Quadratic Interpolation in Hardware Rendering

Category: research

Abstract

Rendering systems often represent curved surfaces as a mesh of planar polygons that are shaded to add realism and to restore a smooth appearance. To increase the rendering speed, complex operations such as the solution of the rendering equation or texture transformation are executed just for a few knot points and the values at other points are interpolated. Usually linear transformation is used since it can be easily implemented in hardware. However, color distribution and texture transformation can be strongly non-linear for which linear interpolation may introduce severe artifacts. Thus this paper proposes quadratic interpolation to tackle this problem and demonstrates that it can be implemented in hardware. The software simulation and the VHDL description of the shading hardware are also presented.

1 Introduction

Computer graphics aims at rendering complex virtual world models and presenting the image for the user. To obtain an image of a virtual world, surfaces visible in pixels are determined, and the rendering equation or its simplified form is used to calculate the intensity of these surfaces, defining 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. Real-time systems, however, allow just a few tens of nanoseconds for the computation of a single pixel, which results in a continuous driving force to develop faster graphics hardware [14, 20, 15, 8].

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 interpolated from these representative points by much simpler expressions. One way of obtaining this is the tessellation of the original surfaces to polygon meshes and using the vertices of the polygons as representative points. In this paper only triangle mesh models are considered. A simple interpolation scheme would compute the color at the vertices and linearly interpolate it inside the triangle (Gouraud shading [10]). However linear interpolation can introduce severe artifacts (left of figure 2). The core of the problem is that the color can 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).

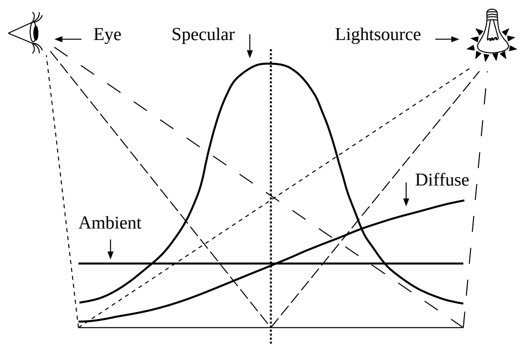

Figure 1: Intensity of ambient, diffuse, and specular reflections

The artifacts of Gouraud shading can be eliminated by a non-linear interpolation called Phong shading [16] (right of figure 2). In Phong shading, vectors used by the rendering equation are interpolated from the real vectors at the vertices of the approximating triangle. In simpler algorithms only the normal vectors are interpolated while the light and view vectors are constant. In more precise computations, the view and light vectors are also interpolated. The interpolated vectors are normalized and the rendering equation is evaluated at each pixel for diffuse and specular reflections and for each light source, which is rather time consuming. The main problem of Phong shading is that it requires complex operations on the pixel level, including interpolation and normalization of the normal, viewing and the light vectors, calculation of their dot products, exponentiation if Phong reflection model is used, and multiplications and additions. The hardware implementation is not feasible only if the number and type of lightsources are limited and the underlying formulae are simplified [2].

The superior rendering quality of Phong shading forced research to try to find a reasonable compromise between Gouraud and Phong algorithms, that keeps the image quality but also allows for hardware implementation. In Textronix terminals, for example, the method called pseudo-Phong shading was implemented. Pseudo-Phong shading recursively decomposes the triangles into small triangles setting the vectors at the vertices according to a linear formula, and uses Gouraud shading when the small triangles are rendered. If the sizes 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. Unfortunately, the artifacts of Gouraud shading are visible even in highly tessellated surfaces (figure 11). Another family of algorithms used highlight tests [22] to de-

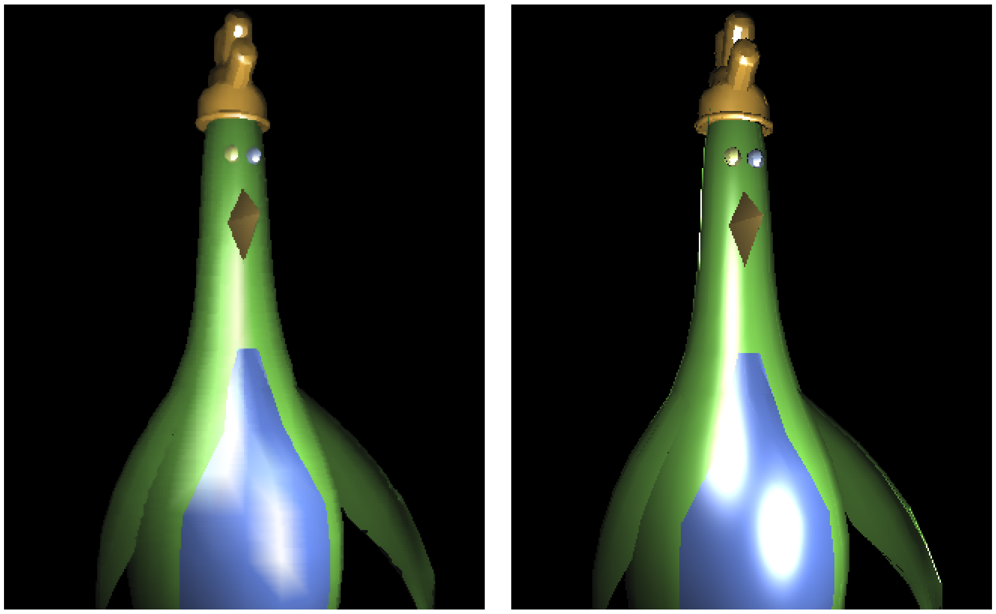
Figure 2: Comparison of linear (Gouraud) interpolation (left) and non-linear interpolation by Phong shading (right).

termine 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 [7] extended the incremental approach of Gouraud shading to Phong shading supposing only diffuse reflection. He obtained the reflected radiance of a single light source as a direct function of the pixel coordinates and evaluated this function using forward differences. His incremental formulae separately calculated the dot product of the normal and the light vector, and the length of the normal vector, thus a division and a square-root operation were still needed for the evaluation of a single pixel. Based on this, Bishop proposed a simplification using Taylor's approximation for both the diffuse and the Phong-Blinn reflection in [3]. The determination of the derivatives of the reflected radiance is quite complicated and requires expensive computation, and this computation must be repeated separately for diffuse and specular reflections and for each light source. 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 [5] compared different simplification strategies of the Phong illumination formulae and vector interpolation. Spherical interpolation elegantly traces back the interpolation to the interpolation of a single angle inside a scan-line [13]. However, finding the parameters of a scan-line is also rather complicated and the method requires 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. Reflection shading [21, 12] tackles the problem of vector normalization by finding a vector that is halfway between the interpolated unnormalized vector and a given normalized vector, and then the halfway vector is interpolated and the given vector is reflected on the halfway vector each time. Finally, Phong shading can also be replaced by texturing [4] in special cases.

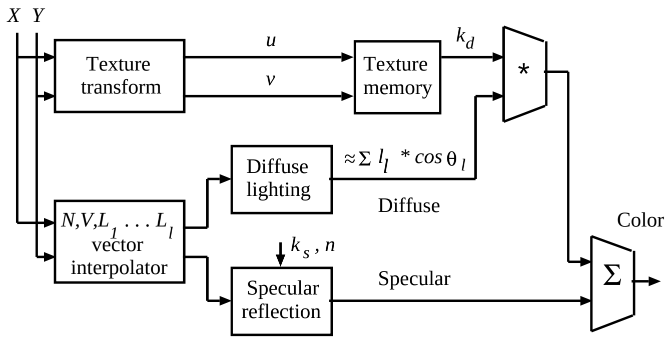

Figure 3: Conventional rendering with Phong shading and texture mapping without interpolation

Another non-linear problem arises in image order texture mapping. For triangles, the screen coordinates and the texture coordinates are connected by a homogeneous linear transformation [19] thus for a pixel X,Y the corresponding texel coordinates u,v can be obtained as

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

where $a_u, \dots f_v$ depend on the positions of the triangle in the texture and image spaces. Note that this operation also contains divisions that are quite intensive computationally and makes the mapping non-linear. Implementing divi-

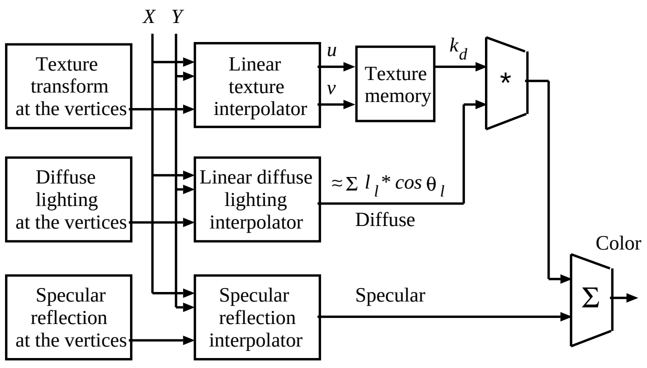
X
Y
Texture transform at the vertices
Linear texture interpolator
u
v
Texture memory
k_d
*
Diffuse lighting at the vertices
Linear diffuse lighting interpolator
≈Σ l_l * cos θ_l
Diffuse
Color
Specular reflection at the vertices
Specular reflection interpolator
Specular
Σ

Figure 4: Linear interpolation, i.e. Gouraud shading and linear texture mapping

sion in hardware is difficult and can be the bottleneck of texture mapping [1]. Approximating this function by a linear transformation, on the other hand, makes the perspective distortion incorrect [9, 17] (figure 12). Demirer [6] proposed a Chebishev polynomial approximation to avoid division.

Summarizing, the main problem of the original shading pipeline used by Phong shading (figure 3) is that it involves complex operations such as lighting calculations and texture transformations, which makes its direct hardware realization impossible. Traditionally, this problem is attacked by linear interpolation as shown in figure 4, but the linear interpolation of the strongly non-linear functions degrades the image quality [18].

In this paper we propose a new interpolation scheme (figure 5) 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.

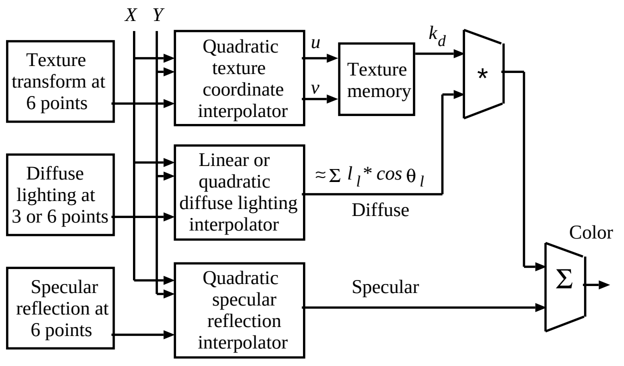
X
Y
Texture transform at 6 points
Quadratic texture coordinate interpolator
u
v
Texture memory
k_d
*
Diffuse lighting at 3 or 6 points
Linear or quadratic diffuse lighting interpolator
≈Σ l_l * cos θ_l
Diffuse
Color
Specular reflection at 6 points
Quadratic specular reflection interpolator
Specular
Σ

Figure 5: Quadratic rendering

2 Quadratic interpolation

Our approach is in between linear interpolation and brute force methods not using interpolation at all. The function of interest, as for example the rendering equation or the texture transformation, is evaluated in a few representative points and the interpolation is done in image space as in Gouraud shading. However, the interpolation is not linear, but rather quadratic. Since a quadratic form has six degrees of freedom, the function will be evaluated at six representative points on the triangle and it is interpolated from the values at these representative points. Let us approximate function I inside the triangle by the following two-variate quadratic form:

$$I(X,Y) = T_5X^2 + T_4XY + T_3Y^2 + T_2X + T_1Y + T_0. \quad (1)$$

To find the unknown parameters $T_0, \ldots, T_5$, the function values are 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 should be 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 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:

$$\begin{aligned}
I_1 &= T_0, \\
I_2 &= T_5X_2^2 + T_4X_2Y_2 + T_3Y_2^2 + T_2X_2 + T_1Y_2 + T_0, \\
I_{12} &= T_5\frac{X_2^2}{4} + T_4\frac{X_2Y_2}{4} + T_3\frac{Y_2^2}{4} + T_2\frac{X_2}{2} + T_1\frac{Y_2}{2} + T_0, \\
I_3 &= T_5X_3^2 + T_4X_3Y_3 + T_3Y_3^2 + T_2X_3 + T_1Y_3 + T_0, \\
I_{13} &= T_5\frac{X_3^2}{4} + T_4\frac{X_3Y_3}{4} + T_3\frac{Y_3^2}{4} + T_2\frac{X_3}{2} + T_1\frac{Y_3}{2} + T_0, \\
I_{23} &= T_5\frac{(X_2+X_3)^2}{4} + T_4\frac{X_2+X_3}{2}\cdot\frac{Y_2+Y_3}{2} + \\
&+ T_3\frac{(Y_2+Y_3)^2}{4} + T_2\frac{X_2+X_3}{2} + T_1\frac{Y_2+Y_3}{2} + T_0.
\end{aligned}$$

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

$$\begin{aligned}
T_0 &= I_1, \\
T_1 &= \frac{C_3X_2 - C_2X_3}{X_2Y_3 - Y_2X_3}, \\
T_2 &= \frac{C_2Y_3 - C_3Y_2}{X_2Y_3 - Y_2X_3}, \\
T_3 &= \frac{2C_{12} - T_5X_2^2 - T_4X_2Y_2}{Y_2^2}, \\
T_4 &= \frac{(4C_{13}Y_2 - C_{23}Y_3)D_{23} - (4C_{12}Y_3 - C_{23}Y_2)D_{32}}{E_{32}D_{23} - E_{32}D_{32}}, \\
T_5 &= \frac{(4C_{12}Y_3 - C_{23}Y_2)E_{32} - (4C_{13}Y_2 - C_{23}Y_3)E_{23}}{E_{32}D_{23} - E_{32}D_{32}},
\end{aligned}$$

where

$$\begin{aligned}
C_2 &= 4I_{12} - 3I_1 - I_2, \\
C_{12} &= I_1 + I_2 - 2I_{12}, \\
C_3 &= 4I_{13} - 3I_1 - I_3, \\
C_{13} &= I_1 + I_3 - 2I_{13}, \\
C_{23} &= 4I_1 - 4I_{12} - 4I_{13} + 4I_{23}, \\
D_{23} &= 2X_2^2Y_3 - 2X_2Y_2X_3, \\
D_{32} &= 2Y_2X_3^2 - 2X_2X_3Y_3, \\
E_{23} &= X_2Y_2Y_3 - Y_2^2X_3, \\
E_{32} &= Y_2X_3Y_3 - X_2Y_3^2.
\end{aligned}$$

The calculation of $T_0, \ldots, T_5$ parameters requires 25 additions, 51 multiplications, and 5 divisions. Having determined the $T_0, \ldots, T_5$ values, if $I(X,Y)$ is needed, then X and Y are substituted into equation (1). This quadratic form will be evaluated by simple additions according to the incremental concept.

3 Error control

The method proposed in the previous section approximates a non-linear function by a quadratic formula. This function is either the radiance or the texture address in our case. If the triangles are too big and the radiance or the texture address change quickly due to a highlight or to the expansion of the texture transformation, 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.

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 (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

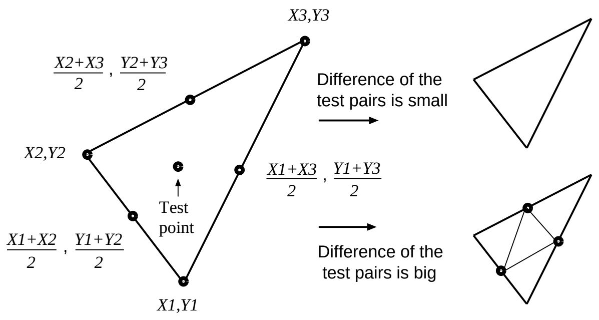

Figure 6: Highlight test and adaptive subdivision

function evaluation is affordable and during subdivision the already computed function values can be reused.

4 Hardware implementation of the quadratic interpolation

This section reviews the implementation strategies of simple functions on scan-lines that are used to fill horizontal sided image space triangles. If the image space triangle is not formed as horizontal sided triangle, then it should be divided into two parts, a lower and an upper. In this section we will consider only the lower horizontal sided triangle. Image space triangle and horizontal sided triangle are shown in figure 7.

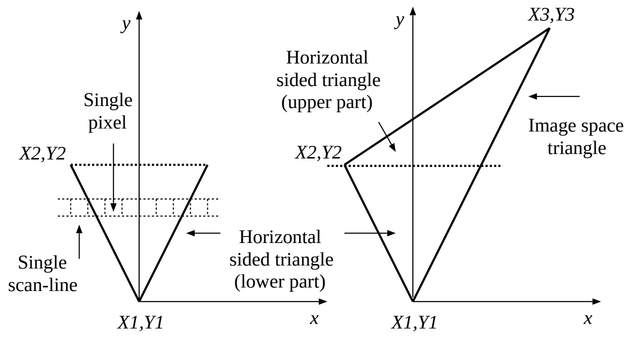

Figure 7: Image space and horizontal sided triangles

If we implemented equation (1) directly, the hardware should compute floating point multiplications and additions for each value, which are rather demanding. To eliminate the multiplications, the incremental concept is used, which traces back the evaluation of the function $I(X,Y)$ to the computation of an increment from the previous values, for instance, from $I(X-1,Y)$. The increments can then be evaluated by simple additions. A triangle filling algorithm should generate the sequence of (X,Y) integer values called pixels that are inside a horizontal sided triangle. The algorithm generates the pixels scan-line by scan-line. In a single scan-line the Y coordinate is constant. To simplify equation (1), we use 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

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

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

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

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 the following equations:

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

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

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

where

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

Now let us discuss the computation on the start edge. When the algorithm steps onto the next scan-line, both $I(X,Y)$ and $\Delta I(X,Y)$ should be recomputed with the data of the new scan-line. The incremental concept can also be used for these computations, which traces back these updates to two additions. The first application of the incremental concept reduces the computation of the quadratic function $I(X,Y)$ to a linear one:

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

where

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

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

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

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

To obtain the incremental value $\Delta I(X,Y)$ at the start edge, we should apply the incremental concept only once since it is already a linear function (equation (2)):

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

Let us group these formulae in the following algorithm:

```
$X_{start} = X_1$, $X_{end} = X_1$
Compute $I_{start}(X,Y)$, $\Delta I_{start}(X,Y)$, $\Delta I(X_{start},Y)$ at $X_1,Y_1$
for $Y = Y_1$ to $Y_2$ do
    $I(X,Y) = I_{start}(X,Y)$
    $\Delta I(X,Y) = \Delta I(X_{start},Y)$
    for $X = X_{start}$ to $X_{end}$ do
        write( $X$, $Y$, $I(X,Y)$ )
        $I(X,Y)$ += $\Delta I(X,Y)$
        $\Delta I(X,Y)$ += $2T_5$
    endfor
    $\Delta I(X_{start},Y)$ += $2T_5A_{start} + T_4$
    $I_{start}(X,Y)$ += $\Delta I_{start}(X,Y)$
    $\Delta I_{start}(X,Y)$ += $2(T_5A_{start}^2 + T_4A_{start} + T_3)$
    $X_{start}$ += $A_{start}$, $X_{end}$ += $A_{end}$
endfor
```

Note that function I and the parameters are not integers, and if we ignored the fractional part, the incremental formula would accumulate the error to an unacceptable level. The realization of floating point arithmetic 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. The number of bits in the fractional part has to be set to avoid incorrect I calculations due to the cumulative error in I. 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

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

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

$$(M_X(M_X-1)+M_Y(M_Y-1))\cdot 2^{-(b_I+1)} < 1 \Longrightarrow$$

$$b_I > \log(M_X(M_X-1)+M_Y(M_Y-1))-1.$$

If the display has 1280×1024 resolution, this results in the requirement of 22 fractional bits.

The hardware implementation is shown in figure 8. 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.

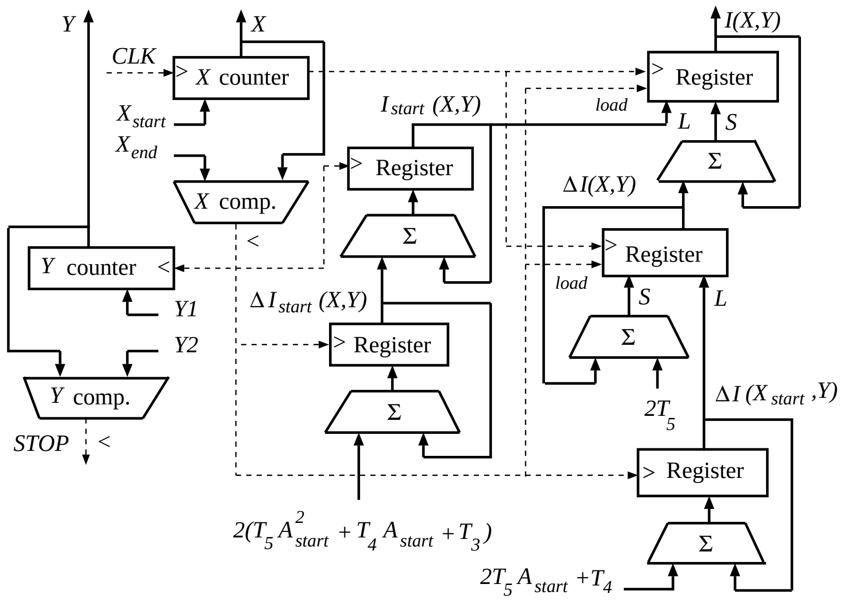

Figure 8: Hardware implementation of two-variate quadratic functions

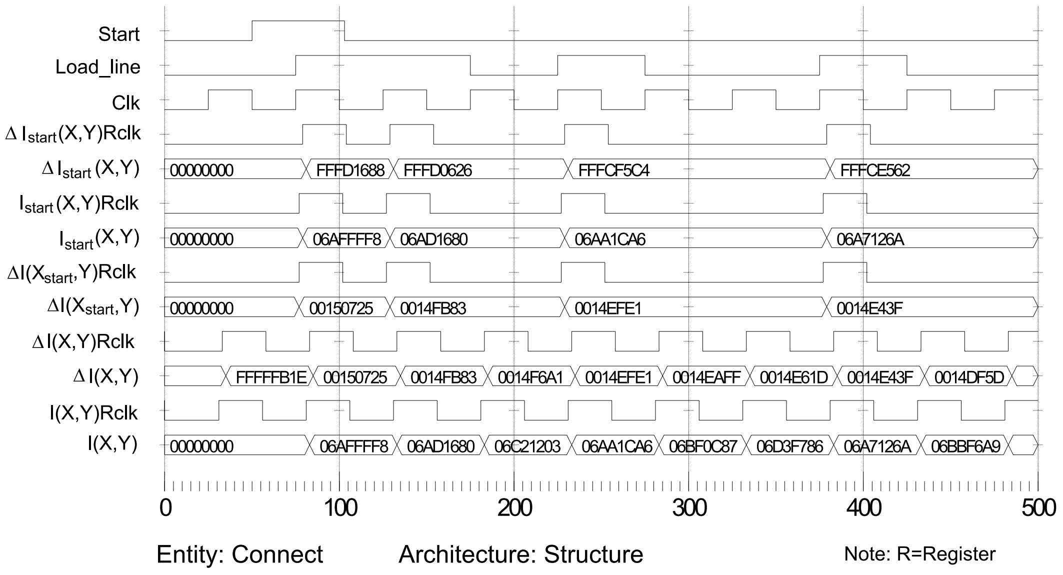

Figure 9: Timing diagram of the hardware generated by the VHDL simulator

5 Simulation results

The proposed algorithm has been implemented first in Microsoft Visual C++ and tested as a software. In figures 10 and 11, spheres tessellated on different levels are compared. Gouraud shading evaluates the rendering 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 of the coarsely tessellated spheres are as follows: Gouraud shading 230 msec, quadratic shading 250 msec, and Phong shading 450 msec. Note that Gouraud shading performs poorly on coarsely tessellated surfaces, but the visual quality of quadratic shading and Phong shading is similar. On the other hand, concerning the speed and the suitability for hardware implementation, quadratic shading is close to Gouraud shading. Figure 13 shows a more complex scene with normal tessellation level. Looking at these images we can conclude that quadratic shading is visually superior to Gouraud shading and indistinguishable from classical Phong shading.

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 12). Note that linear transformation distorts the textures in an unacceptable way, while quadratic approximation handles the perspective shrinking properly.

Having tested the software implementation, the hardware realization was specified in VHDL and simulated in ModelTech environment. The timing diagram of the algorithm is shown by figure 9. The delay times are according to XILINX XCV300-6 FPGA. In this figure we can follow the operation of the hardware. The hardware can generate one pixel per one clock cycle. The length of the 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.

6 Conclusions

This paper proposed a new rendering strategy where the color and the texture coordinates are evaluated by the rendering equation at six representative points, three on the vertices and the other three halfway between the vertices, then they are interpolated inside the triangle according to a quadratic scheme. The algorithm has also been transformed to a hardware design that has been simulated in VHDL demonstrating that 50 nsec pixel drawing time can be obtained. Even if the screen has about 1000×1000 resolution, the complete image can be redrawn 16 times per second which provides the illusion of continuous motion.

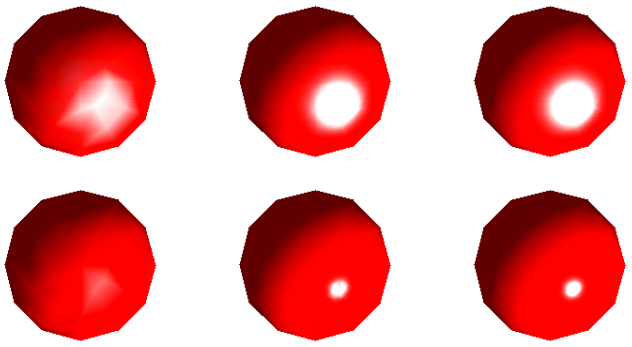

Figure 10: Rendering coarsely tessellated spheres (168 triangles) of specular exponents $n = 5$ (top) and $n = 50$ (bottom) with Gouraud shading (left), quadratic shading (middle) and Phong shading (right)

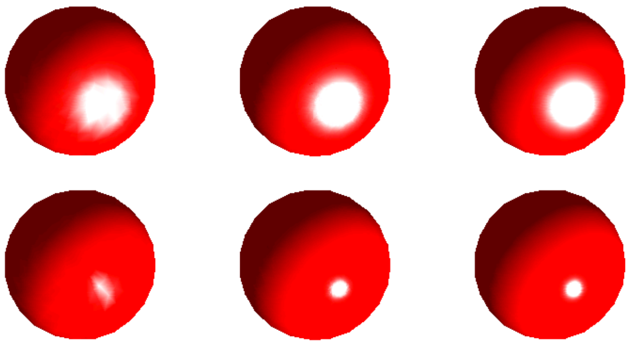

Figure 11: Rendering highly tessellated spheres (690 triangles) of specular exponents $n = 5$ (top) and $n = 50$ (bottom) with Gouraud shading (left), quadratic shading (middle) and Phong shading (right)

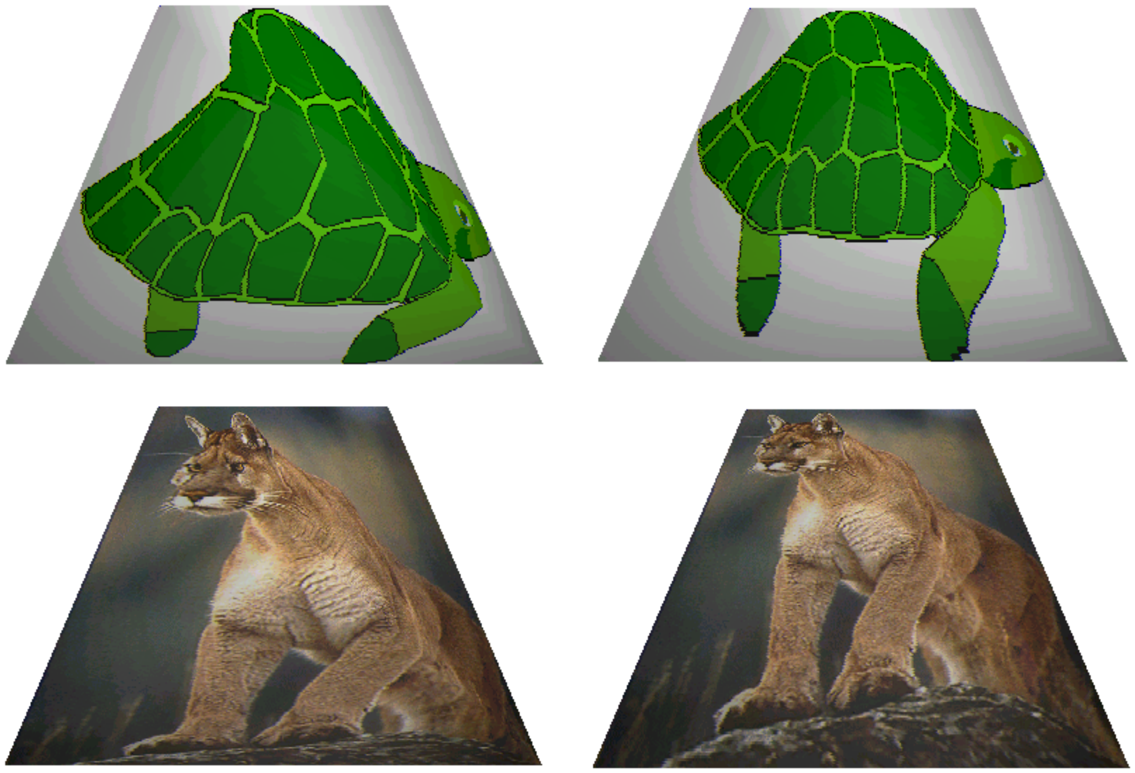

Figure 12: Texture mapping with linear (left), quadratic (right) texture transformation

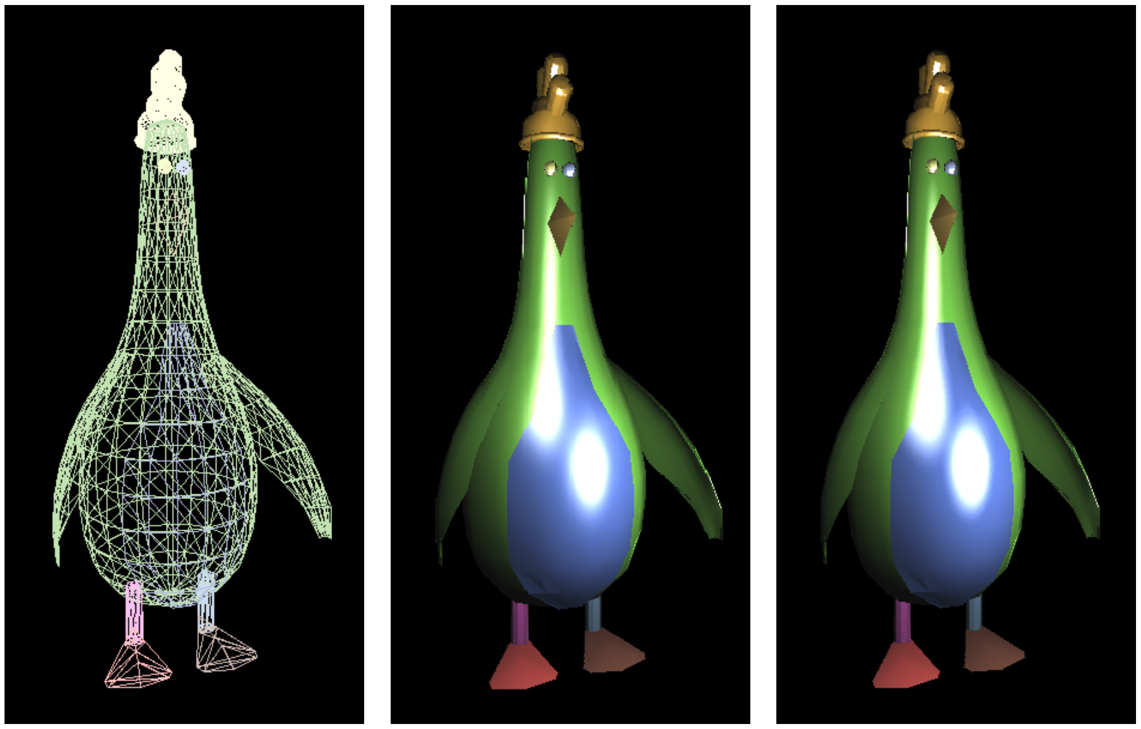

Figure 13: The mesh of a chicken (left) and its image rendered by classical Phong shading (middle) and by quadratic shading (right)

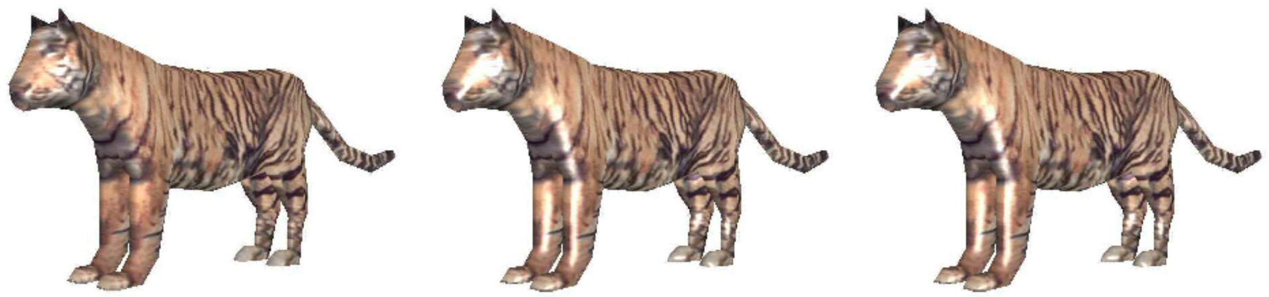
Figure 14: Coarsly tessellated textured and specular tiger with Gouraud (left) Phong (middle) and quadratic (right) shading

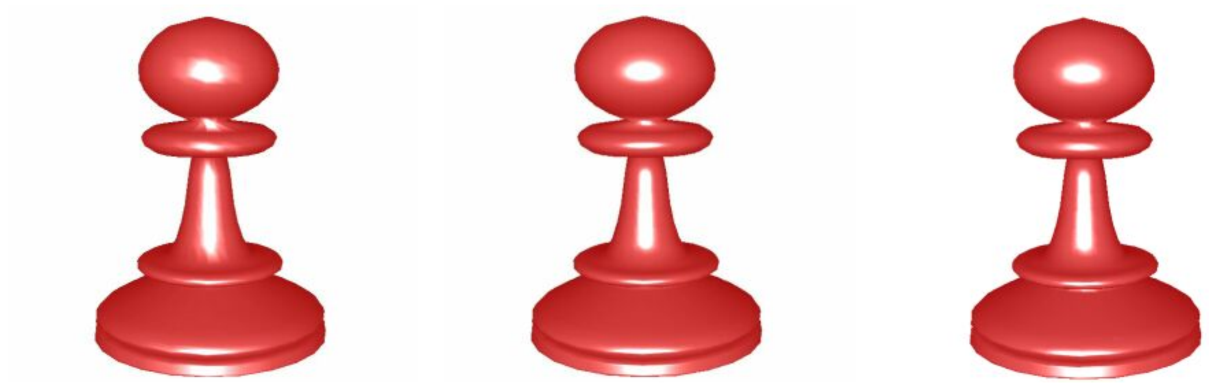
Figure 15: A pawn with Gouraud, Phong and Quadratic shading

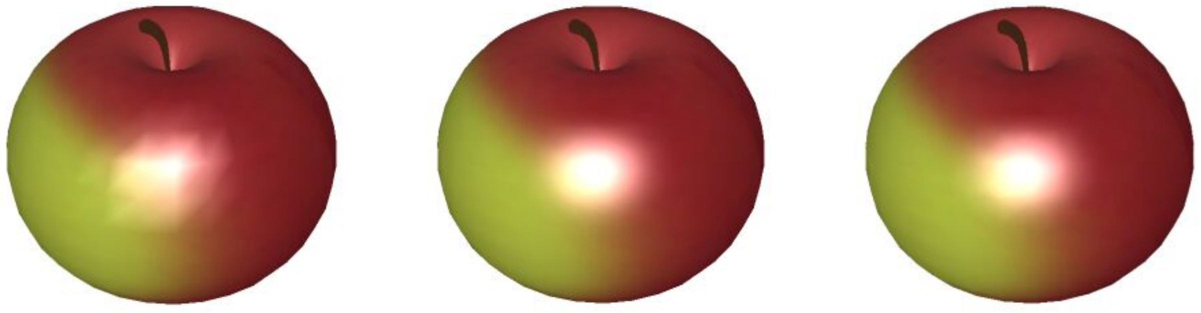
Figure 16: An apple with Gouraud, Phong and Quadratic shading

References

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

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

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

[4] D. Blythe, B. Grantham, M Kilgard, T. McReynolds, S. Nelson, C. Fowler, S. Hui, and P. Womack. Advanced graphics programming techniques using OpenGL: Course notes. Technical report, SIGGRAPH'99, 1999.

[5] U. Claussen. On reducing the Phong shading method. *Computer & Graphics*, pages 73–81, 1990.

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

[7] T. Duff. Smoothly shaded rendering of polyhedral objects on raster displays. In *Computer Graphics (SIGGRAPH '79 Proceedings)*, 1979.

[8] M. Eldridge, H. Igehy, and P. Hanrahan. Pomegranate: A fully scalable graphics architecture. In *SIGGRAPH'00*, 2000.

[9] M. Gangnet, P. Perny, and P. Coueignoux. Perspective mapping of planar textures. In *EUROGRAPHICS '82*, pages 57–71, 1982.

[10] H. Gouraud. Computer display of curved surfaces. *ACM Transactions on Computers*, C-20(6):623–629, 1971.

[11] P. Haeberli and M. Segal. Texture mapping as a fundamental drawing primitive. Technical report, Silicon Graphics, Inc., 1993. http://www-europe.sgi.com/grafica/texmap/.

[12] A. Hast, T. Barrera, and E. Bengtsson. Improved shading performance by avoiding vector normalization. In *Winter School of Computer Graphics '01*, pages 1–8, 2001. Short paper.

[13] A. M. Kuijk and E. H. Blake. Faster Phong shading via angular interpolation. *Computer Graphics Forum*, pages 315–324, 1989.

[14] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-speed rendering using image composition. In *SIGGRAPH'92*, pages 231–240, 1992.

[15] J. Montrym, D. Baum, D. Digman, and C. Migdal. InfiniteReality: A real-time graphics system. In *SIGGRAPH'97*, pages 293–302, 1997.

[16] B. T. Phong. Illumination for computer generated images. *Communications of the ACM*, 18:311–317, 1975.

[17] M. Samek, C. Slean, and H. Weghorst. Texture mapping and distortion in digital graphics. *Visual Computer*, 3:313–320, 1986.

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

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

[20] J. Torborg and J. Kajiya. Talisman: Commodity real-time 3D graphics for the PC. In *SIGGRAPH'96*, pages 353–364, 1996.

[21] D. Voorhies and J. Foran. Reflection vector shading hardware. In *Computer Graphics (SIGGRAPH '94 Proceedings)*, pages 163–166, 1994.

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

Quadratic Interpolation in Hardware Rendering

Ali Mohamed Abbas, László Szirmay-Kalos, Gábor Szijártó, Tamás Horváth and Tibor Fóris
Department of Control Engineering and Information Technology, Technical University of Budapest
Budapest, Pázmány P. s. 1/D, H-1117, HUNGARY
szirmay@iit.bme.hu

Category: research

Format: print

Contact:	László Szirmay-Kalos
	Department of Control Engineering and Information Technology,
	Technical University of Budapest
	Budapest, Pázmány P. s. 1/D, H-1117, HUNGARY
phone:	(361) 463–2030
fax:	(361) 463–2204
email:	szirmay@iit.bme.hu

Estimated # of pages: 8

Keywords: Phong shading, texture mapping, interpolation, computer graphics hardware

Rendering systems often represent curved surfaces as a mesh of planar polygons that are shaded to add realism and to restore a smooth appearance. To increase the rendering speed, complex operations such as the solution of the rendering equation or texture transformation are executed just for a few knot points and the values at other points are interpolated. Usually linear transformation is used since it can be easily implemented in hardware. However, color distribution and texture transformation can be strongly non-linear for which linear interpolation may introduce severe artifacts. Thus this paper proposes quadratic interpolation to tackle this problem and demonstrates that it can be implemented in hardware. The software simulation and the VHDL description of the shading hardware are also presented.