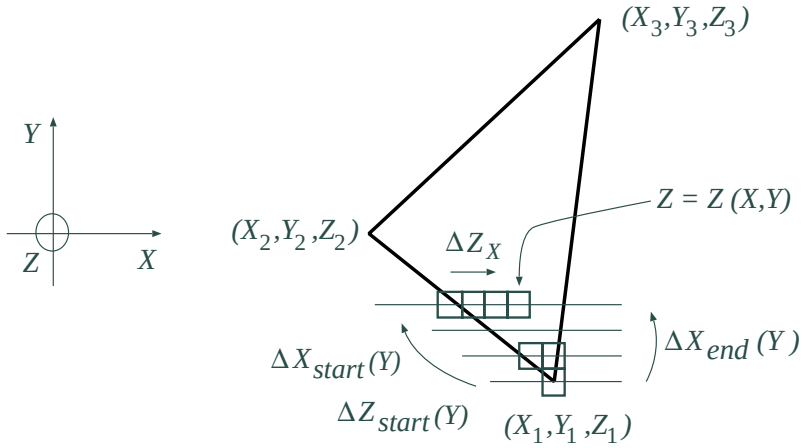


Figure 4.4: Incremental concept in z-buffer calculations

Expressing $Z(X+1,Y)$ as a function of $Z(X,Y)$, we get:

$$Z(X+1,Y) = Z(X,Y) + \frac{\Delta Z(X,Y)}{\Delta X} \cdot 1 = Z(X,Y) - \frac{n_X}{n_Y} = Z(X,Y) + \Delta Z_X. \quad (4.10)$$

Since ΔZ_X does not depend on the actual X, Y coordinates, it has to be evaluated once for the triangle. In a scan-line, the calculation of a Z coordinate requires a single addition according to Equation 4.10.

Since X and Z vary linearly along the left and right edges of the triangle, the start and end edges of a scan-line can be obtained by the following expressions in the range of $Y_1 \leq Y \leq Y_2$:

$$\begin{aligned} X_{start}(Y+1) &= X_{start}(Y) + \frac{X_2 - X_1}{Y_2 - Y_1} = X_{start} + \Delta X_{start}(Y), \\ X_{end}(Y+1) &= X_{end}(Y) + \frac{X_3 - X_1}{Y_3 - Y_1} = X_{end} + \Delta X_{end}(Y), \\ Z_{start}(Y+1) &= Z_{start}(Y) + \frac{Z_2 - Z_1}{Y_2 - Y_1} = Z_{start} + \Delta Z_{start}(Y). \end{aligned} \tag{4.11}$$

The complete incremental z-buffer algorithm is:

$X_{start} = X_1 + 0.5$; $X_{end} = X_1 + 0.5$; $Z_{start} = Z_1 + 0.5$;
for $Y = Y_1$ **to** Y_2 **do**
  $Z = Z_{start}$;
  for X = Trunc (X_{start}) **to** Trunc (X_{end}) **do**
    Z=Trunc (Z);
    if $Z <$ z-buffer (X, Y) **then**
      frame-buffer (X, Y) = computed color; z-buffer $(X, Y) = Z$;
    endif
    Z += ΔZ_X;
  endfor
  X_{start} += $\Delta X_{start}(Y)$; X_{end} += $\Delta X_{end}(Y)$; Z_{start} += $\Delta Z_{start}(Y)$;
endfor

Having represented the numbers in a fixed-point format, the hardware implementation for this algorithm is straightforward. This will be combined with Gouraud shading in Section 4.4.2.

4.4 Incremental shading algorithms

Incremental shading algorithms follow the idea of computing the solution of the rendering equation just at a few representative points and interpolate the result of other points:

$$I^{out} = I^e + k_a \cdot I^a + \sum_{l=1}^{k} \mathcal{R}_l \cdot I_l^{in}, \tag{4.12}$$

where I^{out} is the outgoing radiance, I^e is the self emission, k_a and I^a are the ambient reflection parameter and ambient intensity respectively, and $\mathcal{R}_l = (k_d \cdot \cos\theta_{\vec{L}_l}) + (k_s \cdot \cos^n \delta)$ is the cosine weighted BRDF, i.e. $f_r(\vec{L}_l, \vec{x}, \vec{V}) \cdot \cos\theta_{\vec{L}_l}$.

According to the interpolation strategies, two important classes should be identified: constant shading and Gouraud shading [Gou71] [Nar95] incorporating linear interpolation.

4.4.1 Constant shading

Constant shading, also called flat shading, implies that the rendering equation is evaluated once for each triangle, and the color of the triangle is approximated by a constant value (left of Figure 4.7), usually obtained from the center of the triangle:

$$C(t) = I^{out}, \tag{4.13}$$

where C is the color value at position t.

Constant shading is fast and simple. In general, constant shading of triangle meshes provides an accurate rendering for an object if all the following assumptions are valid:

- The object is not an approximation of an object with a curved surface.
- All light-sources illuminating the object are sufficiently far from the surface, so that $\vec{L} \cdot \vec{N}$ and the attenuation function are constant over the surface.
- The viewing position is sufficiently far from the surface so that $\vec{V} \cdot \vec{R}$ is constant over the surface.

4.4.2 Gouraud shading

In Gouraud shading the rendering equation is evaluated at the vertices of the triangle and the color is linearly interpolated inside the triangles (right of Figure 4.7). Since in this case the difference of the color of the adjacent pixels is constant due to linear interpolation scheme, this strategy requires just a single addition per pixel, which can be easily implemented in hardware.

For the color computation inside the scan line, we have:

$$C(t) = (1-t)I^{out}_{start} + tI^{out}_{end}. \tag{4.14}$$

For each scan-line, the color at the intersection of the scan-line with a triangle edge is linearly interpolated from the colors at the edge end-points.

Recall that the same approach was applied to calculate the Z coordinate in z-buffer method (Section 4.3). Because of their algorithmic similarity, the same hardware implementation can be used to compute the Z coordinate, and the R, G, B, color coordinates.

The last equation in Equation 4.11 can be used again, but it actually represents three equations, one for each color coordinate, (R, G, B):

$$C(Y+1) = C(Y) + \frac{I^{out}_{end} - I^{out}_{start}}{Y_{end} - Y_{start}} = C(Y) + \Delta I. \tag{4.15}$$

The complete incremental algorithm for Gouraud shading is:

$X_{start} = X_1 + 0.5;\ \ X_{end} = X_1 + 0.5;$
$R_{start} = R_1 + 0.5;\ \ G_{start} = G_1 + 0.5;\ \ B_{start} = B_1 + 0.5;$
for $Y = Y_1$ **to** Y_2 **do**
  $R = R_{start};\ G = G_{start};\ B = B_{start};$
  for X = Trunc (X_{start}) **to** Trunc (X_{end}) **do**
    Store (Trunc(R), Trunc(G), Trunc(B)) in raster memory at X, Y;
    R += ΔR_X; G += ΔG_X; B += ΔB_X;
  endfor
  X_{start} += $\Delta X_{start}(Y)$; X_{end} += $\Delta X_{end}(Y)$;
  R_{start} += $\Delta R_{start}(Y)$; G_{start} += $\Delta G_{start}(Y)$; B_{start} += $\Delta B_{start}(Y)$;
endfor

4.5 Hardware implementation of Gouraud shading and z-buffer algorithms

The possibility of hardware implementation makes Gouraud shading very attractive and popular in advanced graphics hardware systems, although it has several severe drawbacks.

In this section we combine the algorithms of Gouraud shading and z-buffer introduced in the previous sections for a common hardware implementation. The block scheme of this hardware is shown in Figure 4.5, we can follow the operations of the hardware of a lower horizontal sided triangle through its timing sequence shown in Figure 4.6.

The linear interpolator of the hardware implementation of Gouraud shading algorithm combined with z-buffer algorithm is given by the following behavioral model:

```
ARCHITECTURE Behavior OF Interpolator IS
BEGIN
  PROCESS ( CLK )
    VARIABLE TmpVal: bit_vector_24 := (others => '0');
    BEGIN
      IF ( CLK = '1' ) THEN
        IF ( sel = '1' ) THEN
          TmpVal := InitVal;
        ELSE
          IF ( step = '1' ) THEN
            int_to_bit_Vector(bit_vector_to_int(TmpVal) +
                              bit_vector_to_int(StepVal), TmpVal);
          END IF;
        END IF;
        OutVal <= TmpVal AFTER Delay;
      END IF;
    END PROCESS;
END Behavior;
```

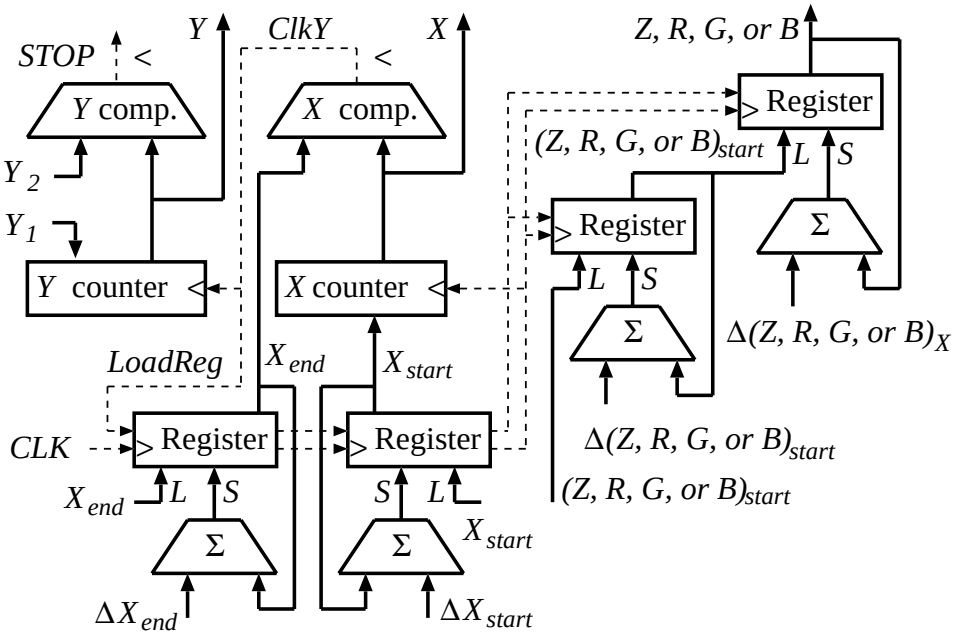


Figure 4.5: Hardware implementation of Gouraud shading and z-buffer algorithms

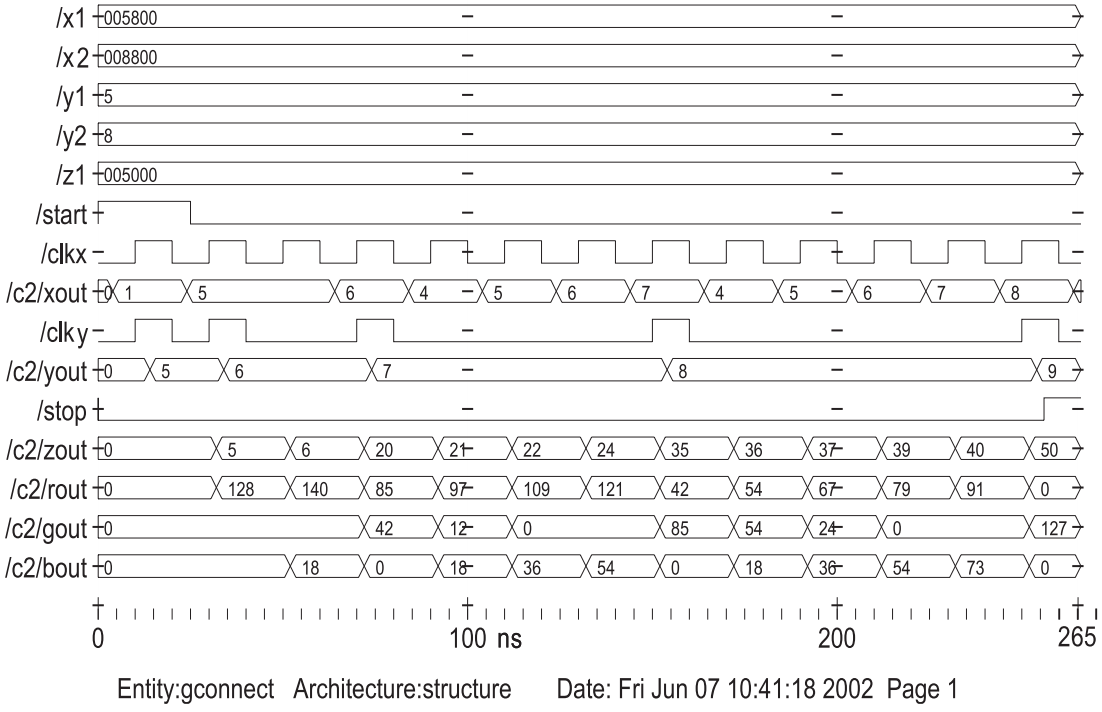
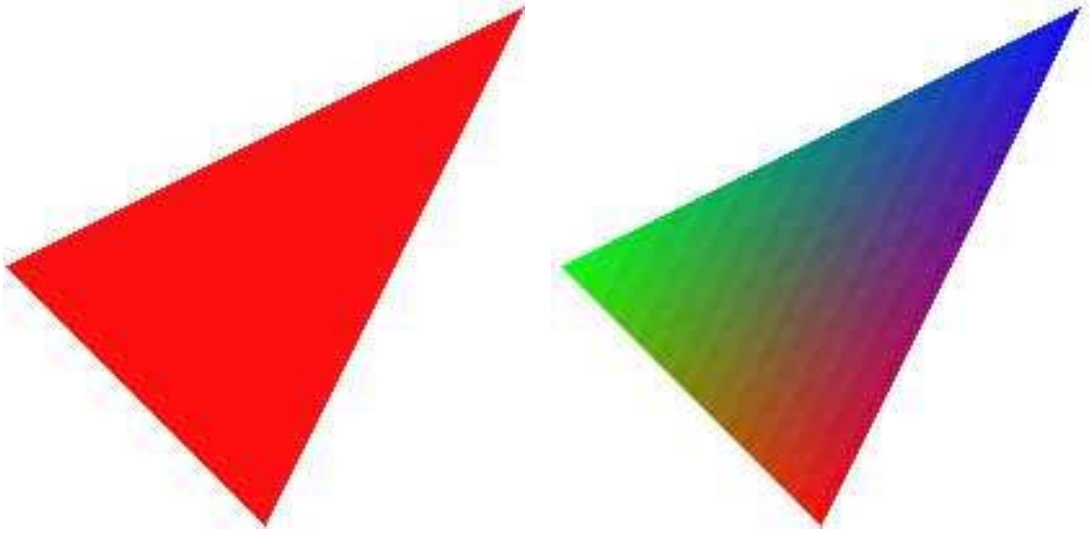


Figure 4.6: Timing diagram of the hardware implementation of Gouraud shading and z-buffer algorithms



Constant shading Gouraud shading

Figure 4.7: An interpolation of color in a triangle, with constant and Gouraud shadings

Chapter 5

Drawing triangles with Phong shading

Phong shading implies that at every pixel the vectors being involved in the BRDF are interpolated, normalized and their dot product is computed, then substituted into the simplified rendering equation:

$$I^{out} = I^e + k_a \cdot I^a + \sum_{l=1}^{k} \mathcal{R}_l \cdot I_l^{in}, \tag{5.1}$$

where I^e is the self emission, k_a and I^a are the ambient reflection parameter and ambient intensity respectively, and

$$\mathcal{R}_l = (k_d \cdot \cos\theta_{\vec{L}_l}) + (k_s \cdot \cos^n \delta)$$

is the cosine weighted BRDF, where $\theta_{\vec{L}_l}$ is the angle between the surface normal $\vec{N}$ and the direction of the lth light source $\vec{L}$, and δ is the angle between the halfway vector $\vec{H}$ and the surface normal $\vec{N}$. According to this formula, the rendering equation requires the computation of the angle of vectors that vary inside the triangle. In order to emphasize this, I^{out} can be regarded as a function of $\vec{N}$, $\vec{H}$ and $\vec{L}$ vectors (note that this function depends on as many $\vec{H}$ and $\vec{L}$ vectors as many light sources exist). Alternatively, I^{out} can be supposed to be the function of angles $\theta_{\vec{L}_l}$ and δ. Thus Phong shading can be interpreted as the calculation of these vectors and angles and then the evaluation of the simplified rendering equation.

To be general, let us consider the interpolation of two vectors $\vec{u}$ and $\vec{v}$ that can be any from the light vector $\vec{L}$, normal vector $\vec{N}$, viewing vector $\vec{V}$, etc. on a single scan-line.

Suppose that the vectors vary according to a linear function. The generic formulae of the computation of the cosine of the angle between $\vec{u}(t)$ and $\vec{v}(t)$ are then:

$$\begin{aligned}
\vec{u}(t) &= (1-t)\vec{u}_{start} + t\vec{u}_{end}, \\
\vec{u}^0 &= \frac{\vec{u}(t)}{|\vec{u}(t)|}, \\
\vec{v}(t) &= (1-t)\vec{v}_{start} + t\vec{v}_{end}, \\
\vec{v}^0 &= \frac{\vec{v}(t)}{|\vec{v}(t)|}, \\
\cos\theta &= \vec{u}^0 \cdot \vec{v}^0.
\end{aligned} \tag{5.2}$$

Note that this method requires a lot of pixel level operations, including linear interpolation of vectors and the evaluation of the rendering equation. Since dot products provide the cosine angle only if the vectors are unit vectors, normalization is required which involves 3 multiplications, 2 additions, a square root and 3 divisions. These complex operations are rather expensive computationally and make Phong shading slow and inappropriate for real time hardware rendering.

The superior rendering quality of Phong shading forced research to try to find a reasonable compromise between Gouraud and Phong algorithms, to keep the image quality but also to allow hardware implementation. In Textronix terminals, for example, the method called pseudo-Phong shading was implemented. Pseudo-Phong shading recursively decomposed the triangles into small triangles setting the vectors at the vertices according to a linear formula, and used Gouraud shading when the small triangles are rendered. If the size of the small triangles are comparable to the size of the pixels, then this corresponds to Phong shading. However, when they are close to the original triangle, this corresponds to Gouraud shading. Another family of algorithms used highlight tests [Wat89] to determine whether or not a specular highlight intersects the triangle. If there is no intersection, then Gouraud shading is used, otherwise the triangle is rendered with Phong shading. Duff [Duf79] extended the incremental approach of Gouraud shading to Phong shading. The simplification using Taylor's approximation proposed in [BW86]. This approach assumed that the Phong-Blinn reflection model is used. The determination of the derivatives of the reflected radiance is quite complicated and requires expensive computation, and this computation must be repeated at each pixel for diffuse and specular reflections and for each lightsource. Besides, according to the nature of Taylor's series, the approximation is good around the point where the derivatives were computed. Neighboring triangles may have different color variation on their edges, which leads to Mach banding over the edges of the triangles. Claussen [Cla90] compared different simplification strategies of the Phong illumination formulae and vector interpolation [BERW97]. Spherical interpolation elegantly traces back the interpolation to the interpolation of a single angle inside a scan-line [KB89]. However, finding the parameters of a scan-line is also rather complicated where it requires vector computation for diffuse and specular reflections and the evaluation of the rendering equation at each pixel and for each light-source. The computational cost is also proportional to the number of light sources.

The following sections review the possible interpolation strategies.

5.1 Normals shading

This is an approximation algorithm for speeding up Phong shading where no time-consuming normalization is needed, but some artifacts will occur in the image [Cla90] [HBB01] .

$$\begin{aligned}
\vec{L}(t) &\approx (1-t)\vec{L}_{start} + t\vec{L}_{end}, \\
\vec{N}(t) &\approx (1-t)\vec{N}_{start} + t\vec{N}_{end}, \\
\vec{H}(t) &\approx (1-t)\vec{H}_{start} + t\vec{H}_{end}, \\
C(t) &= I^{out}(\vec{N}(t)\cdot\vec{L}(t), \vec{N}(t)\cdot\vec{H}(t)).
\end{aligned} \tag{5.3}$$

This approach saves the time consuming normalization step. This, of course, results in incorrect cosine angles, which can be tolerated if the vectors do not change intensively on the triangles [DWS+88].

5.2 Dot product interpolation

Dot product interpolation is a reduced type of Phong shading [Cla90], which is intermediate in complexity between Gouraud shading and Phong shading. It is used to avoid the expensive computation and normalization of any of the direction vectors.

$$\begin{aligned} \vec{N}(t) \cdot \vec{L}(t) &\approx (1-t)(\vec{N} \cdot \vec{L})_{start} + t(\vec{N} \cdot \vec{L})_{end}, \\ \vec{N}(t) \cdot \vec{H}(t) &\approx (1-t)(\vec{N} \cdot \vec{H})_{start} + t(\vec{N} \cdot \vec{H})_{end}, \\ C(t) &= I^{out}(\vec{N}(t) \cdot \vec{L}(t), \vec{N}(t) \cdot \vec{H}(t)). \end{aligned} \tag{5.4}$$

Dot product interpolation applies a linear interpolation for the cosine angles ($\cos\theta_{\vec{L}} = \vec{N} \cdot \vec{L}$). It is as good as this cosine function can be assumed to be linear.

5.3 Polar angles interpolation

Polar angle interpolation interpolates the polar angles:

$$\begin{aligned} \theta_{start} &= \arccos(\vec{N} \cdot \vec{L})_{start}, \\ \theta_{end} &= \arccos(\vec{N} \cdot \vec{L})_{end}, \\ \delta_{start} &= \arccos(\vec{N} \cdot \vec{H})_{start}, \\ \delta_{end} &= \arccos(\vec{N} \cdot \vec{H})_{end}, \\ \theta(t) &= (1-t)\theta_{start} + t\theta_{end}, \\ \delta(t) &= (1-t)\delta_{start} + t\delta_{end}, \\ C(t) &= I^{out}(\cos\theta(t), \cos\delta(t)). \end{aligned} \tag{5.5}$$

5.4 Angular interpolation

The angular rotations of direction vectors $\vec{L}, \vec{N}$ or $\vec{H}$ are linearly related to the position along a straight line across the triangle. Vectors interpolated according to this assumption have a constant length and are all in one plane, the plane spanned by start and end vector (Figure 5.1).

The interpolation will be done by two steps, first the vector is interpolated along the edges; next, the resulting vectors are used for interpolation along the span.

For the sake of simplicity, we shall consider only diffuse reflections, thus only the angle of $\vec{L}$ and $\vec{N}$ is used. For each $\vec{L}$ and $\vec{N}$, there is a mapping of the triangle on the unisphere indicating the range of that vector across the triangle. A scan-line across the triangle is mapped on two circular paths, indicating the variation of $\vec{L}$ and $\vec{N}$ along the scan-line. These paths,

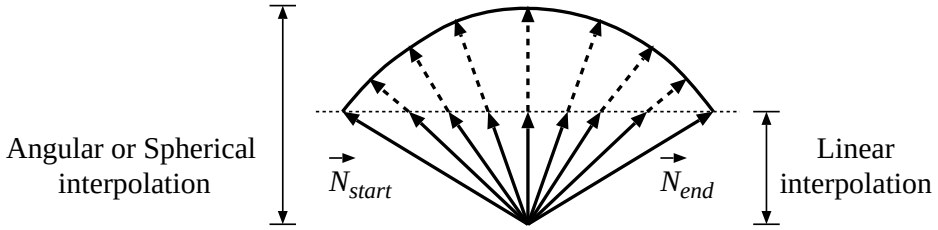


Figure 5.1: Comparison of linear and spherical interpolation of direction vectors

from $\vec{L}_{start}$ to $\vec{L}_{end}$ and from $\vec{N}_{start}$ to $\vec{N}_{end}$, are parts of great-circles. These two great-circles intersect at S. Let γ be the angle between the two great-circles, l be the angle between S and $\vec{L}_{start}$, n be the angle between S and $\vec{N}_{start}$. Having t linearly changing along the scan-line, we define l_t to be the angle between $\vec{L}_{start}$ and $\vec{L}_t$, and n_t to be the angle between $\vec{N}_{start}$ and $\vec{N}_t$. $\vec{L}_t$ and $\vec{N}_t$ are linearly interpolated along the scan-line. With this we have the spherical triangle S, L_t, N_t, dependent on t (Figure 5.2). For this triangle a standard formula is given by the spherical trigonometry, which leads to the following relation between $\theta_{\vec{L}_t}$ (the angle between $\vec{L}_t$ and $\vec{N}_t$) and the linearly incremented angles l_t and n_t [KB89]:

$$\begin{aligned} \cos\theta_{\vec{L}_t} &= \cos(l+l_t)\cos(n+n_t) + \sin(l+l_t)\sin(n+n_t)\cos\gamma, \\ C(t) &= I^{out}(\cos\theta_{\vec{L}_t}). \end{aligned} \tag{5.6}$$

Note that l, n, and γ are constant along the scan-line:

$$\cos\gamma = (\vec{L}_{start} \times \vec{L}_{end}) \cdot (\vec{N}_{start} \times \vec{N}_{end}). \tag{5.7}$$

In [KB89] it was also shown that instead of interpolating the two independently varying vectors $\vec{L}$ and $\vec{N}$, only one vector can be interpolated. Realizing that only the relative position of vectors $\vec{L}$ and $\vec{N}$ is of interest not their absolute position. Vector $\vec{Q}$ is defined at each vertex of the triangle that is found by rotating $\vec{L}$ around the same axis and with the same angle as needed to rotate $\vec{N}$ to be aligned with a fixed vector $\vec{O}$, (where $\vec{O} = (0,0,1)$). In this case only vector $\vec{Q}$ is interpolated across the formed triangle $Q_t\ O\ S$, where $\vec{S}$ is the third vertex lies on the great circle, such that the arc segment $Q_t\ S$ is perpendicular to the arc segment $S\ O$, and α is defined as the changing angle between $\vec{Q}_t$ and $\vec{O}$.

In this special case we have:

$$\begin{aligned} \cos\alpha_t &= \cos(q+q_t)\cos s, \\ C(t) &= I^{out}(\cos\alpha_t), \end{aligned} \tag{5.8}$$

where q_t is the angle between $\vec{Q}_{start}$ and $\vec{Q}_t$, q is the angle between $\vec{Q}_{start}$ and $\vec{S}$, and s is the angle between $\vec{S}$ and $\vec{O}$.

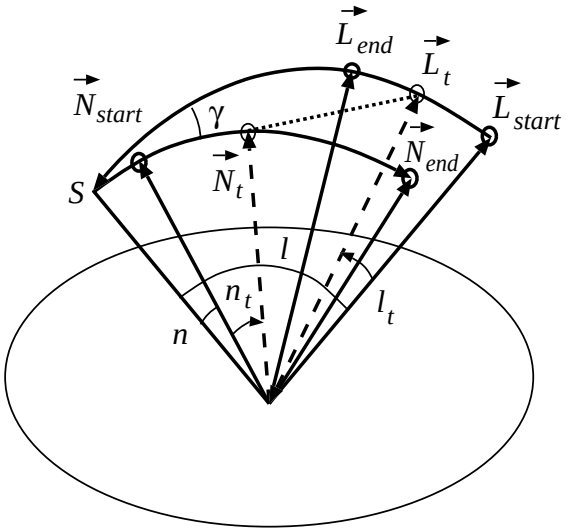


Figure 5.2: Vectors and angles variations along the mapped scan-line on two circular paths

5.5 Phong shading and Taylor's series approximation

For the sake of simplicity, assume that the light source is at infinity and the interpolated triangles are planar, so $\vec{L}^0$ is independent of the surface point, resulting that the linear interpolation is only dependent on $\vec{N}^0$ in diffuse reflection and on $(\vec{N}^0 \cdot \vec{H}^0)^n$ in specular reflection, considering the viewing direction $\vec{V}^0$ at finite position.

Phong shading (Blinn model) can be implemented more efficiently by combining the following reflection and interpolation equations of successive X and Y [BW86]:

$$\begin{aligned}
\vec{N}(X,Y) &= \vec{A}X + \vec{B}Y + \vec{C}, \\
\vec{H}(X,Y) &= \vec{D}X + \vec{E}Y + \vec{F}, \\
\vec{N}^0 &= \frac{\vec{N}}{|\vec{N}|}, \\
\vec{H}^0 &= \frac{\vec{H}}{|\vec{H}|}, \\
\cos\theta(X,Y) &= (\vec{N}^0 \cdot \vec{L}^0), \\
\cos\delta(X,Y) &= (\vec{N}^0 \cdot \vec{H}^0),
\end{aligned} \tag{5.9}$$

where $\vec{A}$, $\vec{B}$, and $\vec{C}$ are chosen to interpolate the normal $\vec{N}$ across the triangle and $\vec{D}, \vec{E}$, and $\vec{F}$ are chosen to interpolate the halfway vector $\vec{H}$ across the triangle.

5.5.1 Diffuse part for directional light sources

Let us combine the two-variate linear (reflection and interpolation) equations in Equation 5.9 for diffuse reflection:

$$\cos\theta = \left(\vec{L}^0 \cdot \frac{\vec{A}X + \vec{B}Y + \vec{C}}{|\vec{A}X + \vec{B}Y + \vec{C}|}\right). \tag{5.10}$$

Performing the indicated dot products and expanding the vector magnitude yields:

$$\cos\theta = \frac{aX + bY + c}{\sqrt{dX^2 + eXY + fY^2 + gX + hY + i}}, \tag{5.11}$$

where

$$\begin{aligned} a &= \left(\frac{\vec{A}\cdot\vec{L}}{|\vec{L}|}\right), \quad b = \left(\frac{\vec{B}\cdot\vec{L}}{|\vec{L}|}\right), \quad c = \left(\frac{\vec{C}\cdot\vec{L}}{|\vec{L}|}\right), \\ d &= \vec{A}\cdot\vec{A}, \quad e = 2(\vec{A}\cdot\vec{B}), \quad f = \vec{B}\cdot\vec{B}, \\ g &= 2(\vec{A}\cdot\vec{C}), \quad h = 2(\vec{B}\cdot\vec{C}), \quad i = \vec{C}\cdot\vec{C}. \end{aligned}$$

Applying second order Taylor's series approximation, and shifting the triangle to the coordinate origin yields to the following quadratic function:

$$\cos\theta(X,Y) = T_5X^2 + T_4XY + T_3Y^2 + T_2X + T_2Y + T_0, \tag{5.12}$$

where

$$\begin{aligned} T_5 &= \frac{3cg^2 - 4cdi - 4agi}{8i^2\sqrt{i}}, \\ T_4 &= \frac{3cgh - 2cei - 2bgi - 2ah}{4i^2\sqrt{i}}, \\ T_3 &= \frac{3ch^2 - 4cfi - 4bhi}{8i^2\sqrt{i}}, \\ T_2 &= \frac{2ai - cg}{2i\sqrt{i}}, \\ T_1 &= \frac{2bi - ch}{2i\sqrt{i}}, \\ T_0 &= \frac{c}{\sqrt{i}}. \end{aligned}$$

Note that for every pixel the resulting value of $\cos\theta$ should be multiplied by $k_d \cdot I^{in}$.

5.5.2 Specular part for directional light sources

Let us combine the two-variate linear (reflection and interpolation) equations in Equation 5.9 for specular reflection:

$$\cos\delta = \frac{(\vec{A}X + \vec{B}Y + \vec{C}) \cdot (\vec{D}X + \vec{E}Y + \vec{F})}{|\vec{A}X + \vec{B}Y + \vec{C}| \cdot |\vec{D}X + \vec{E}Y + \vec{F}|}. \tag{5.13}$$

Performing the indicated dot products and expanding the vector magnitude yields:

$$\cos\delta = \frac{aX^2 + bXY + cY^2 + dX + eY + f}{\sqrt{(gX^2 + hXY + iY^2 + jX + kY + l) \cdot (mX^2 + nXY + oY^2 + pX + qY + r)}}, \tag{5.14}$$

where

$$\begin{aligned}
a &= \vec{A}\cdot\vec{D}, \quad b = \vec{A}\cdot\vec{E} + \vec{B}\cdot\vec{D}, \quad c = \vec{B}\cdot\vec{E}, \quad d = \vec{A}\cdot\vec{F} + \vec{C}\cdot\vec{D}, \quad e = \vec{B}\cdot\vec{F} + \vec{C}\cdot\vec{E}, \\
f &= \vec{C}\cdot\vec{F}, \quad g = \vec{A}\cdot\vec{A}, \quad h = 2(\vec{A}\cdot\vec{B}), \quad i = \vec{B}\cdot\vec{B}, \quad j = 2(\vec{A}\cdot\vec{C}), \quad k = 2(\vec{B}\cdot\vec{C}), \\
l &= \vec{C}\cdot\vec{C}, \quad m = \vec{D}\cdot\vec{D}, \quad n = 2(\vec{D}\cdot\vec{E}), \quad o = \vec{E}\cdot\vec{E}, \quad p = 2(\vec{D}\cdot\vec{F}), \\
q &= 2(\vec{E}\cdot\vec{F}), \quad r = \vec{F}\cdot\vec{F}.
\end{aligned}$$

Applying second order Taylor's series approximation, and shifting the triangle to the coordinate origin yields to the following quadratic function:

$$\cos\delta(X,Y) = T_5X^2 + T_4XY + T_3Y^2 + T_2X + T_1Y + T_0F, \tag{5.15}$$

where

$$\begin{aligned}
T_5 &= \frac{8al^2r^2 - 4djlr^2 - 4fglr^2 + 3fj^2r^2 - 4dl^2pr + 2fjlpr - 4fl^2mr + 3fl^2p^2}{8l^2r^2\sqrt{lr}}, \\
T_4 &= \frac{4bl^2r^2 - 2dklr^2 - 2ejlr^2 - 2fhlr^2 + 3fjkr^2 - 2dl^2qr}{4l^2r^2\sqrt{lr}} \\
&+ \frac{-fjlqr - 2el^2pr + fklpr - 2f^2nr + 3fl^2pq}{4l^2r^2\sqrt{lr}}, \\
T_3 &= \frac{8cl^2r^2 - 4eklr^2 - 4filr^2 + 3fk^2r^2 - 4el^2qr + 2fklqr - 4fl^2or + 3fl^2q^2}{8l^2r^2\sqrt{lr}}, \\
T_2 &= \frac{2dlr - fjr - flp}{2lr\sqrt{lr}}, \\
T_1 &= \frac{2elr - fkr - flq}{2lr\sqrt{lr}}, \\
T_0 &= \frac{f}{\sqrt{lr}}.
\end{aligned}$$

Note that for every pixel the resulting value of $\cos\delta$ should be powered by the required n which can be realized by a memory table, then it will be multiplied by $k_s \cdot I^{in}$.

Chapter 6

Spherical interpolation

Spherical interpolation traces back the vector interpolation to the interpolation of a single angle inside a scan-line. It interpolates linearly the angles between the start and the end of the vectors $\vec{L}$, normals $\vec{N}$, and vectors $\vec{H}$ for each scan-line, resulting that no normalization is needed.

Suppose that we intend to interpolate between two unit vectors $\vec{u}_1$ and $\vec{u}_2$, in a way that the interpolant $\vec{u}(t)$ is moving uniformly between the two vectors and its length is always one, (Figure 6.1). An appropriate interpolation method must generate the great arc between $\vec{u}_1$ and $\vec{u}_2$, and as can easily be shown, this great arc has the following form:

$$\vec{u}(t) = \frac{\sin(1-t)\gamma}{\sin\gamma} \cdot \vec{u}_1 + \frac{\sin t\gamma}{\sin\gamma} \cdot \vec{u}_2, \tag{6.1}$$

where $\cos\gamma = \vec{u}_1 \cdot \vec{u}_2$ (Figure 6.1).

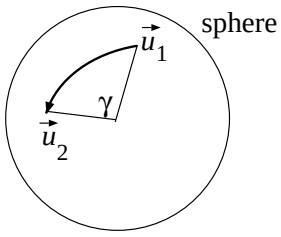


Figure 6.1: Interpolation of vectors on a unit sphere

In order to demonstrate that this really results in a uniform interpolation, the following equations must be proven for $\vec{u}(t)$:

$$|\vec{u}(t)|^2 = 1, \quad \vec{u}_1 \cdot \vec{u}(t) = \cos t\gamma, \qquad \vec{u}_2 \cdot \vec{u}(t) = \cos(1-t)\gamma. \tag{6.2}$$

That is, the interpolated vector is really on the surface of the sphere, and the angle of rotation is a linear function of parameter t.

Let us first prove the second assertion (the third can be proven similarly):

$$\vec{u}_1 \cdot \vec{u}(t) = \frac{\sin(1-t)\gamma}{\sin\gamma} + \frac{\sin t\gamma}{\sin\gamma} \cdot \cos\gamma = \frac{\sin\gamma \cdot \cos t\gamma}{\sin\gamma} - \frac{\sin t\gamma \cdot \cos\gamma}{\sin\gamma} + \frac{\sin t\gamma}{\sin\gamma} \cdot \cos\gamma = \cos t\gamma. \tag{6.3}$$

Concerning the first assertion, i.e. the norm of the interpolated vector, we can use the definition of the norm and the previous result, thus we obtain:

$$\begin{aligned} |\vec{u}(t)|^2 &= \vec{u}(t) \cdot \vec{u}(t) = \left(\frac{\sin(1-t)\gamma}{\sin\gamma} \cdot \vec{u}_1 + \frac{\sin t\gamma}{\sin\gamma} \cdot \vec{u}_2 \right) \cdot \vec{u}(t) \\ &= \frac{\sin(1-t)\gamma}{\sin\gamma} \cdot \cos t\gamma + \frac{\sin t\gamma}{\sin\gamma} \cdot \cos(1-t)\gamma = \frac{\sin\left((1-t)\gamma + t\gamma\right)}{\sin\gamma} = 1. \end{aligned} \tag{6.4}$$

6.1 Independent spherical interpolation of a pair of vectors

In our case the light, normal and halfway vectors should be interpolated:

$$\begin{aligned} \vec{L}^0(t) &= \frac{\sin(1-t)l}{\sin l} \cdot \vec{L}_{start} + \frac{\sin tl}{\sin l} \cdot \vec{L}_{end}, \\ \vec{N}^0(t) &= \frac{\sin(1-t)n}{\sin n} \cdot \vec{N}_{start} + \frac{\sin tn}{\sin n} \cdot \vec{N}_{end}, \\ \vec{H}^0(t) &= \frac{\sin(1-t)h}{\sin h} \cdot \vec{H}_{start} + \frac{\sin th}{\sin h} \cdot \vec{H}_{start}, \\ C(t) &= I^{out}(\vec{L}^0(t) \cdot \vec{N}^0(t), \vec{N}^0(t) \cdot \vec{H}^0(t)). \end{aligned} \tag{6.5}$$

We will consider only diffuse reflection $\cos\theta_{\vec{L}} = \vec{L}^0 \cdot \vec{N}^0$ for implementation:

$$\vec{L}^0(t) \cdot \vec{N}^0(t) = \left(\frac{\sin(1-t)l}{\sin l} \cdot \vec{L}_{start} + \frac{\sin tl}{\sin l} \cdot \vec{L}_{end} \right) \cdot \left(\frac{\sin(1-t)n}{\sin n} \cdot \vec{N}_{start} + \frac{\sin tn}{\sin n} \cdot \vec{N}_{end} \right)$$

$$\begin{aligned} &= \frac{\sin(1-t)l \cdot sin(1-t)n \cdot (\vec{L}_{start} \cdot \vec{N}_{start}) + \sin tl \cdot \sin tn \cdot (\vec{L}_{end} \cdot \vec{N}_{end})}{\sin l \cdot \sin n} \\ &+ \frac{\sin(1-t)l \cdot \sin tn \cdot (\vec{L}_{start} \cdot \vec{N}_{end}) + \sin tl \cdot \sin(1-t)n \cdot (\vec{L}_{end} \cdot \vec{N}_{start})}{\sin l \cdot \sin n} \\ &= \frac{a \cdot \sin(1-t)l \cdot \sin(1-t)n + b \cdot \sin tl \cdot \sin tn}{\sin l \cdot \sin n} \\ &+ \frac{c \cdot \sin(1-t)l \cdot \sin tn + d \cdot \sin tl \cdot \sin(1-t)n}{\sin l \cdot \sin n}. \end{aligned}$$

This formula can be evaluated by sine tables. However, there is an even more effective approach which takes into account that we are interested in the angle of the two interpolated vectors, thus they do not have to be independently interpolated. This approach is discussed in the following section.

6.2 Simultaneous spherical interpolation of a pair of vectors

Having discussed how vectors can be interpolated without modifying their length, we can start examining how the angle between two interpolated vectors can be determined. Let us assume that $\vec{u}(t)$ is interpolated from $\vec{u}_1$ to $\vec{u}_2$ while $\vec{v}(t)$ is interpolated from $\vec{v}_1$ to $\vec{v}_2$, and we are interested in $\cos\theta(t) = \vec{u}(t) \cdot \vec{v}(t)$ (left of Figure 6.2).

One obvious possibility is to use the previous results separately for $\vec{u}(t)$ and $\vec{v}(t)$ and to compute the dot product for each t. However, we can realize that a similar interpolation can be obtained keeping one vector — say $\vec{v}_1$ — fixed and the other is rotated by the composition of its own transformation and the inverse of the transformation of the other vector (right of Figure 6.2). It means that while $\vec{v}'(t) = \vec{v}_1$ is fixed, $\vec{u}'(t)$ is interpolated between $\vec{u}_1$ and $\vec{u}'_2$ which is obtained by rotating $\vec{u}_2$ by the inverse of the rotation from $\vec{v}_1$ to $\vec{v}_2$.

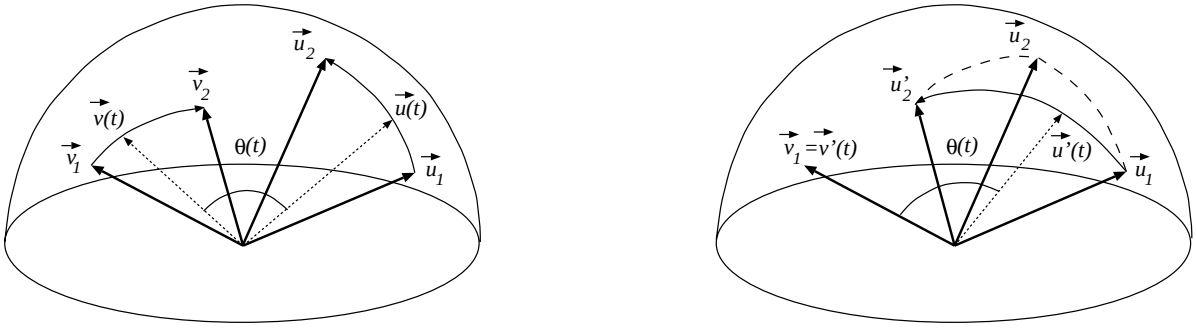


Figure 6.2: Interpolation of two vectors

The new end point $\vec{u}'_2$ can, for instance, be expressed by quaternion multiplications [SK95]. The unit quaternion that rotates $\vec{v}_1$ to $\vec{v}_2$ is:

$$q = \left[\cos\frac{v}{2}, \sin\frac{v}{2} \cdot \frac{\vec{v}_1 \times \vec{v}_2}{|\vec{v}_1 \times \vec{v}_2|}\right] = \left[\cos\frac{v}{2}, \sin\frac{v}{2} \cdot \frac{\vec{v}_1 \times \vec{v}_2}{\sin v}\right],$$

where v is the angle between $\vec{v}_1$ and $\vec{v}_2$. Applying the inverse of this quaternion to $\vec{u}_2$ we get:

$$[0, \vec{u}'_2] = q^{-1} \cdot [0, \vec{u}_2] \cdot q = \left[0, \vec{u}_2 - (\vec{v}_1 \times \vec{v}_2) \times \vec{u}_2 + \frac{(\vec{v}_1 \times \vec{v}_2) \times ((\vec{v}_1 \times \vec{v}_2) \times \vec{u}_2)}{1 + \cos v}\right].$$

Vector $\vec{u}'(t)$ is obtained by spherical interpolation from $\vec{u}_1$ to $\vec{u}'_2$, thus the angle between this vector and the fixed $\vec{v}_1$ is:

$$\begin{aligned}
\cos\theta(t) &= \vec{u}'(t) \cdot \vec{v}_1 = \frac{\sin(1-t)\gamma}{\sin\gamma} \cdot (\vec{u}_1 \cdot \vec{v}_1) + \frac{\sin t\gamma}{\sin\gamma} \cdot (\vec{u}'_2 \cdot \vec{v}_1) \\
&= \frac{\sin(1-t)\gamma}{\sin\gamma} \cdot \cos\theta_1 + \frac{\sin t\gamma}{\sin\gamma} \cdot \cos\theta_2 \\
&= \cos t\gamma \cdot \cos\theta_1 + \sin t\gamma \cdot \frac{\cos\theta_2 - \cos\theta_1 \cdot \cos\gamma}{\sin\gamma}, \qquad (6.6)
\end{aligned}$$

where $\cos\gamma = \vec{u}_1 \cdot \vec{u}_2'$. Note that this interpolation does not give exactly the same values as interpolating the two vectors separately. Since the interpolation is only used for approximating the vectors, this is as acceptable as the separate spherical interpolation.

Let us express $\cos\theta_1$ and $(\cos\theta_2 - \cos\theta_1 \cdot \cos\gamma)/\sin\gamma$ by A and α, respectively, in the following way:

$$A \cdot \cos\alpha = \cos\theta_1, \qquad A \cdot \sin\alpha = \frac{\cos\theta_2 - \cos\theta_1 \cdot \cos\gamma}{\sin\gamma}. \tag{6.7}$$

Substituting these into Equation 6.6, we obtain:

$$\cos\theta(t) = A \cdot (\cos t\gamma \cdot \cos\alpha + \sin t\gamma \cdot \sin\alpha) = A \cdot \cos(t\gamma - \alpha). \tag{6.8}$$

Let us realize that the complex sequence of operations including the spherical interpolations of two vectors and the computation of their dot product have been traced back to the calculation of a single cosine value. Based on this simplification, even the hardware realization of Phong shading becomes possible, as detailed in the next section for the Blinn illumination model. Similar hardware architectures can be developed for other BRDFs as well.

6.3 Interpolation and Blinn BRDF calculation by hardware

Substituting Equation 6.8 into the reflected radiance formula (Equation 1.8), and assuming Blinn type BRDF and a single lightsource, we get:

$$I^{out} = I^e + k_a \cdot I^a + I^{in}(\vec{L}) \cdot k_d \cdot \cos\theta_{\vec{L}} + I^{in}(\vec{L}) \cdot k_s \cdot \cos^n\delta. \tag{6.9}$$

Let us consider the most difficult part, i.e. the calculation of the specular reflection:

$$I^{in}(\vec{L}) \cdot k_s \cdot \cos^n\delta = I^{in}(\vec{L}) \cdot k_s \cdot (\vec{N} \cdot \vec{H})^n = I^{in}(\vec{L}) \cdot k_s \cdot A^n \cdot \cos^n(t\gamma - \alpha). \tag{6.10}$$

The constant $I^e + k_a \cdot I^a$ can be added in a trivial way, while the diffuse part $I^{in}(\vec{L}) \cdot k_d \cdot \cos\theta_{\vec{L}}$ can be computed in a similar way without the exponentiation.

Factor $I^{in}(\vec{x}, \vec{L}) \cdot k_s \cdot A^n = C$ is constant in the scan-line, thus only $\cos^n(t\gamma - \alpha)$ should be computed pixel by pixel and the result should be multiplied by this constant C.

The computation of $\cos^n(t\gamma - \alpha)$ consists of three elementary operations: the calculation of $t(X) \cdot \gamma - \alpha$ from the actual pixel coordinate X, the application of the cosine function, and finally exponentiation with n. These operations are too complex to allow direct hardware implementation thus further simplifications are needed.

The $t(X) \cdot \gamma - \alpha$ term is a linear function, thus it is a primary candidate for the application of the incremental concept. The cosine and the exponentiation are a little bit more difficult. In fact, we could use two tables of tabulated function values for this purpose. This would work well for the cosine function since it is relatively flat, but the accurate representation of the exponentiation would require a large table, which should be reinitialized each time when n changes (note that the practical values of n can range from 2 to a few hundred). Thus a different approximation strategy is used.

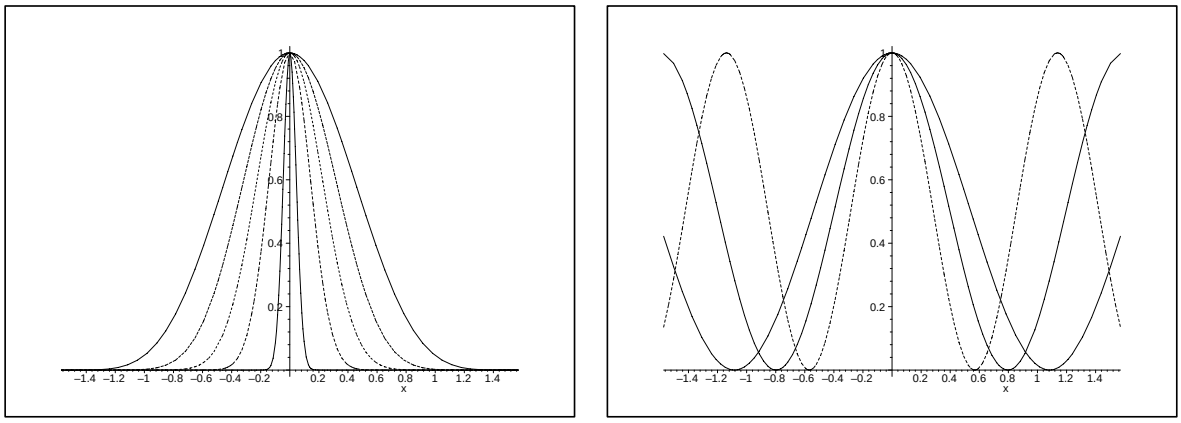


Figure 6.3: The bell shapes of $\cos^n x$ *for* $n = 5, 10, 20, 50, 500$ *(left) and of* $\cos^2 ax$ *for* $a = 1.45, 1.98, 2.76$ *(right)*

Looking at the bell shapes of the $\cos^n x$ functions for different n values (Figure 6.3), we can realize that these functions are approximately similar and can be transformed to each other by properly scaling the abscissa. For example, we can use the horizontally scaled versions of $\cos^2 x$, i.e. $\cos^2 ax$ to approximate $\cos^n x$ for arbitrary n. The reason of using the square of the cosine function is that n is greater than 2 in practical cases and the square cosine already has the bell shape caused by the inflection point. Thus our formal approximation is:

$$\cos^n x \approx \cos^2 ax \qquad \text{if } -\tfrac{\pi}{2a} \leq x \leq \tfrac{\pi}{2a}, \tag{6.11}$$

and zero otherwise. Parameter a should be found to maximize the accuracy for all possible x values. We can, for example, require the weighted integrals of the two functions to be equal in order to obtain the parameter a. Note that the approximation is exact for $x = 0$ regardless of the parameter a, that is the zero point of all cosines are fixed. This consideration makes it worth emphasizing the accuracy of larger x values when a is determined. Let us use the $\sin x$ weighting function, thus the criterion for determining a is:

$$\int_0^{\frac{\pi}{2}} \cos^n x \cdot \sin x \, dx = \int_0^{\frac{\pi}{2a}} \cos^2 ax \cdot \sin x \, dx. \tag{6.12}$$

Expressing these integrals in closed form, we get:

$$\frac{1}{n+1} = \frac{2a^2 \cdot (1 - \cos\frac{\pi}{2a}) - 1}{4a^2 - 1}.$$

This equation needs to be solved once for a set of values and the results can be stored in a table. A few representative results are shown in Table 6.1. The quality of the approximation is quite good as demonstrated by Figure 6.4.

n	2	5	10	20	50	100	500
a	1.0000	1.4502	1.9845	2.7582	4.3143	6.0791	13.5535

Table 6.1: Correspondence between n and a

Let us return to the computation of the reflected radiance. The

$$C \cdot \cos^n(t(X) \cdot \gamma - \alpha)$$

has been simplified to the evaluation of

$$C \cdot \cos^2(a(t(X) \cdot \gamma - \alpha)) = C \cdot \cos^2 \xi(X),$$

where

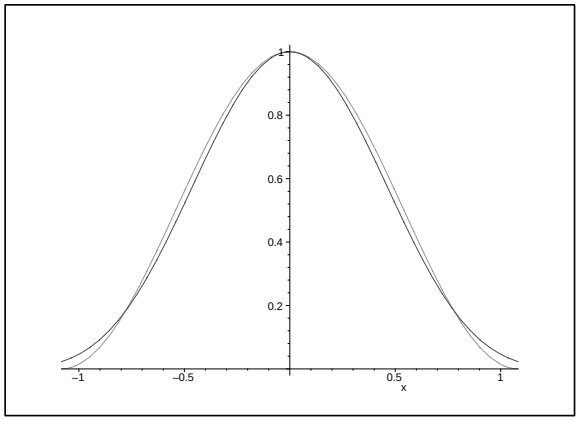
$$\xi(X) = a \cdot \gamma \cdot \frac{X - X_{start}}{X_{end} - X_{start}} - a \cdot \alpha = s \cdot X + b.$$

Since $\xi(X)$ is a linear function, it can conveniently be generated by the incremental concept. Its basic idea is that instead of computing ξ from X, it can be computed from the previous value, i.e. from $\xi(X-1)$. Recall that a complete scan-line is filled, that is when pixel X is shaded, the results of pixel $X-1$ are already available. In our case:

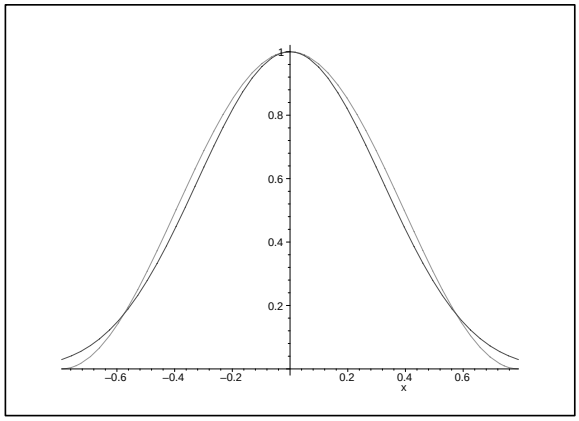
$$\xi(X) = \xi(X-1) + s,$$

thus the new value of ξ requires just a single addition.

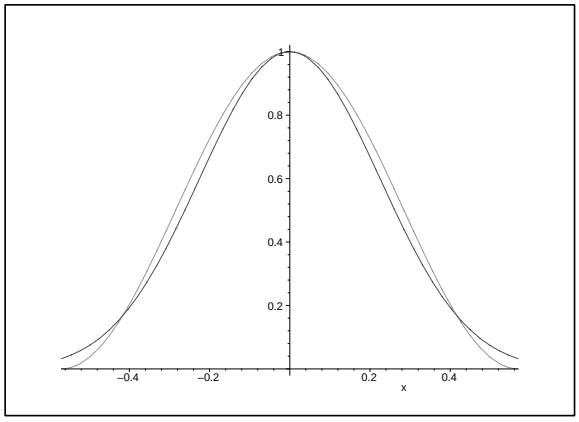
Having the ξ value generated, it should be input to the $\cos^2 \xi(X)$ function that can be implemented as a read only memory. The number of address and data bits of this memory, i.e. the number and the length of the words are determined from the requirement of accurate representation. Figure 6.5 shows the original $\cos^2 x$ function together with its table representations for different address and data bit numbers. Note that using six bit address and data, which means that our memory stores $2^6 \times 6 = 384$ bits, provides sufficient accuracy.



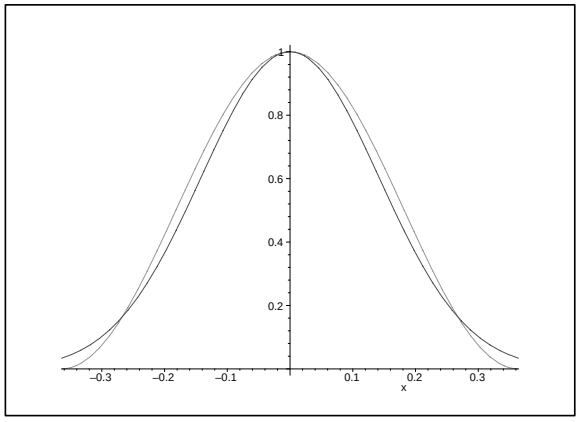
$n = 5$, $a = 1.4502$



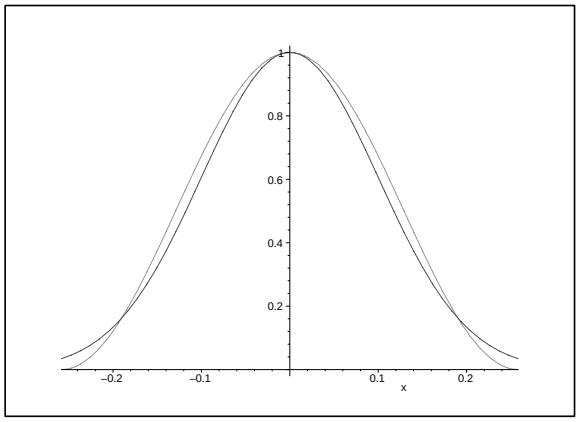
$n = 10$, $a = 1.9845$



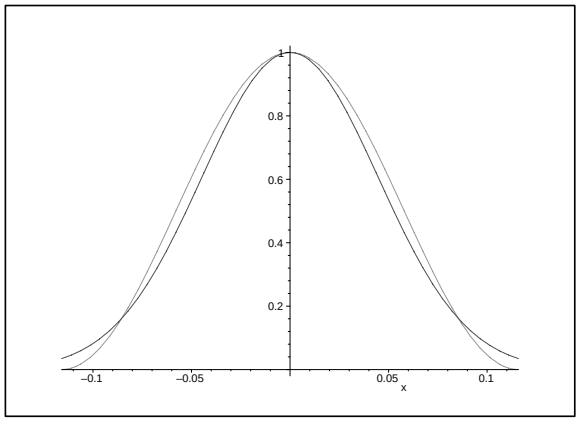
$n = 20$, $a = 2.7582$



$n = 50$, $a = 4.3143$



$n = 100$, $a = 6.0791$



$n = 500$, $a = 13.5535$

Figure 6.4: Approximation of $\cos^n x$ *by* $\cos^2 ax$

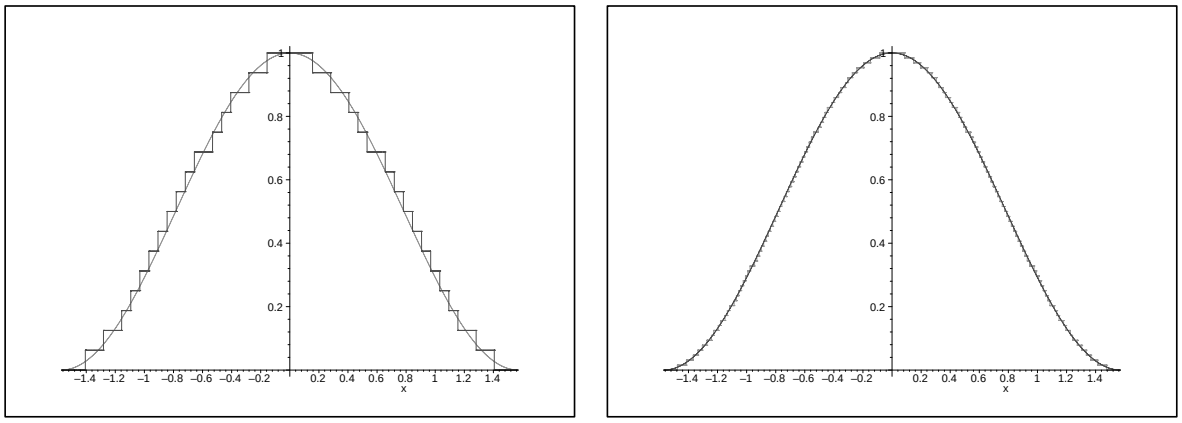


Figure 6.5: Quantization errors of the $\cos^2 ax$ function for 4 and 6 address/data bits verses the origonal $\cos^2 x$ function

The complete hardware is shown in Figure 6.6. The hardware has two parts, one for the diffuse and one for the specular components, and each part has two stages. In the specular part, the first stage is a linear interpolator, which provides the $\cos^2$ table with angle ξ, according to $\xi(X+1) = \xi(X) + s$. Since it has a register at its output, this stage can operate in parallel with the multiplier unit. Assuming white light sources and wavelength independent specular factor k_s, a single linear interpolator can be used for all color channels. However, the diffuse part, which is responsible for coloring, requires 3 channels. The cosine, and square cosine functions can be implemented by ROMs. At the initial phase, for each scan line, the constant parameters must be loaded into hardware. Then, for each step, the hardware will generate R, G, B values.

The VHDL specification is straightforward for the multiplicators and for the ROMs. Here, as an example, the linear interpolator is given by the following behavioral model:

```
ARCHITECTURE Behavior OF Line_interpolator IS
 SIGNAL add_Out, mpx_Out, reg_Out: bit_vector_12;
 BEGIN
  add_Out <= s + reg_Out AFTER t_add;
  reg_out <= mpx_out AFTER t_reg WHEN step'EVENT AND step = '1';
  ra <= reg_out(11 DOWNTO 6);
  mpx_process:
    PROCESS ( sxsb,add_out,init )
    BEGIN
      IF ( init = '1' ) THEN
        mpx_out <= sxsb AFTER t_mpx;
      ELSE
        mpx_out <= add_out AFTER t_mpx;
      END IF;
    END PROCESS;
 END Behavior;
```

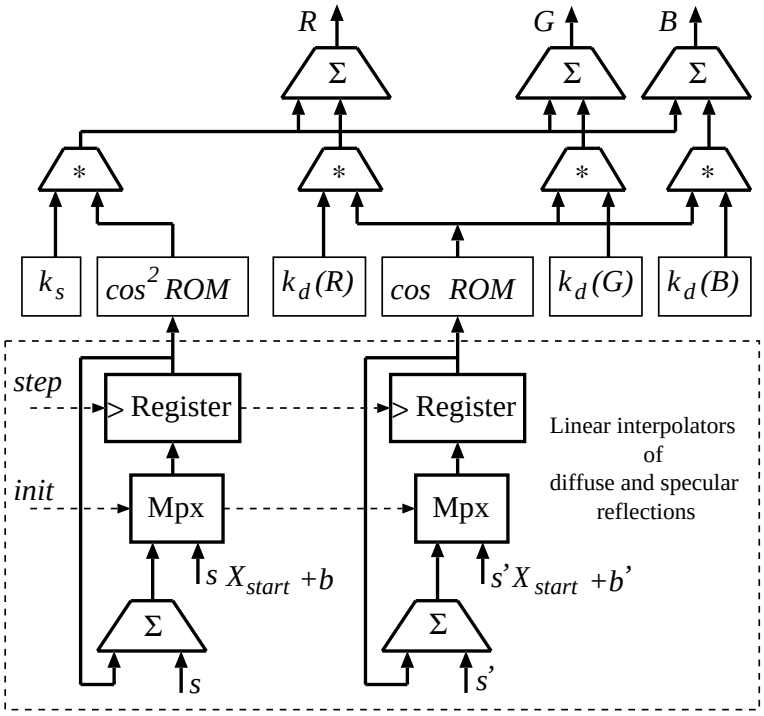


Figure 6.6: Hardware implementation of Phong shading using spherical interpolation

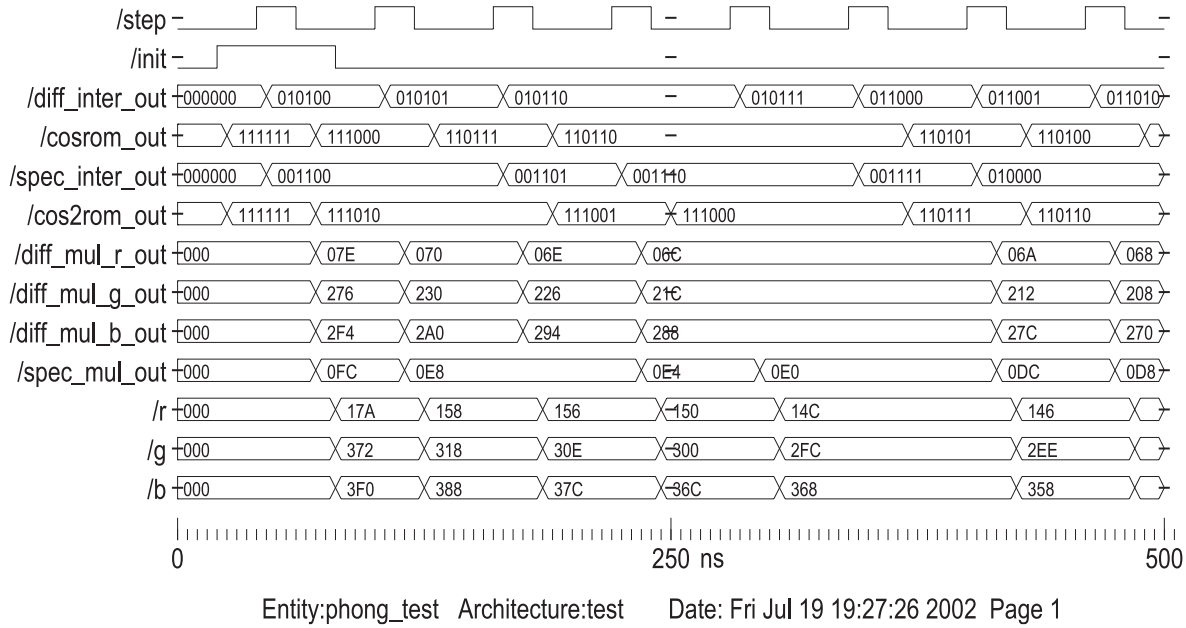


Figure 6.7: Timing diagram of the hardware of Phong shading using spherical interpolation

6.4 Simulation results

The proposed algorithm has been implemented first in C++ and tested as a software. First the difference between the simultaneous vector interpolation and the method of keeping one vector fixed while rotating the other vector by the composition of the two rotations was investigated, and we concluded that the results are visually indistinguishable. Then the quality of the $\cos^n x \approx \cos^2 ax$ approximation has been studied.

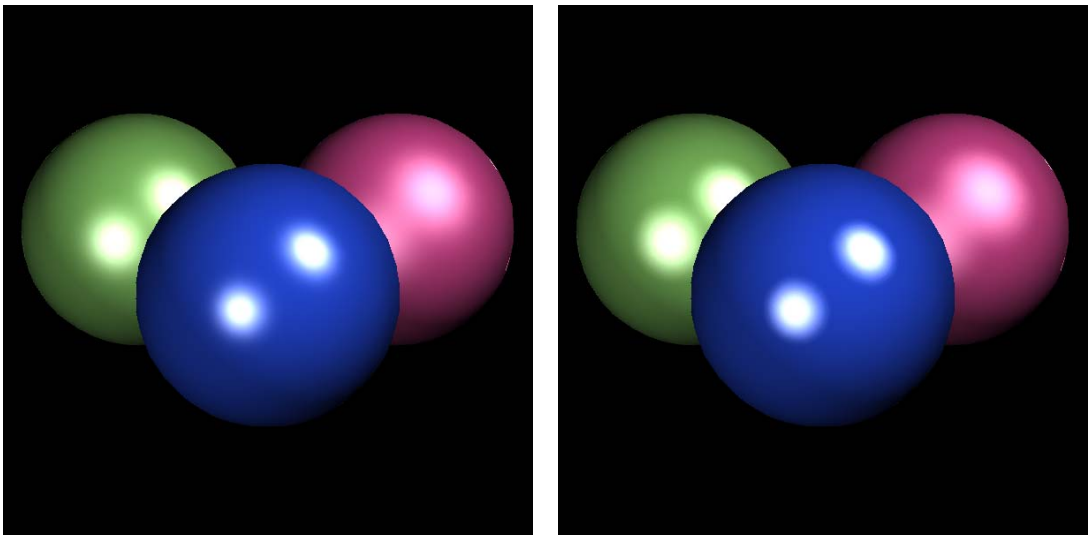


Figure 6.8: Evaluation of the visual accuracy approximation of the functions $\cos^n x$ *(left)* $\approx \cos^2 ax$ *(right). The shine* (n) *parameters of the rendered spheres are* $5, 10$ *and* 20

Note that the halos in the left image of Figure 6.8 obtained with the $\cos^n x$ function are slightly bigger but the centers are smaller. This is also obvious looking at the bell shapes of Figure 6.4 since the $\cos^2 ax$ is zero if $x = \pi/2a$ while $\cos^n x$ only converges to zero, while having the same integral.

Finally, the necessary precision, i.e. the number of bits, was determined. Since the $\cos^2 ax$ function is implemented as a memory, it is the most sensitive to the word length. Figure 6.9 shows the results assuming 4 and 6 bit precision, respectively, where a rather coarse surface tessellation was used to emphasize the possible interpolation errors. Note that with 4 bits the quantization errors are visible in the form of concentric halo circles around the highlight spots. However, these circles disappear when 6 bit precision is used. This also conforms with the quantization error functions of Figure 6.5.

Finally, the hardware was specified in VHDL and simulated in Model-Technology environment. The delay times are according to Xilinx Synthesis Technology real FPGA device. The timing diagram of the operation is shown by Figure 6.7. In this figure we can follow the overlapped operation of the two stages while the cycle time of the “step” signal is $60\ nsec$.

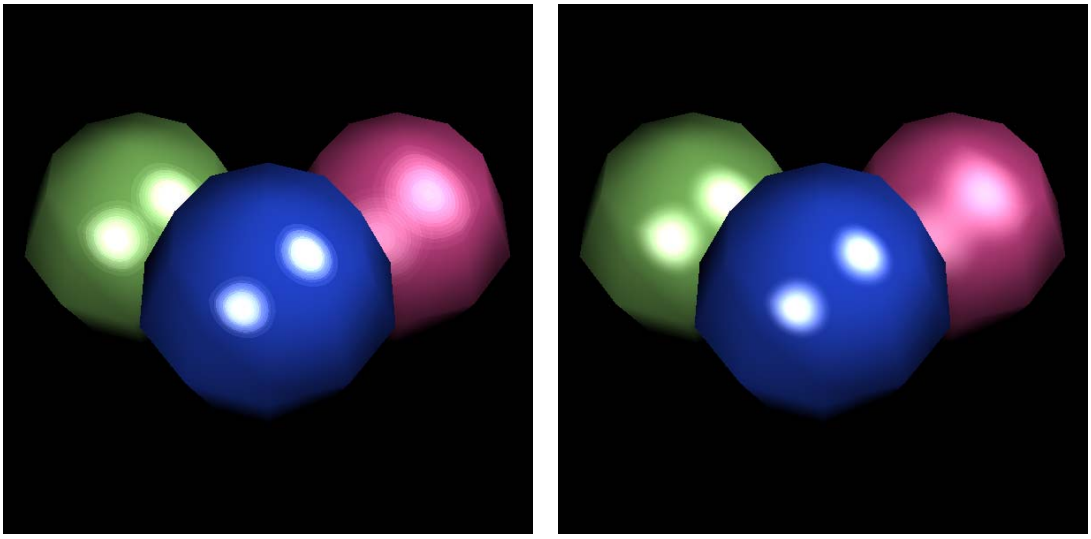


Figure 6.9: Rendering of coarsely tessellated spheres with the proposed spherical interpolation, 4 *bit precision (left) and* 6 *bit precision (right)*

Figure 6.10: The mesh of a chicken (left) and its image rendered by classical Phong shading (middle) and by the proposed spherical interpolation (right)

Chapter 7

Quadratic interpolation

As mentioned in Section 1.2.1, interpolation can be used to speed up the rendering of the triangle mesh, where the expensive computations take place just at the vertices and the data of the internal points are interpolated.

A simple interpolation scheme (Figure 10.1) would compute the color and linearly interpolate it inside the triangle (**Gouraud shading** [Gou71]). However, the core of the problem of Gouraud shading is that the color may be a strongly non-linear function of the pixel coordinates, especially if specular highlights occur on the triangle, and this non-linear function can hardly be well approximated by a linear function, thus linear interpolation may introduce severe artifacts.

The artifacts of Gouraud shading can be eliminated by a non-linear interpolation scheme (Figure 10.2) called **Phong shading** [Pho75]. In Phong shading, vectors used by the BRDFs in the rendering equation are interpolated from the real vectors at the vertices of the approximating triangle. The interpolated vectors are normalized and the rendering equation is evaluated at every pixel for diffuse and specular reflections and for each lightsource, which is rather time consuming. The main problem of Phong shading is that it requires complex operations on the pixel level, thus its hardware implementation is not suitable for real-time rendering.

Here we introduce a new approach called **quadratic interpolation** (Figure 10.3), which is in between Gouraud shading and Phong shading. The rendering equation is evaluated in a few representative points and the interpolation is done in color space as in Gouraud shading. However, the interpolation is non-linear, but rather quadratic. Since the general two-variate quadratic function has six degrees of freedom, thus six-knot points are needed to establish the interpolation formula without ambiguity. The rendering equation will be evaluated at six representative points on the triangle and the defined color at these points should be interpolated across the triangle.

Let us approximate the color inside the triangle by the following two-variate quadratic function:

$$I(X,Y) = T_5X^2 + T_4XY + T_3Y^2 + T_2X + T_1Y + T_0. \tag{7.1}$$

To find the unknown parameters $T_0, \ldots, T_5$, the color obtained from the rendering equation is substituted into this scheme at six points, and the six-variate linear equation is solved for the parameters. The selection of these representative points should take into account different

criteria. The error should be roughly uniform inside the triangle but less on the edges and on the vertices in order to avoid Mach banding. On the other hand, the resulting linear equation should be easy to solve in order to save computation time. An appropriate selection meeting both requirements uses the three vertices:

$$I(X_1, Y_1) = I_1,\ I(X_2, Y_2) = I_2,\ I(X_3, Y_3) = I_3,$$

and the other three points on the edges half way between the two vertices, as follows:

$$I\left(\frac{X_1+X_2}{2}, \frac{Y_1+Y_2}{2}\right) = I_{12},\ I\left(\frac{X_1+X_3}{2}, \frac{Y_1+Y_3}{2}\right) = I_{13},\ I\left(\frac{X_2+X_3}{2}, \frac{Y_2+Y_3}{2}\right) = I_{23}.$$

Translating the triangle to have its bottom vertex at the coordinate origin yields:

$$\begin{aligned}
I_1 &= T_0, \\
I_2 &= T_5 X_2^2 + T_4 X_2 Y_2 + T_3 Y_2^2 + T_2 X_2 + T_1 Y_2 + T_0, \\
I_{12} &= T_5 \frac{X_2^2}{4} + T_4 \frac{X_2 Y_2}{4} + T_3 \frac{Y_2^2}{4} + T_2 \frac{X_2}{2} + T_1 \frac{Y_2}{2} + T_0, \\
I_3 &= T_5 X_3^2 + T_4 X_3 Y_3 + T_3 Y_3^2 + T_2 X_3 + T_1 Y_3 + T_0, \\
I_{13} &= T_5 \frac{X_3^2}{4} + T_4 \frac{X_3 Y_3}{4} + T_3 \frac{Y_3^2}{4} + T_2 \frac{X_3}{2} + T_1 \frac{Y_3}{2} + T_0, \\
I_{23} &= T_5 \frac{(X_2+X_3)^2}{4} + T_4 \frac{X_2+X_3}{2} \cdot \frac{Y_2+Y_3}{2} + T_3 \frac{(Y_2+Y_3)^2}{4} \\
&+ T_2 \frac{X_2+X_3}{2} + T_1 \frac{Y_2+Y_3}{2} + T_0.
\end{aligned} \tag{7.2}$$

This system of linear equations can be solved in a straightforward way resulting in:

$$\begin{aligned}
T_0 &= I_1, \\
T_1 &= \frac{C_3 X_2 - C_2 X_3}{X_2 Y_3 - Y_2 X_3}, \\
T_2 &= \frac{C_2 Y_3 - C_3 Y_2}{X_2 Y_3 - Y_2 X_3}, \\
T_3 &= \frac{2C_{12} - T_5 X_2^2 - T_4 X_2 Y_2}{Y_2^2}, \\
T_4 &= \frac{(4C_{13} Y_2 - C_{23} Y_3)(2X_2^2 Y_3 - 2X_2 Y_2 X_3) - (4C_{12} Y_3 - C_{23} Y_2)(2Y_2 X_3^2 - 2X_2 X_3 Y_3)}{(2X_2^2 Y_3 - 2X_2 Y_2 X_3)(Y_2 X_3 Y_3 - X_2 Y_3^2) - (X_2 Y_2 Y_3 - Y_2^2 X_3)(2Y_2 X_3^2 - 2X_2 X_3 Y_3)}, \\
T_5 &= \frac{(4C_{12} Y_3 - C_{23} Y_2)(Y_2 X_3 Y_3 - X_2 Y_3^2) - (4C_{13} Y_2 - C_{23} Y_3)(X_2 Y_2 Y_3 - Y_2^2 X_3)}{(2X_2^2 Y_3 - 2X_2 Y_2 X_3)(Y_2 X_3 Y_3 - X_2 Y_3^2) - (X_2 Y_2 Y_3 - Y_2^2 X_3)(2Y_2 X_3^2 - 2X_2 X_3 Y_3)},
\end{aligned} \tag{7.3}$$

where

$$
\begin{aligned}
C_2 &= 4I_{12} - 3I_1 - I_2, \\
C_{12} &= I_1 + I_2 - 2I_{12}, \\
C_3 &= 4I_{13} - 3I_1 - I_3, \\
C_{13} &= I_1 + I_3 - 2I_{13}, \\
C_{23} &= 4I_1 - 4I_{12} - 4I_{13} + 4I_{23}.
\end{aligned} \tag{7.4}
$$

Having computed the $T_0, \ldots, T_5$ parameters, we should run a quadratic interpolation scheme which has already been discussed in Section 2.2.3.

7.1 Error control

The method proposed above approximates a non-linear function by a quadratic formula. If the triangles are too big and the radiance changes quickly due to a highlight, then this approximation can still be inaccurate. In order to avoid this problem, the accuracy of the approximation is estimated, and if it exceeds a certain threshold, then the triangle is adaptively subdivided into 4 triangles by halving the edges (Figure 7.1).

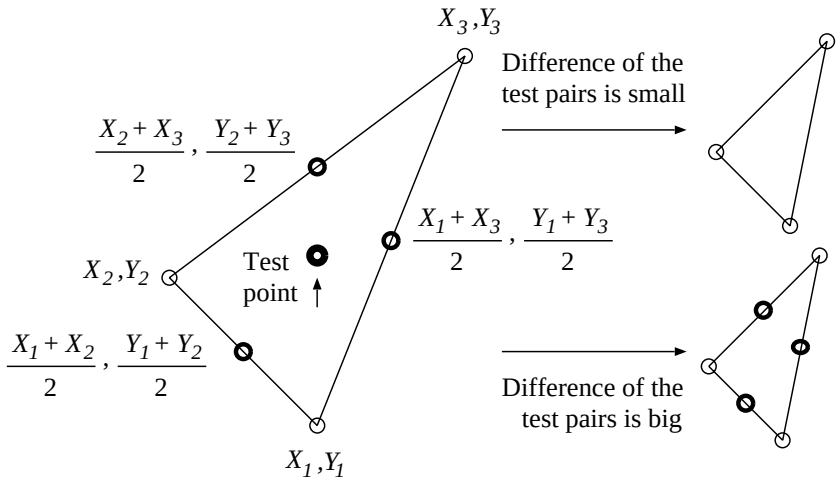


Figure 7.1: Highlight test and adaptive subdivision

Recall that the knot points of the interpolation are the vertices and the middle points of the edges. Thus a reasonable point where the error can be measured is the center of the triangle. This leads to the following highlight test algorithm. Having computed the $T_0, \ldots, T_5$ parameters, the function value is estimated at the center of the triangle using Equation 7.1, and the result is compared with the real value of $I(X, Y)$. In case of big difference, adaptive subdivision takes place. Note that the overhead of one more function evaluation is affordable and during subdivision the already computed function values can be reused.

7.2 Hardware implementation of quadratic interpolation

The quadratic function (Equation 7.1) in its original form is hard to be implemented directly in hardware. Fortunately, the main advantage of a quadratic function is that it can be traced back to simple additions using the incremental concept. It means that the color of a pixel is obtained using the color of the previous pixel and an increment is linear if the original function is quadratic. Furthermore, the increment itself is obtained as the increment value of the previous pixel by a simple addition. This requires altogether two fixed-point additions per pixel (Section 2.2.3).

The hardware implementation of the proposed quadratic interpolation is straightforward. We use registers with a feedback through an adder to realize the increment operation. Two such networks are used, one for the color and the other one for the increment of the color. An additional counter (we call it X counter) is responsible for providing the pixels addresses inside the scan-line. In fact, the same trick can be used on the edges of the triangle, which leads to a hardware that automatically renders a complete half triangle, not only a triangle scan-line.

The block scheme of the hardware implementation of quadratic shading is shown in Figure 2.7, the linear interpolator of this hardware is given by the following behavioral model:

```
ARCHITECTURE Behavior OF Interpolator IS

  SIGNAL Adder_Out, Reg_Out: bit_vector_32;

  Register_Process:

    PROCESS ( Clk,Load_Step )
      IF ( Load_Step = '1' ) THEN
        Reg_Out <= InitVal AFTER DelayReg;
      ELSE
        IF ( Clk'EVENT AND Clk = '1' ) THEN
          Reg_Out <= Adder_Out AFTER DelayReg;
        END IF;
      END IF;
    END PROCESS;

  Adder_Process:

    PROCESS ( Reg_Out,Stepval )
      VARIABLE TmpVal: bit_vector_32 := (others => '0');

      int_to_bit_vector(bit_vector_to_int(Reg_Out)
                        + bit_vector_to_int(StepVal), TmpVal);
      Adder_Out <= TmpVal AFTER DelayAdder;
    END PROCESS;

  OutVal <= Reg_Out;
END Behavior;
```

7.3 Simulation results

The new interpolation scheme that uses appropriately selected quadratic functions can be implemented in hardware and can be initialized without the computational burden of the Taylor's series approach. Unlike previous techniques the new method can simultaneously handle arbitrary number of light sources and arbitrary BRDF models. The proposed algorithm has been implemented in C++ and tested as a software. In Figures 7.3, 7.4 and 7.5 spheres are tessellated on different levels are compared. Gouraud shading evaluates the shading equation for every vertex, quadratic shading for every vertex and edge centers and Phong shading for each pixel. The difference of the algorithms is significant when the tessellation is not very high. The measured times of drawing highly tessellated shaded sphere (690 triangles) is as follows: Gouraud shading $230\ msec$, quadratic shading $250\ msec$, and Phong shading $450\ msec$. Note that Gouraud shading performs poorly on coarsely tessellated spheres, but the visual quality of quadratic and Phond shadings are similar. On the other hand, concerning the speed and the suitability for hardware implementation, quadratic shading is close to Gouraud shading. Finally, the hardware was specified in VHDL, which is a popular hardware description language, and simulated in Model-Technology environment. The delay times are according to Xilinx Synthesis Technology real FPGA device. We can follow the operation of the hardware from the timing sequence (Figure 7.2). The hardware can generate one pixel per one clock cycle. The length of clock cycle — which is also the pixel drawing time — depends on FPGA devices and on the screen memory access time. For the mentioned device it can be less than $50\ nsec$. While the hardware draws the actual triangle, the software can compute the initial values for the next triangle, so initialization and triangle drawing are executed parallely. The number of triangles per second depends only on initialization time (for small triangles, initialization time can be grater than drawing time).

In the Appendix at the end of this thesis, we map the RGB interpolators of the quadratic interpolation for the lower half triangle on Xilinx Spartan2e device, FPGA Family Members, xc2s100e type. Upon the results of this mapping we can conclude that this hardware is really suitable for real time rendering.

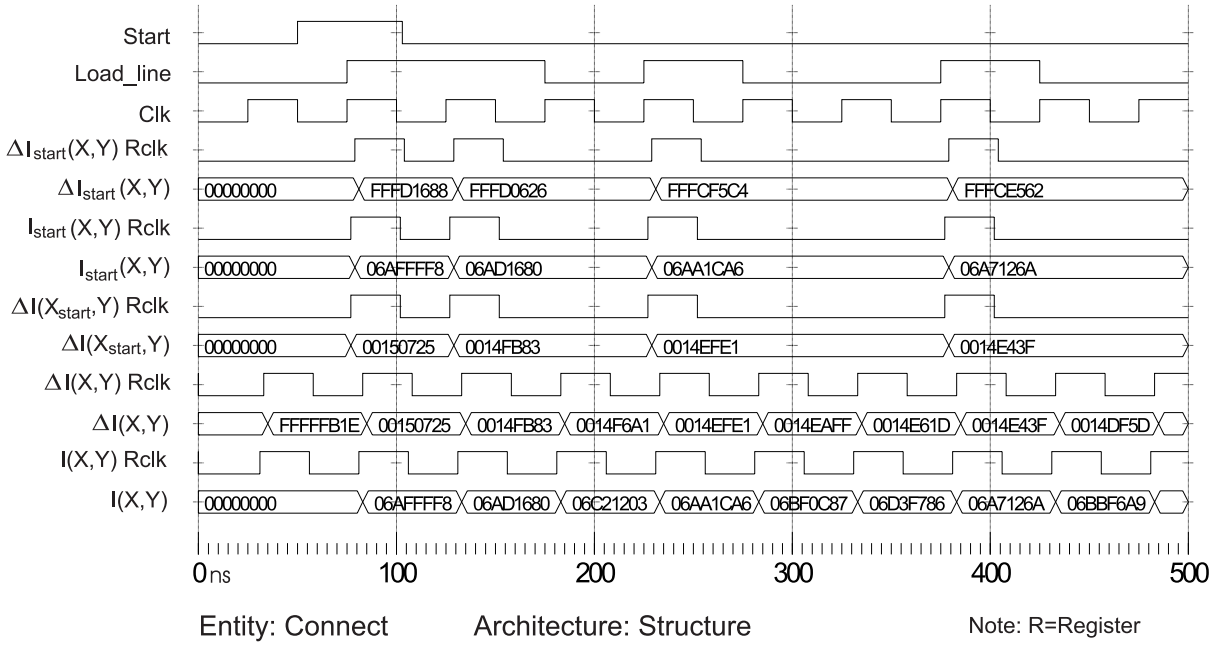


Figure 7.2: Timing diagram of the hardware implementation of quadratic shading

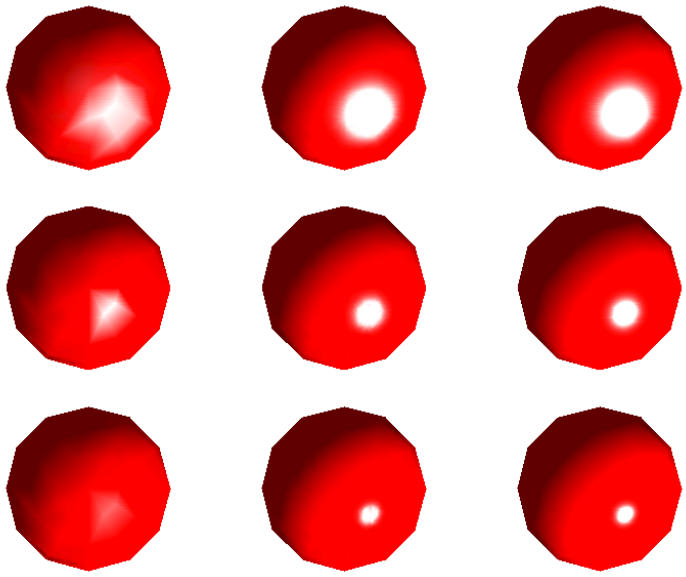


Figure 7.3: Rendering of coarsely tessellated spheres (168 triangles) of specular exponents $n = 5$ (top), $n = 20$ (middle) and $n = 50$ (bottom) with Gouraud shading (left), quadratic shading (middle) and Phong shading (right)

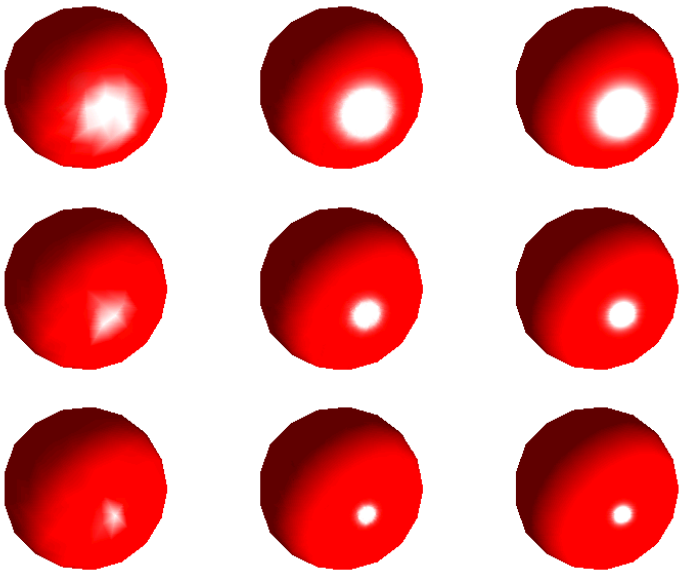


Figure 7.4: Rendering of normal tessellated spheres (374 triangles) of specular exponents $n = 5$ (top), $n = 20$ (middle) and $n = 50$ (bottom) with Gouraud shading (left), quadratic shading (middle) and Phong shading (right)

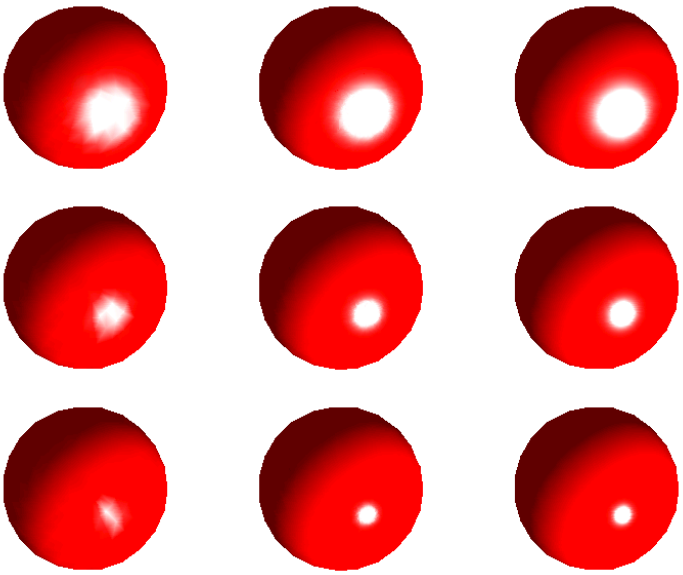


Figure 7.5: Rendering of highly tessellated spheres (690 triangles) of specular exponents $n = 5$ (top), $n = 20$ (middle) and $n = 50$ (bottom) with Gouraud shading (left), quadratic shading (middle) and Phong shading (right)

Chapter 8

Texture mapping

Since linear interpolation may degrade the image quality, while the non-linear interpolation involves complex operations on the pixel level, which makes its hardware realization impossible. Texture mapping is another famous non-linear operation in incremental rendering. Our new approach called **quadratic interpolation** introduced in Chapter 7, which is in between these two strategies is also implemented here to handle texture mapping, we call it **quadratic texturing** (Figure 10.3).

Texture mapping is a technique for adding realism to computer-generated scene. In its basic form, texture mapping lays an image (the texture, i.e. surface details) onto an object in a scene [KSKS96]. Texture mapping requires the determination of the surface parameters each time the rendering equation is evaluated for a surface point. When mapping an image onto an object, the color of the object at each pixel is modified by a corresponding color from the image. In general, obtaining this color from the image conceptually requires several steps. The varying optical parameters required by the rendering equation, on the other hand, are usually defined and stored in a separate coordinate system, called **texture space**. The texture information can be represented by some data stored in an array or by a function that returns the value needed for the points of the texture space. The image first should be filtered to remove high frequency components causing an aliasing by one of the filtering techniques such as **Mip-Map** or **Summed Area Table** [EWWL98]. In order to have a correspondence between texture space data and the surface points, a transformation is associated with the texture, which maps texture space onto the surface defined in its local coordinate system. This transformation is called **parameterization**. Modeling transformation maps these local coordinate system points to the world coordinate system where the shading is calculated. Incremental shading models, however, need another transformation from world coordinates to screen space where the hidden surface elimination and simplified color computation take place. This latter mapping is regarded as **projection** in texture mapping (Figure 8.1).

Since the parameters of the rendering equation are required in screen space, but they are available only in texture space, the mapping between the two spaces must be evaluated for each pixel.

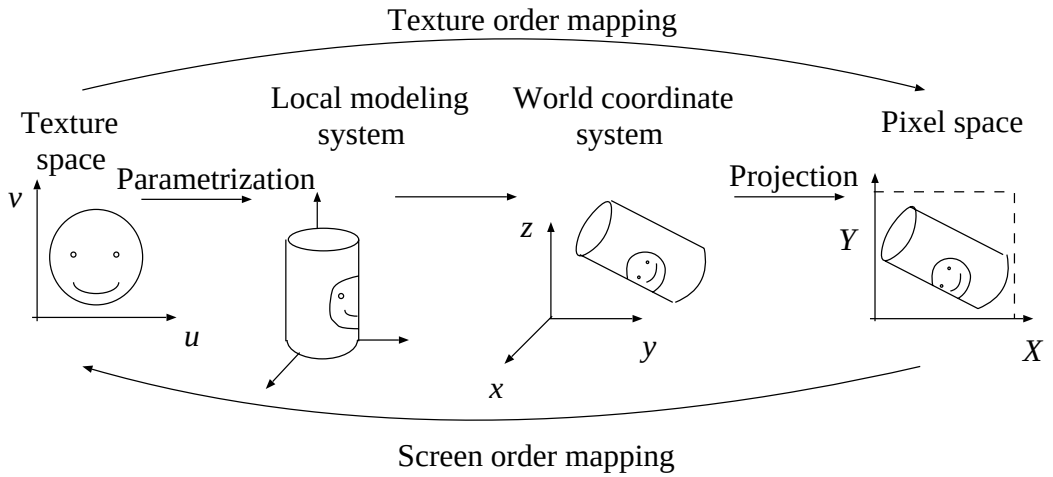


Figure 8.1: Survey of texture mapping

Generally, two major implementations are possible:

1. **Texture order** or **direct mapping** which scans the data in texture space and maps from texture space to screen space.

2. **Screen order** or **inverse mapping** which scans the pixels in screen space and uses the mapping from screen space to texture space.

Screen order is more popular, because it is appropriate for image precision hidden surface removal algorithms. In our approach screen order scheme is implemented.

8.1 Quadratic texturing

Since texture mapping finds a point in texture space for each pixel in screen space. Mapping a triangle from $2D$ screen space by the inverse camera transformation to the $2D$ texture space requires a homogeneous linear transformation, which becomes non-linear for Cartesian coordinates. For triangles, the screen coordinates and the texture coordinates are connected by a homogeneous linear transformation [SK95], thus for a pixel X, Y the corresponding texel coordinates u, v can be obtained as:

$$u = \frac{a_u X + b_u Y + c_u}{d_u X + e_u Y + f_u}, \quad v = \frac{a_v X + b_v Y + c_v}{d_v X + e_v Y + f_v}, \tag{8.1}$$

where $a_u, \ldots, f_v$ depend on the positions of the triangle in the texture and screen spaces. Note that this operation also contains divisions that are quite intensive computationally and makes the mapping non-linear. Implementing division in hardware is difficult and can be the bottleneck of texture mapping [Ack96]. Approximating this function by a linear transformation, on the other hand, makes the perspective distortion incorrect [GPC82, SSW86] [DG96] (Figure 8.2).

We propose the application of the **quadratic interpolation** introduced in Chapter 7 also for this problem. Since the quadratic function (Equation 7.1) has six degrees of freedom, its parameters $T_5, \ldots, T_0$ should also be solved for the u and v texture coordinates at six representative points on an equivalent triangle in texture space. Again, we can select the vertices and the middle points of the edges as representatives and check the accuracy of the approximation at the middle of the triangle.

8.2 Simulation results

In order to compare the quality of linear and quadratic approximation of texture transformation a tiger and a turtle texture were assigned to a rectangle divided into two triangles (Figure 8.2). Note that linear transformation distorts the textures in an unacceptable way, while quadratic approximation handles the perspective shrinking properly.

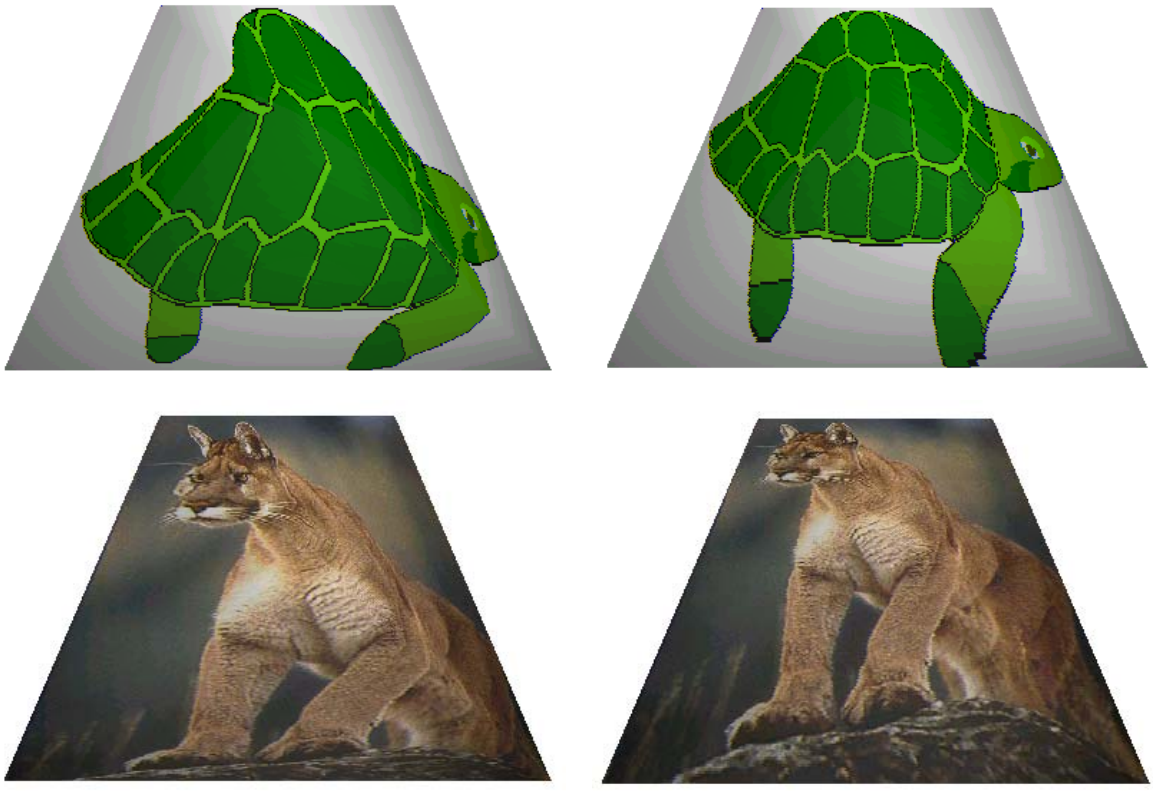


Figure 8.2: Texture mapping with linear (left), quadratic (right) texture transformation

In general we can conclude that, incorrect highlights and texture distortion all disappears, so quadratic texturing can really solve the drawbacks of linear texture mapping.

Chapter 9

Shaded surface rendering using global illumination

Global illumination algorithms aim at solving the **rendering equation** (Equation 1.1). This equation expresses the radiance of a surface as a sum of its own emission I^e and the reflection of the radiances of those points that are visible from here $(\mathcal{T}I)$. To find the possible visible points, all incoming directions should be considered and the contribution of these directions should be summed, which results in an integral operator (called light transport operator, Equation 1.2).

9.1 The global illumination problem

Since the rendering equation contains the unknown radiance function both inside and outside the integral, in order to express the solution, this coupling should be resolved [KSKA01]. The possible solution techniques fall into one of the following three categories: inversion, expansion or iteration. Here only iteration is discussed in details since it is the most appropriate for fast rendering. Iteration realizes that the solution of the rendering equation is the fixed point of the following iteration scheme:

$$I^{(n)} = I^e + \mathcal{T}I^{(n-1)}.$$

This scheme requires the temporary representation of the radiance function I until it is substituted into the iteration formula. In order to represent this function by finite number of variables, finite-element methods can be applied. Since the radiance function has four variates and changes quickly, the finite element approximation requires very many basis functions, which makes this approach rather time and memory consuming. Fortunately this problem can be effectively attacked by the concept of **stochastic iteration** [SK00] [MSKSA01].

The basic idea of stochastic iteration is that instead of approximating operator $\mathcal{T}$ in a deterministic way, a much simpler random operator is used during the iteration which "behaves" as the real operator just in the "average" case. Suppose that we have a random linear operator $\mathcal{T}^*$ so that:

$$E[\mathcal{T}^*I] = \mathcal{T}I, \tag{9.1}$$

for any integrable function I.

During stochastic iteration a random sequence of operators $\mathcal{T}_1^*, \mathcal{T}_2^*, \ldots \mathcal{T}_i^* \ldots$ is generated, which are instantiations of $\mathcal{T}^*$, and this sequence is applied to the radiance function:

$$I^{(n)} = I^e + \mathcal{T}_n^* I^{(n-1)}. \tag{9.2}$$

Note that this scheme does not converge but the radiance estimates will fluctuate around the real solution. The real solution can be obtained as the average of these estimates:

$$I = \lim_{n\to\infty} \frac{1}{M} \cdot \sum_{n=1}^{M} I^{(n)}. \tag{9.3}$$

9.2 Ray-bundle based transfer

So far we have given a complete freedom to the definition of the transport operator. Obviously those operators should be preferred which can be evaluated quickly and for which hardware support is feasible. Since this is true for the visibility problem assuming fixed eye position and parallel projection, we use a random transport operator that transfers the radiance of all surface points of the scene in a single random direction.

In order to store the temporary radiance during the iteration, finite element techniques are used, that tessellate the surfaces into elementary planar patches and assume that a patch has uniform radiance in a given direction (note that this does not mean that the patch has the same radiance in every direction, thus the non-diffuse case can also be handled). According to the concept of finite-elements, the radiance, the emission and the BRDF of patch i are assumed to be independent of the actual point inside the patch, and are denoted by $I_i(\vec{V})$, $I_i^e(\vec{V})$ and $\tilde{f}_i(\vec{L}, \vec{V})$, respectively. It means that the radiance function is approximated in the following form:

$$I(\vec{x}, \vec{V}) \approx \sum_i I_i(\vec{V}) \cdot b_i(\vec{x}), \tag{9.4}$$

where $b_i(\vec{x})$ is 1 on patch i and 0 otherwise. The $I_i(\vec{V})$ patch radiance can be considered as the average of the radiances of the points on the patch:

$$I_i(\vec{V}) = \frac{1}{A_i} \cdot \int_{A_i} I(\vec{x}, \vec{V})\, d\vec{x}. \tag{9.5}$$

Applying the random transport operation for the radiance represented in this form, we obtain:

$$I^{(n)}(\vec{x}, \vec{V}) = I^e(\vec{x}, \vec{V}) + \mathcal{T}_n^* I^{(n-1)}(\vec{y}, \vec{L}). \tag{9.6}$$

From this function, the patch radiances are generated as follows:

$$I_i^{(n)}(\vec{V}) = I_i^e(\vec{V}) + \frac{1}{A_i} \cdot \int_{A_i} \mathcal{T}_n^* I^{(n-1)}(\vec{y}, \vec{L})\, d\vec{x}. \tag{9.7}$$

Taking into account that in a given direction other patches are seen that have constant radiance, this integral can also be presented in closed form:

$$I_i^{(n)}(\vec{V}) = I_i^e(\vec{V}) + \sum_{j=1}^{n} \tilde{f}_i(\vec{L}, \vec{V}) \cdot A(i, j, \vec{L}) \cdot I_j^{(n-1)}(\vec{L}), \tag{9.8}$$

where $A(i, j, \vec{L})$ expresses the projected area of patch j that is visible from patch i in direction $\vec{L}$. In the unoccluded case this is the intersection of the projections of patch i and patch j onto a plane perpendicular to $\vec{L}$. If occlusion occurs, the projected areas of other patches that are in between patch i and patch j should be subtracted as shown in Figure 9.1.

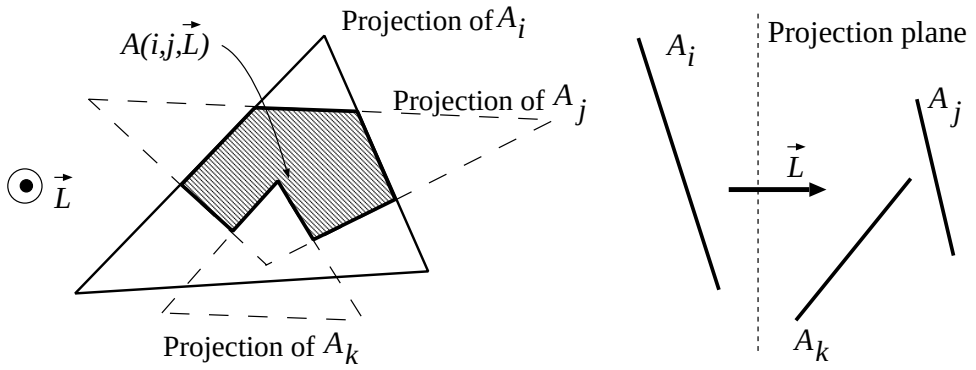


Figure 9.1: Interpretation of $A(i, j, \vec{L})$

The resulting algorithm is quite simple. In a step of the stochastic iteration an image estimate is computed by reflecting the previously computed radiance estimate towards the eye, and a new direction is found and this direction together with the previous direction are used to evaluate the random transport operator. The complete algorithm — which requires just one variable for each patch i, the previous radiance $I[i]$ — is summarized in the following algorithm:

Generate the first random global direction $\vec{V}_1$;
for each patch i **do**
  $I[i] = I_i^e(\vec{V}_1)$;
endfor
for $m = 1$ **to** M **do** *// iteration cycles*
  Calculate the image estimate reflecting $I[1], I[2], \ldots I[n]$ from $\vec{V}_m$ towards the eye;
  Average the estimate with the *Image*;
  Generate random global direction $\vec{V}_{m+1}$;
  for each patch i **do**
    $I^{\text{new}}[i] = I_i^e(\vec{V}_{m+1}) + 4\pi \cdot \sum_{j=1}^{n} \tilde{f}_i(\vec{V}_m, \vec{V}_{m+1}) \cdot A(i, j, \vec{V}_m)/A_i \cdot I[j]$;
  endfor
endfor

9.3 Calculation of the radiance transport in a single direction

To evaluate the transport operator, we need to know which patches are visible from a given patch, and then we have to weight the radiances of visible patches by the ratio of their visible sizes and the size of the given patch.

This requires the solution of a global visibility problem, where the eye position visits all surface points but the viewing direction is fixed to the selected random direction. This fixed direction is called the *transillumination direction* (Figure 9.2).

At a given point of all global visibility algorithms the objects visible from the points of a patch must be known (Figure 9.3). This information is stored in a data structure called the **visibility map**. The visibility map can also be regarded as an image on the plane perpendicular to the transillumination direction. This plane is called the *transillumination plane*.

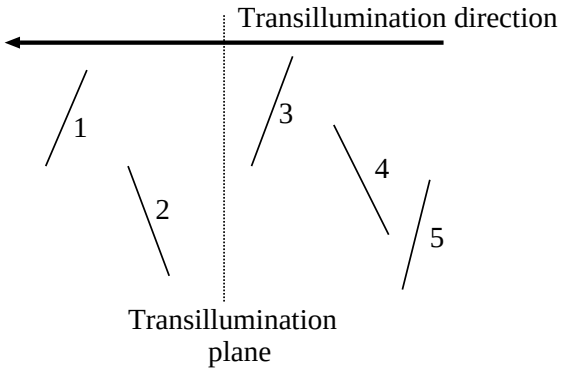


Figure 9.2: Global visibility algorithms

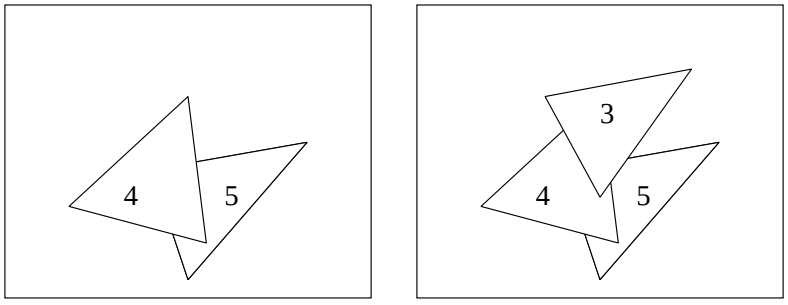


Image seen from patch 3 Image seen from patch 2

Figure 9.3: Scene as seen from two subsequent patches

Discrete algorithms, which decompose the transillumination plane to small pixels of size ΔP, can solve the problem much faster. For discrete algorithms, the visibility map is simply a rasterized image where each pixel can store either the index of the visible patch or the radiance of the visible point.

Discrete algorithms determine the image of the visible patches through a discretized window assuming the eye to be on patch i, the window to be on the transillumination plane and the color of patch j to be j if the patch is facing to patch i and to be 0 otherwise.

We use an extension of the z-buffer algorithm to identify the patches that see each other through the pixels of the transillumination plane. The main difference from the original z-buffer algorithm is that now a pixel should be capable to store a list of patch indices and z-values, not just the values of the closest patch (Figure 9.4). The lists are sorted according to the z-values. The patches are rendered one after the other into the buffer using a modified z-buffer algorithm which keeps all visible points not just the nearest one. Traversing the generated lists the pairs of mutually visible points can be obtained. For each pair of points, the radiance transfer is computed and the transferred radiance is multiplied by the BRDF, resulting in the reflected radiance.

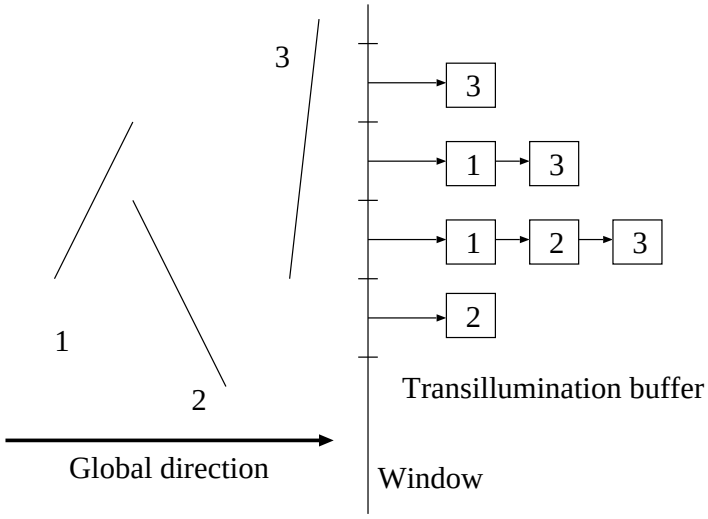


Figure 9.4: Organization of the transillumination buffer

9.4 Hardware implementation of the proposed radiance transfer algorithm

The hardware of the radiance transfer algorithm is composed of two dependent stages, stage one (input interface and sorting the input data in memory) and stage two (output interface of the sorted data in memory). The input data is produced by the software and it is composed of arrays of patches with their related Z values according to the screen position that these patches can be

seen. When the software initiates the data for the hardware, stage one fetches it and halts stage two until it finishes its operation of sorting the input data in memory according to the smallest Z value of the patches that can be seen from the specific screen position. When stage one finishes its operation generates a signal to stage two to output the sorted data to the software.

The block scheme of this hardware is shown in Figure 9.5. This hardware runs according to the data illustrated in Figure 9.4. The time sequence of stage one is shown in Figure 9.6 and its VHDL description code is given by the following behavioral model:

```
ARCHITECTURE Behavior OF Control1 IS
SIGNAL   FP: ADDRESS := ("00........01");  -- Pointing to SCREEN_ADDRESS 1
SIGNAL   PR,PPR: ADDRESS;                  -- Pointer field is 24 bits wide
SIGNAL   ZR: DATA;                         -- DATA field is 24 bits wide
SIGNAL   Zero: DATA := (others => '0');
BEGIN
  PROCESS
    PROCEDURE Mem_Write (Addr: IN ADDRESS; d_w: IN DATA; Sel: in Sel_Type);
    PROCEDURE Mem_Read (Addr: IN ADDRESS; d_r: OUT DATA; Sel: in Sel_Type);
  BEGIN
    Mem_Write(FP,Z,Sel_Z); Mem_Write(FP,I,Sel_I); Mem_Write(FP,Zero,Sel_P);
    Mem_Read(SCREEN_to_ADDRESS(S), PR, Sel_S);
                          -- SCREEN_ADDRESS 20 bits, extended to 24 bits
    IF ( IsNull(PR) ) THEN
      Mem_Write(SCREEN_to_ADDRESS(S), FP, Sel_S);
    ELSE                                   -- Sel_Z selects Z_value field
      PPR <= PR;                           -- Sel_S selects Screen field
      WHILE ( TRUE ) LOOP                  -- Sel_P selects pointer field
        Mem_Read(PR,ZR,Sel_Z);             -- Read z-value addressed by PR
        IF ( IsLessOrEq(Z,ZR) ) THEN       -- Sel_I selects patches field
          EXIT;
        END IF;
        PPR <= PR;  Mem_Read(PR,PR,Sel_P);
        IF ( IsNull(PR) ) THEN             -- If true, then end of storing
          EXIT;
        END IF;
      END LOOP;
    END IF;
    -- PPR is a pointer holding the previous value of a specific pointer
    IF ( IsNull(PR) ) THEN
      Mem_Write(PPR,FP,Sel_P);             -- Store the first address
    ELSEIF ( PR = PPR ) THEN
          Mem_Write(SCREEN_to_ADDRESS(S),FP,Sel_S);
          Mem_Write(FP,PR,Sel_P);
        ELSE
          Mem_Write(PPR,FP,Sel_P);  Mem_Write(FP,PR,Sel_P);
        END IF;
    END IF;

    FP <= Increment(FP);                   -- Free SCREEN pointer
  END PROCESS;
END Behavior;
```

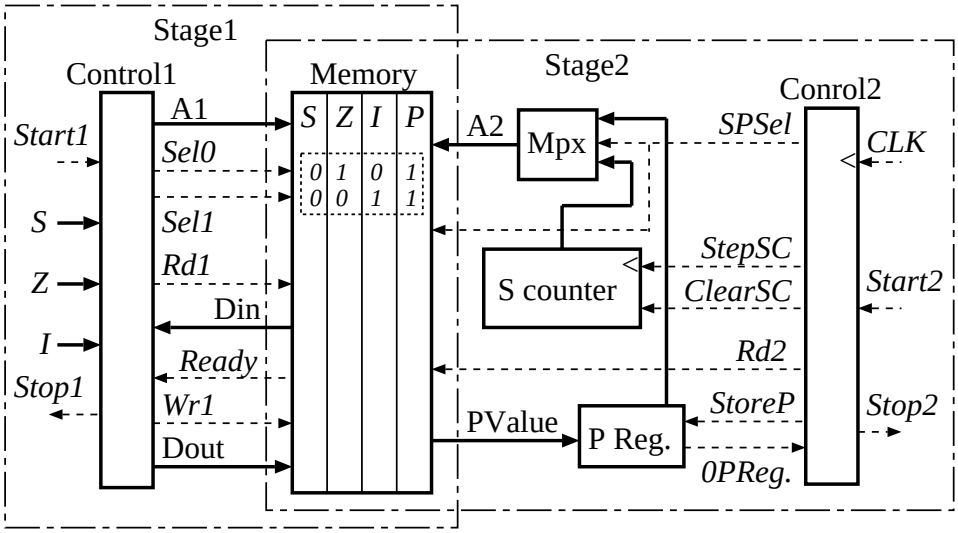


Figure 9.5: Hardware implementation of radiance transfer algorithm

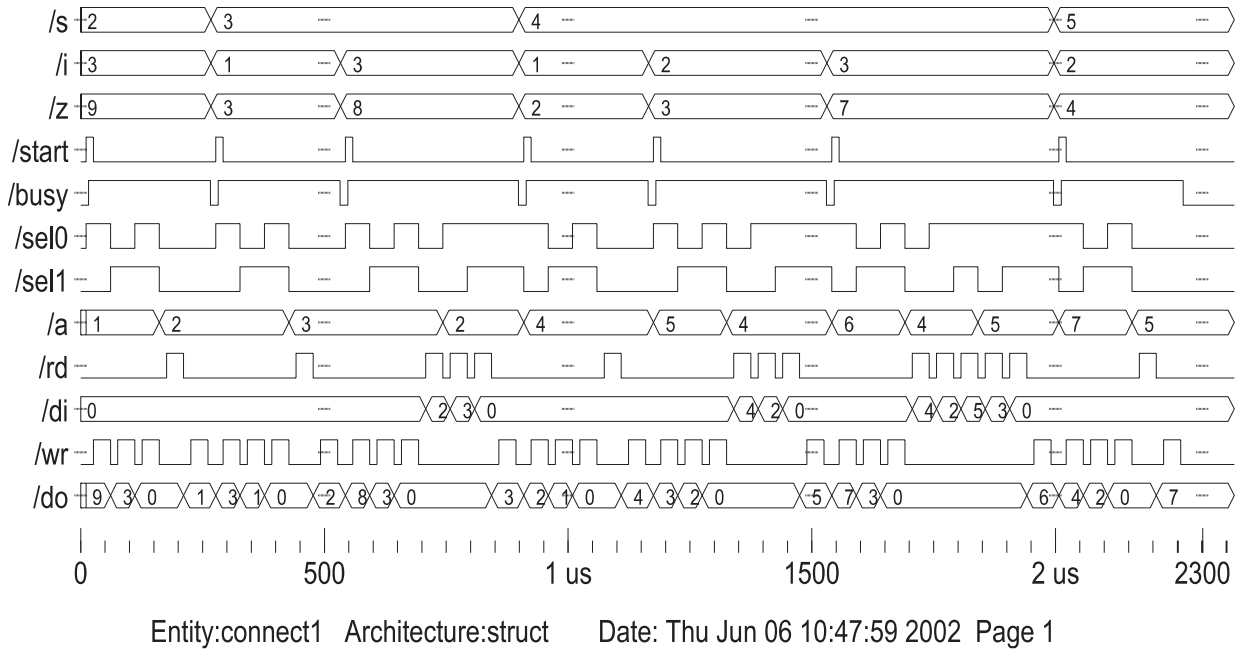


Figure 9.6: Stage one time sequence of the hardware implementation of radiance transfer algorithm

Chapter 10

Conclusions and summary of new results

In this thesis we proposed new image synthesis algorithms. These algorithms have solved the drawbacks of linear interpolations and are comparable in image quality with the already known, sophisticated techniques but they have allowed simple hardware implementation. These algorithms cover the incremental cone-filtering lines, an alternative Phong shading using spherical interpolation, quadratic interpolation in Phong shading and texture mapping and ray-bundle based global illumination. The main framework and the conclusions of these rendering algorithms are discussed in the following sections. The suggestions of the future research work is declared in Section 10.6.

10.1 General framework to compute simple functions on 2D triangles

In incremental rendering the geometric primitives are tessellated to triangles and line segments and are transformed and projected onto the screen coordinates. The projected triangles are filled and the line segments are rasterized, i.e. those pixels are identified that approximate them, and pixel colors are computed simultaneously. Realizing that the pixel color can be a non-linear function of the pixel coordinates, a general framework and a supporting hardware were developed to realize this filling operation together with the non-linear function computation. The computation is based on the so-called incremental concept, which means that a pixel value is obtained as the increment value of the previous pixel. Since if the function is an n degree polynomial, the increment is an $n-1$ degree polynomial, thus the computation of an n degree polynomial can be traced back to an addition and to the evaluation of an $n-1$ degree polynomial. Using this idea recursively n parallel and fixed-point additions can compute the new value. The required accuracy was also investigated and we concluded that the required number of fractional bits is proportional to the degree of the polynomial.

10.2 Hardware implementation of incremental cone-filtering lines

In order to reduce the jaggies of digital lines, the Gupta-Sproull algorithm applies conic filtering that computes the conic volume above the intersection of the pixel and the base of the cone. The resulting lines are of good quality, but the algorithm is computationally intensive. We propose the transformation of this algorithm using the incremental concept that allows its hardware realization. In the new algorithm the distance between the line and the pixel center is evaluated incrementally and the volume of the conic segment is obtained through a look-up table.

10.3 Hardware implementation of Phong shading using spherical interpolation

Classical Phong shading interpolates the normal, view and light vectors inside the triangle and evaluates the rendering equation for each pixel in the triangle. The vector interpolation involves vector normalization, while the rendering equation requires dot product computation and exponentiation of scalar values. These operations are far too complex to be implemented in hardware. To attack the problem of vector normalization, we proposed the spherical interpolation of vectors [ASKH00], i.e. when the interpolation is done on the great arc of the surface of a unit sphere rather than on the line. Using quaternion algebra, the dot product of two spherically interpolated vectors has been expressed as a simple cosine. Finally, the exponentiation of the cosine has been approximated by the $\cos^2(s \cdot X + b)$ function, where parameters s and b can be determined from the original material parameters. We demonstrated that the approximations needed by the algorithm simplification do not degrade the visual quality of the rendered images.

10.4 Quadratic interpolation in rendering

Since rendering involves strongly non-linear operations, classical approaches using linear interpolation are inadequate for rendering high-quality images. To eliminate those artifacts we propose a quadratic scheme [ASKHF00] for a function I:

$$I(X,Y) = T_5X^2 + T_4XY + T_3Y^2 + T_2X + T_1Y + T_0.$$

In order to compute the unknown $T_0, \ldots, T_5$ parameters, the function to be interpolated should be evaluated at six representative points. In order to take into account the ease of computation and the elimination of visual inaccuracy called Mach banding, we propose the triangle vertices and the half points between the vertices for such representative points. We concluded that this selection results in simple formulae and does not necessitate the solution of linear equations.

10.4.1 Adaptive error control in quadratic interpolation

In order to control the error of the quadratic interpolation, the difference between the original function value and its approximation is compared and if it exceeds a certain threshold the triangle is subdivided into 4 equivalent triangles.

10.4.2 Application of quadratic rendering for Phong shading and texture mapping

The idea of quadratic interpolation [ASKH+01a] was used to develop an alternative shading algorithm for Phong shading. The solution of the rendering equation is evaluated at six representative points and the color interpolation is done by the quadratic scheme in image space. We demonstrated that the quadratic interpolation is comparable to Phong shading in image quality.

Quadratic interpolation has also been used to solve the non-linearity problem of texture mapping [ASKH+01a] [ASKH+01b]. The texture coordinates are evaluated at six representative points and the other pixels are approximated according to the quadratic formula. We demonstrated that the quadratic interpolation can avoid texture distortions.

Summarizing, the main problem of the original shading pipeline used by Phong shading (Figure 10.1) is that it involves complex operations such as lighting calculations and texture transformations, which makes its direct hardware realization impossible.

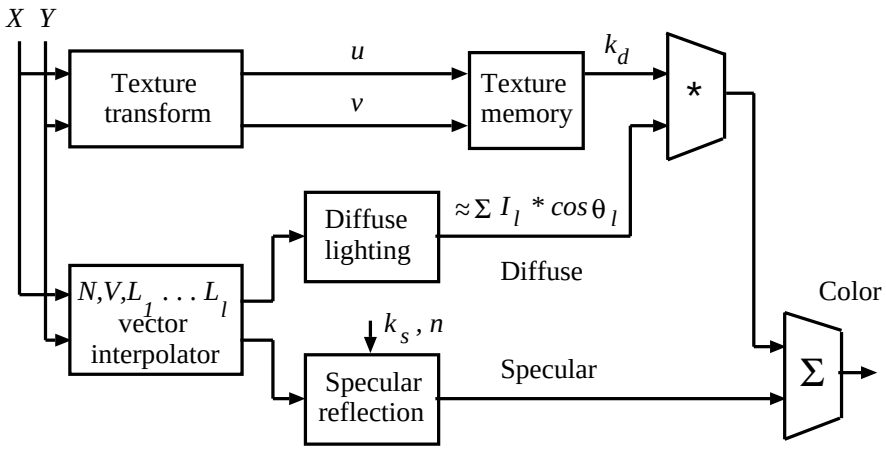


Figure 10.1: Conventional rendering with Phong shading and texture mapping without interpolation

Traditionally, this problem is attacked by linear interpolation as shown by Figure 10.2, but the linear interpolation of the strongly non-linear functions degrades the image quality [SAFL99].

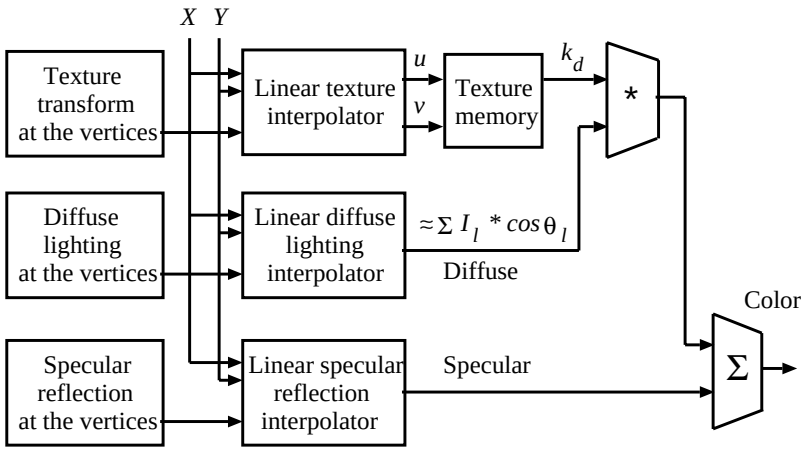


Figure 10.2: Linear interpolation, i.e. Gouraud shading and linear texture mapping

In this thesis we proposed a new interpolation scheme (Figure 10.3) that uses appropriately selected quadratic functions which can be implemented in hardware and can be initialized without the computational burden of the Taylor's series approach. Unlike previous techniques the new method can simultaneously handle arbitrary number of light sources and arbitrary BRDF models.

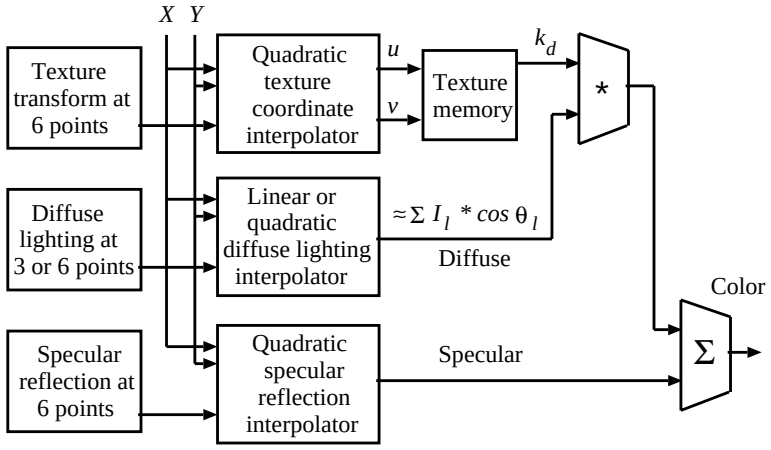


Figure 10.3: Quadratic rendering

Figures 10.4, 10.5 and 10.6 show specular objects of more complex scenes with normal tessellation levels rendered with Gouraud shading, classical Phong shading and with the proposed quadratic rendering. Looking at these images we can conclude that the quadratic rendering is visually superior to Gouraud shading and indistinguishable from classical Phong shading.

Figure 10.4: A shaded pawn with Gouraud (left), Phong (middle) and Quadratic (right)

Figure 10.5: A shaded and textured apple with Gouraud (left), Phong (middle) and Quadratic (right)

Figure 10.6: Coarsely tessellated, shaded, textured and specular tiger with Gouraud (left) Phong (middle) and quadratic (right)

10.5 Hardware implementation of global illumination

The ray-bundle based stochastic iteration algorithm can render complex scenes according to the global illumination principles. In this thesis we presented a global visibility algorithm and its hardware realization, which can support its operation.

10.6 Suggestions of future research

We realized that during the last two decades there were enormous developments in the field of computer graphics software and hardware. The main objective of computer graphics is to generate realistically looking images on a computer screen and as fast as possible. Taking into account the computational burden associated with this process, real-time image synthesis is possible only with either hardware support or with high-level parallelization. Recently, parallelization has become available on the low levels that are close to the hardware. The famous pixel shaders of the latest graphics cards delivered low level firmware programming features to the application developers. One possibility of the future research work could be the consideration of pixel shaders as an implementation framework for the proposed algorithms. On the other hand, there are many other computationally intensive methods of rendering, which can be attacked by the simplification methods proposed in this thesis, including for example, sophisticated texture filtering, and bump-, environment-, reflection- and displacement-mapping, volume visualization algorithms, etc. We believe that even these complex operations can be executed directly by the hardware in the future.

BIBLIOGRAPHY

[Abb95] A. M. Abbas. *Programmable Logic Controller based Intel 80C196KB Microprocessor, M.Sc. Project*. Faculty of Electrical Engineering and Informatics, Technical University of Budapest, Hungary, December, 1995.

[Abb98] A. M. Abbas. On 2D Line Scan Conversion. In *Conference on The Latest Results in Information Technology*, pages 50–57, Department of Control Engineering and Information Technology, Technical University of Budapest, Hungary, 1998.

[Abb01] A. M. Abbas. Photorealistic Images in Real-Time. *Journal of Industrial Researches*, Accepted for publication, Industrial Research Center, Tripoli Libya, June 2001.

[Ack96] H. Ackermann. Single chip hardware support for rasterization and texture mapping. *Computers & Graphics*, 20(4):503–514, 1996.

[Ash90] Peter J. Ashenden. *The VHDL Cookbook*. Department of Computer Science, University of Adelaide, South Australia, 1990.

[ASKH00] A. M. Abbas, L. Szirmay-Kalos, and T. Horváth. Hardware Implementation of Phong Shading using Spherical Interpolation. *Periodica Polytechnica*, 44(3-4):283–301, 2000.

[ASKH+01a] A. M. Abbas, L. Szirmay-Kalos, T. Horváth, T. Fóris, and Szijátó G. Quadratic Interpolation in Hardware Rendering. In *Spring Conference on Computer Graphics*, pages 239–248, Budmerice, Slovakia, 25–28 April 2001.

[ASKH+01b] A. M. Abbas, L. Szirmay-Kalos, T. Horváth, T. Fóris, and G. Szijártó. Quadratic Interpolation in Hardware Phong Shading and Texture Mapping. In *IEEE Computer Society Press in the Post-Proceedings of The 17th. Spring Conference on Computer Graphics*, pages 181–188, Budmerice, Slovakia, 25–28 April 2001.

[ASKHF00] A. M. Abbas, L. Szirmay-Kalos, T. Horváth, and T. Fóris. Quadratic Shading and its Hardware Interpolation. *Machine GRAPHICS & VISION*, 9(4):825–839, 2000.

[AVJ01] P. Arató, T. Visegrády, and I. Jankovits. *High Level Synthesis of Pipelined Datapaths*. JOHN WILEY & SONS, LTD, IIT, Budapest University of Technology and Economics, Hungary, 2001.

[BB99] V. Boyer and J. J. Bourdin. Fast Lines: a Span by Span Method. *EUROGRAPHICS'99*, 18(3):C–377–C–384, 1999.

[BERW97] K. Bennebroek, I. Ernst, H. Rüsseler, and O. Witting. Design principles of hardware-based Phong shading and bump-mapping. *Computers & Graphics*, 21(2):143–149, 1997.

[Bli77] J. F. Blinn. Models of Light Reflections for Computer Synthetized Pictures. In *Computer Graphics (SIGGRAPH '77 Proceedings)*, pages 192–198, 1977.

[BW86] G. Bishop and D. M. Weimar. Fast Phong Shading. *Computer Graphics*, 20(4):103–106, 1986.

[Che97] Jim X. Chen. Multiple Segment Line Scan-conversion. *COMPUTER GRAPHICS forum*, 16(5):257–268, 1997.

[Cla90] U. Claussen. On Reducing the Phong Shading Method. *Computer & Graphics*, 14(1):73–81, 1990.

[Cro81] F. C. Crow. A Comparison of Antialiasing Techniques. *Computer Graphics and Applications*, 1(1):40–48, 1981.

[CT81] R. Cook and K. Torrance. A Reflectance Model for Computer Graphics. *Computer Graphics*, 15(3), 1981.

[CW99] Jim X. Chen and Xusheng Wang. Approximate Line Scan-Conversion and Antialiasing. *COMPUTER GRAPHICS forum*, 18(1):69–78, 1999.

[DG96] D. Demirer and R. Grimsdale. Approximation techniques for high performance texture mapping. *Computers and Graphics*, 20(4):483–490, 1996.

[Duf79] T. Duff. Smoothly Shaded Rendering of Polyhedral Objects on Raster Displays. In *Computer Graphics (SIGGRAPH '79 Proceedings)*, 1979.

[DWS+88] M. Deering, S. Winner, B. Schediwy, C. Duffy, and Hunt N. The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics. In *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 21–30, 1988.

[EWWL98] Jon P. Ewins, Marcus D. Waller, M. White, and Paul F. Lister. MIP-MAP Level Selection for Texture Mapping. *IEEE TRANSACTION ON VISULATION AND COMPUTER GRAPHICS*, 4(4):317–329, 1998.

[FvDFH90] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, Mass., 1990.

[Gar75] P. L. Gardner. Modification of Bresenham's Algorithm for Displays. *IBM Technical Disclosure Bulletin*, 18(5):1595–1596, 1975.

[GK92] R. L. Grimsdale and A. Kaufman. *Advanced in computer graphics hardware V*. Springer [etc.] 1992, Eurographics seminars, Berlin, 1992.

[Gou71] H. Gouraud. Computer Display of Curved Surfaces. *ACM Transactions on Computers*, C-20(6):623–629, 1971.

[GPC82] M. Gangnet, P. Perny, and P. Coueignoux. Perspective Mapping of Planar Textures. In *EUROGRAPHICS '82*, pages 57–71, 1982.

[GSS81] S. Gupta, R. Sproull, and I. Sutherland. Filtering Edges for Gray-Scale Displays. In *Computer Graphics (SIGGRAPH '81 Proceedings)*, pages 1–5, 1981.

[HBB01] A. Hast, T. Barrera, and E. Bengtsson. IMPROVED SHADING PERFORMANCE BY VECTOR NORMALIZATION. In *Winter School of Computer Graphics'2001*, volume I–II, pages 344–351, Plzen, Czech Republic, 14–18 February 2001.

[Inc94] Model Technology Incorporated. *VHDL Simulation for PCs Running Windows & Windows NT*. Model Technology Incorporated, Beaverton OR 97008-7159 USA, 1994.

[Kaj86] J. T. Kajiya. The Rendering Equation. In *Computer Graphics (SIGGRAPH '86 Proceedings)*, pages 143–150, 1986.

[Kau93] A. Kaufman. *Rendering, Visualization and Rasterization Hardware*. EUROGRAPHICS, Department of Computer Science, State University of NY at Stony Brook, USA, 1993.

[KB89] A. A. M. Kuijk and E. H. Blake. Faster Phong Shading via Angular Interpolation. *Computer Graphics Forum*, (8):315–324, 1989.

[KSKA01] L. Kovács, L. Szirmay-Kalos, and A. M. Abbas. Testing Global Illumination Methods with Analytically Computable Scenes. In *Winter School of Computer Graphics'01*, volume II, pages 419–426, Plzen, Czech Republic, 14–18 February 2001.

[KSKS96] G. KNITTEL, A. SCHILLING, A. KUGLER, and W. STRABER. HARDWARE FOR SUPERIOR TEXTURE PERFORMANCE. *Computer & Graphics*, 20(4):475–481, 1996.

[Kun93] Tosiyasu L. Kunii. *Computer graphics: Theory and applications*. Springer [etc.], InterGraphics, Tokyo, 1993.

[Kuz95] Yevgeny P. Kuzmin. Bresenham's Lin generation Algoritm with Built-in Clipping. *Computer Graphics Forum*, 14(5):275–280, 1995.

[Lui94] Yong-Kui Lui. An All-integer Algorithm for Drawing Anti-aliased Straight Lines. *COMPUTER GRAPHICS forum*, 13(4):219–221, 1994.

[Min41] M. Minnaert. The Reciprocity Principle in Lunar Photometry. *Astrophysical Journal*, 93:403–410, 1941.

[MSKSA01] R. Martinez, L. Szirmay-Kalos, M. Sbert, and A. M. Abbas. Parallel Implementation of Stochastic Iteration Algorithms. In *Winter School of Computer Graphics'01*, volume II, pages 344–351, Plzen, Czech Republic, 14–18 February 2001.

[Nar95] Chandrasekhar Narayanaswami. Efficient Parallel Gouraud Shading and Linear Interpolation over Triangles. *Computer Graphics Forum*, 14(1):17–24, 1995.

[Per91] Douglas L. Perry. *VHDL (Computer hardware description language)*. R. R. Donnelley & Sons Company, San Ramon, California, 1991.

[Pho75] B. T. Phong. Illumination for Computer Generated Images. *Communications of the ACM*, 18:311–317, 1975.

[RBX90] J. G. Rokne, W. Brian, and Wu. Xiaolin. Fast Line Scan-Conversion. *ACM Transactions on Graphics*, 9(4):376–388, 1990.

[RL98] Hassan K. Reghbati and Anson Y. C. Lee. *Computer graphics hardware*. IEEE Computer Society Press, USA, 1998.

[SAFL99] M. Segal, K. Akeley, C. Frazier, and J. Leech. The OpenGL graphics system: A specification. Technical Report (version 1.2.1), Silicon Graphics, Inc., 1999.

[SK95] L. Szirmay-Kalos. *Theory of Three Dimensional Computer Graphics*. Akadémia Kiadó, Budapest, 1995. http//www.iit.bme.hu/˜szirmay.

[SK99a] L. Szirmay-Kalos. *Monte-Carlo Methods in Global Illumination*. Institute of Computer Graphics, Vienna University of Technology, Vienna, 1999.

[SK99b] L. Szirmay-Kalos. Stochastic iteration for non-diffuse global illumination. *Computer Graphics Forum (Eurographics'99)*, 18(3):233–244, 1999.

[SK00] L. Szirmay-Kalos. *Photorealistic Image Synthesis with Ray-Bundles*. Hungarian Academy of Sciences, D. Sc. Dissertation, Budapest, Hungary, 2000.

[SKM94] L. Szirmay-Kalos and G. Márton. On Hardware Implementation of Scan-conversion Algorithms. In *8th. Symp. on Microcomputer Appl.*, Budapest, Hungary, 1994.

[SKP98] L. Szirmay-Kalos and W. Purgathofer. Global ray-bundle tracing with hardware acceleration. In *Rendering Techniques '98*, pages 247–258, 1998.

[SSW86] M. Samek, C. Slean, and H. Weghorst. Texture Mapping and Distortion in Digital Graphics. *Visual Computer*, 3:313–320, 1986.

[Str87] W. Strasser. *Advanced in computer graphics I*. Springer [etc.], Eurographics seminars, Eurographics workshop on graphics hardware, Lisbon, 1987.

[Wat89] A. Watt. *Fundamentals of Three-dimensional Computer Graphics*. Addison-Wesley, 1989.

[WWD+95] M. White, M. D. Waller, G. J. Dunnett, P. F. Lister, and R. L. Grimsdale. Graphics ASIC Design Using VHDL. *Computer & Graphics*, 19(2):301–308, 1995.

[Xil01] Xilinx. *Xilinx Synthesis Technology, Series Spartan-II FPGA Family Members*. XILINX, 2001.

[YR97] Chengfu Yao and Jon G. Rokne. Applying Rounding-Up Integral Linear Interpolation to the Scan-Conversion of Filled Polygons. *Computer Graphics Forum*, 16(2):101–106, 1997.

Appendix

Mapping the RGB interpolators of the quadratic interpolation for the lower half triangle on Xilinx Spartan2e device, FPGA Family Members, xc2s100e type

```
library IEEE;
use IEEE.std_logic_$1164$.all;
library unisim;
use unisim.vcomponents.all;

package newpack is
subtype sbyte is STD_LOGIC_VECTOR$(11$ downto $0$);
subtype sword is STD_LOGIC_VECTOR($31$ downto $0$);
subtype slong_word is STD_LOGIC_VECTOR($47$ downto $0$);
end newpack;
package body newpack is
end newpack;

library IEEE;
use IEEE.std_logic_$1164$.all;
library unisim;
use unisim.vcomponents.all;
use work.newpack.all;
entity IXYinterface is
  port(StepIXsYR,StepIX1R,StepIY1R,InitIXsYR,InitIY1R, InitIY2R:out sword;
       StepIXsYG,StepIX1G,StepIY1G,InitIXsYG,InitIY1G, InitIY2G,
       StepIXsYB,StepIX1B,StepIY1B,InitIXsYB,InitIY1B, InitIY2B:out sword;
       Sel: in std_logic_vector(4 downto 0);
       iclk: in std_logic; idata: in sword);
end IXYinterface;

Architecture behaviour of IXYinterface is
begin
register_field_process:
process(iclk) -- synchronous register load at rising edge of iclk
begin
if iclk’event and iclk = ’1’ then
case Sel is
 when "00000" => StepIXsYR <= idata;
 when "00001" => StepIX1R <= idata;
 when "00010" => StepIY1R <= idata;
```

```
 when "00011" => InitIXsYR <= idata;
 when "00100" => InitIY1R <= idata;
 when "00101" => InitIY2R <= idata;
 when "00110" => StepIXsYG <= idata;
 when "00111" => StepIX1G <= idata;
 when "01000" => StepIY1G <= idata;
 when "01001" => InitIXsYG <= idata;
 when "01010" => InitIY1G <= idata;
 when "01011" => InitIY2G <= idata;
 when "01100" => StepIXsYB <= idata;
 when "01101" => StepIX1B <= idata;
 when "01110" => StepIY1B <= idata;
 when "01111" => InitIXsYB <= idata;
 when "10000" => InitIY1B <= idata;
 when "10001" => InitIY2B <= idata;
 when others => null;
end case;
end if;
end process;
end behaviour;

library IEEE;
use IEEE.std_logic_1164.all;
library IEEE;
use IEEE.std_logic_signed.all;
library unisim;
use unisim.vcomponents.all;
use work.newpack.all;

ENTITY Interpolator Is
      PORT(clk,load_step: in STD_LOGIC; InitVal,StepVal: in sword;
  OutVal: out sword);
end Interpolator;

Architecture Behaviour of Interpolator is
signal Adder_Out, Reg_Out: sword;
begin
register_process:
process(clk) -- sychronous load/step at rising edge of clk
begin
  if clk'event and clk = '1' then
      if load_step = '1' then
           Reg_Out <= InitVal ;
      else
           Reg_Out <= Adder_Out ;
      end if;
  end if;
end process;
```

```
adder_process:
process(Reg_Out,Stepval)
begin
           Adder_Out <= Reg_out + StepVal ;
end process;

OutVal <= Reg_Out;
end Behaviour;

library IEEE;
use IEEE.std_logic_1164.all;
library unisim;
use unisim.vcomponents.all;
use work.newpack.all;
entity IXYgenerator is
  port(clk,startX,startY:in STD_LOGIC;
       StepIXsYR,StepIX1R,StepIY1R,InitIXsYR,InitIY1R, InitIY2R:in sword;
       StepIXsYG,StepIX1G,StepIY1G,InitIXsYG,InitIY1G, InitIY2G,
       StepIXsYB,StepIX1B,StepIY1B,InitIXsYB,InitIY1B, InitIY2B:in sword;
       IXsYR,IX1R,IXYR,IXYG,IXYB:out sbyte);
end IXYgenerator;

architecture structure of IXYgenerator is
component  interpolator
  port(clk,load_step:in STD_LOGIC; InitVal,StepVal:in sword;
       OutVal:out sword);
end component;
component  Uinterpolator
  port(clk,load_step:in STD_LOGIC; InitVal,StepVal:in sword;
       OutVal:out sword);
end component;

signal StepIXsYRs,InitIXsYRs, IXsYoutRs : sword;
signal StepIX1Rs, IX1outRs, IX2outRs : sword;
signal StepIY1Rs, InitIY1Rs, IY1outRs : sword;
signal InitIY2Rs, IY2outRs: sword;
signal StepIXsYGs, InitIXsYGs, IXsYoutGs : sword;
signal StepIX1Gs,  IX1outGs, IX2outGs : sword;
signal StepIY1Gs,  InitIY1Gs, IY1outGs : sword;
signal InitIY2Gs,  IY2outGs: sword;
signal StepIXsYBs, InitIXsYBs, IXsYoutBs : sword;
signal StepIX1Bs,  IX1outBs, IX2outBs : sword;
signal StepIY1Bs,  InitIY1Bs, IY1outBs : sword;
signal InitIY2Bs,  IY2outBs: sword;
signal dIX1clk, dIY1clk,dIY2clk,dIXsYclk,dIX2clk,dclk:STD_LOGIC;

begin
dclk <= clk;
dIY2clk <= dclk and startX;
dIY1clk <= dclk and startX;
dIX2clk   <= dclk ;
dIX1clk  <= dclk ;
```

```
dIXsYclk <= dclk and startX;

IXsYRI:Uinterpolator
      port map (dIXsYclk,startY,InitIXsYRs,StepIXsYRs,IXsYoutRs);
 IX1RI:interpolator
      port map (dIX1clk,startX,IXsYoutRs,StepIX1Rs,IX1outRs);
IX2RI:interpolator
      port map (dIX2clk,startX,IY2outRs,IX1outRs,IX2outRs);
 IY1RI:interpolator
      port map (dIY1clk,starty,InitIY1Rs,StepIY1Rs,IY1outRs);
IY2RI:interpolator
      port map (dIY2clk,starty,InitIY2Rs,IY1outRs,IY2outRs);
IXsYGI:Uinterpolator
      port map (dIXsYclk,starty,InitIXsYGs,StepIXsYGs,IXsYoutGs);
 IX1GI:interpolator
      port map (dIX1clk,startX,IXsYoutGs,StepIX1Gs,IX1outGs);
IX2GI:interpolator
      port map (dIX2clk,startX,IY2outGs,IX1outGs,IX2outGs);
 IY1GI:interpolator
      port map (dIY1clk,starty,InitIY1Gs,StepIY1Gs,IY1outGs);
IY2GI:interpolator
      port map (dIY2clk,starty,InitIY2Gs,IY1outGs,IY2outGs);
IXsYBI:Uinterpolator
      port map (dIXsYclk,starty,InitIXsYBs,StepIXsYBs,IXsYoutBs);
 IX1BI:interpolator
      port map (dIX1clk,startX,IXsYoutBs,StepIX1Bs,IX1outBs);
IX2BI:interpolator
      port map (dIX2clk,startX,IY2outBs,IX1outBs,IX2outBs);
 IY1BI:interpolator
      port map (dIY1clk,starty,InitIY1Bs,StepIY1Bs,IY1outBs);
IY2BI:interpolator
      port map (dIY2clk,starty,InitIY2Bs,IY1outBs,IY2outBs);

StepIXsYRs <= StepIXsYR;
InitIXsYRs <= InitIXsYR;
StepIX1Rs <= StepIX1R;
StepIY1Rs <= StepIY1R;
InitIY1Rs <= InitIY1R
InitIY2Rs <= InitIY2R;
IXsYR<= IXsYoutRs(31 downto 20);
IX1R <= IX1outRs(31 downto 20);
IXYR <= IX2outRs(31 downto 20);
StepIXsYGs <= StepIXsYG;
InitIXsYGs <= InitIXsYG;
StepIX1Gs <= StepIX1G;
StepIY1Gs <= StepIY1G;
InitIY1Gs <= InitIY1G;
InitIY2Gs <= InitIY2G;
IXYG <= IX2outGs(31 downto 20);
StepIXsYBs <= StepIXsYB;
InitIXsYBs <= InitIXsYB;
StepIX1Bs <= StepIX1B;
```

```
StepIY1Bs <= StepIY1B;
InitIY1Bs <= InitIY1B;
InitIY2Bs <= InitIY2B;
IXYB <= IX2outBs(31 downto 20);
end structure;

library IEEE;
use IEEE.std_logic_1164.all;
library IEEE;
use IEEE.std_logic_signed.all;
library unisim;
use unisim.vcomponents.all;
use work.newpack.all;

ENTITY UInterpolator Is
    PORT(clk,load_step: in STD_LOGIC; InitVal,StepVal: in sword;
         OutVal: out sword);
end UInterpolator;

Architecture Behaviour of UInterpolator is
signal Adder_Out, Reg_Out: sword;
begin
register_process:
process(clk,load_step, InitVal)
begin
  if load_step = '1' then
 Reg_Out <= InitVal ;
  else
    if clk'event and clk = '1' then
           Reg_Out <= Adder_Out ;
    end if;
  end if;
end process;

adder_process:
process(Reg_Out,Stepval)
begin
Adder_Out <= Reg_Out + Stepval ;
end process;

OutVal <= Reg_Out;
end Behaviour;

library IEEE;
use IEEE.std_logic_1164.all;
library unisim;
use unisim.vcomponents.all;
```

```
use work.newpack.all;
entity conn is
port(Sel: in std_logic_vector(4 downto 0); iclk: in std_logic;
     idata: in sword; clk,startX,startY:in STD_LOGIC;
     IXsYRb,IX1Rb,IXYRb,IXYGb,IXYBb:out sbyte);
end conn;

architecture structure of conn is
component IXYinterface
port(StepIXsYR,StepIX1R,StepIY1R,InitIXsYR,InitIY1R, InitIY2R:out sword;
       StepIXsYG,StepIX1G,StepIY1G,InitIXsYG,InitIY1G, InitIY2G,
       StepIXsYB,StepIX1B,StepIY1B,InitIXsYB,InitIY1B, InitIY2B:out sword;
       Sel: in std_logic_vector(4 downto 0);
       iclk: in std_logic; idata: in sword);
end component;

component IXYgenerator
  port(clk,startX,startY:in STD_LOGIC;
       StepIXsYR,StepIX1R,StepIY1R,InitIXsYR,InitIY1R, InitIY2R:in sword;
       StepIXsYG,StepIX1G,StepIY1G,InitIXsYG,InitIY1G, InitIY2G,
       StepIXsYB,StepIX1B,StepIY1B,InitIXsYB,InitIY1B, InitIY2B:in sword;
       IXsYR,IX1R,IXYR,IXYG,IXYB:out sbyte);
end component;

signal StepIXsYR,StepIX1R,StepIY1R,InitIXsYR,InitIY1R, InitIY2R,
       StepIXsYG,StepIX1G,StepIY1G,InitIXsYG,InitIY1G, InitIY2G,
       StepIXsYB,StepIX1B,StepIY1B,InitIXsYB,InitIY1B, InitIY2B:sword;
signal IXsYR,IX1R,IXYR,IXYG,IXYB: sbyte;
begin
IXsYRb <= IXsYR;
IX1Rb <= IX1R;
IXYRb <= IXYR;
IXYGb <= IXYG;
IXYBb <= IXYB;

Generator_hw:
IXYgenerator port map (clk,startX,startY,
       StepIXsYR,StepIX1R,StepIY1R,InitIXsYR,InitIY1R, InitIY2R,
       StepIXsYG,StepIX1G,StepIY1G,InitIXsYG,InitIY1G, InitIY2G,
       StepIXsYB,StepIX1B,StepIY1B,InitIXsYB,InitIY1B, InitIY2B,
       IXsYR,IX1R,IXYR,IXYG,IXYB);

Interface_hw:
IXYinterface port map (StepIXsYR,StepIX1R,StepIY1R,
 InitIXsYR,InitIY1R, InitIY2R,StepIXsYG,StepIX1G,StepIY1G,
 InitIXsYG,InitIY1G, InitIY2G,StepIXsYB,StepIX1B,StepIY1B,
 InitIXsYB,InitIY1B, InitIY2B, Sel, iclk, idata);
end structure;
```

```
Design Information
------------------
Target Device  : x2s100e
Target Package : ft256
Target Speed   : -6

Design Summary
--------------
   Number of Slices:                883 out of  1,200   73%
   Number of Slices containing
      unrelated logic:                0 out of    883    0%
   Number of Slice Flip Flops:    1,056 out of  2,400   44%
   Number of 4 input LUTs:        1,183 out of  2,400   49%
   Number of bonded IOBs:            99 out of    178   55%
   Number of GCLKs:                   2 out of      4   50%
   Number of GCLKIOBs:                2 out of      4   50%
Total equivalent gate count for design:  18,336
Additional JTAG gate count for IOBs:  4,848

---- Target Parameters
Target Device                     : xc2s100e-ft256-6
Target Technology                 : spartan2e

---- Source Options
Automatic FSM Extraction          : YES
FSM Encoding Algorithm            : Auto
FSM Flip-Flop Type                : D
Mux Extraction                    : YES
Resource Sharing                  : YES
Complex Clock Enable Extraction   : YES
ROM Extraction                    : Yes
RAM Extraction                    : Yes
RAM Style                         : Auto
Mux Style                         : Auto
Decoder Extraction                : YES
Priority Encoder Extraction       : YES
Shift Register Extraction         : YES
Logical Shifter Extraction        : YES
XOR Collapsing                    : YES
Automatic Register Balancing      : No

---- Target Options
Add IO Buffers                    : YES
Equivalent register Removal       : YES
Add Generic Clock Buffer(BUFG)    : 4
Global Maximum Fanout             : 100
Register Duplication              : YES
Move First FlipFlop Stage         : YES
Move Last FlipFlop Stage          : YES
Slice Packing                     : YES
Pack IO Registers into IOBs       : auto
Speed Grade                       : 6
```

```
---- General Options
Optimization Criterion             : Speed
Optimization Effort                : 1
Check Attribute Syntax             : YES
Keep Hierarchy                     : No
Global Optimization                : AllClockNets
Write Timing Constraints           : No
=========================================================
HDL Synthesis Report FPGA Macro Statistics
# Registers                        : 33
  32-bit register                  : 33
# Multiplexers                     : 15
  2-to-1 multiplexer               : 15
# Adders/Subtractors               : 15
  32-bit adder                     : 15
=========================================================
Design Statistics
# IOs                              : 101

Cell Usage :
# BELS                             : 2114
#      BUF                         : 5
#      GND                         : 1
#      LUT2                        : 212
#      LUT2_D                      : 15
#      LUT2_L                      : 450
#      LUT3                        : 35
#      LUT3_L                      : 465
#      LUT4                        : 1
#      MUXCY                       : 465
#      XORCY                       : 465
# FlipFlops/Latches                : 1056
#      FD                          : 384
#      FDCP                        : 96
#      FDE                         : 576
# Clock Buffers                    : 2
#      BUFGP                       : 2
# IO Buffers                       : 99
#      IBUF                        : 39
#      OBUF                        : 60
=========================================================
Timing Summary:
---------------
   Minimum period: 7.148ns (Maximum Frequency: 139.899MHz)
   Minimum input arrival time before clock: 14.470ns
   Maximum output required time after clock: 6.778ns
```