

The Setup for Triangle Rasterization

Anders Kugler

University of Tübingen - Computer Graphics Laboratory ⁽¹⁾

Abstract

Integrating the slope and setup calculations for triangles to the rasterizer offloads the host processor from intensive calculations and can significantly increase 3D system performance. The processing on the host is greatly reduced and much less data is passed from the host to the graphics subsystem. A setup architecture handling generalized triangle meshes and computing all necessary parameters for a high-end raster pipeline to generate Gouraud shaded, texture- and bump-mapped triangles is described and its benefits on the final bandwidth are shown. To efficiently compute the slopes and color gradients for each triangle, some implementation aspects on division and multiplication pipelines are discussed.

1 Introduction

Graphics performance is increased through the development of various hardware-supported graphics architectures. The majority of these architectures include a rasterizer to which the 3D vertex coordinates in image space and associated color values are sent. For rasterization it is common to use triangles or triangle strips as basic drawing primitives. The rasterizer interpolates the depth and color values for all the pixels bounded by the edges which define the triangles.

Triangles are planar shapes and this property suggests the use of constant increments to linearly interpolate the color and depth along the scanlines. Traditional shaded triangle scan conversion is typically performed by a pipeline of an edge-walking phase followed by the span interpolation. During edge interpolation, a triangle is scanned horizontally from top to bottom, delivering the boundaries of the triangle, the starting and ending values of RGB α and Z for

the span interpolation. Span interpolation forms the inner loop of the triangle shading pipeline; it interpolates the RGB α and z values along the current span, bounded by the starting and ending values for color and depth.

Performance bottlenecks in graphics rendering systems typically appear at four stages:

1. the world coordinate transformation to screen coordinate and computing the vertex colors
2. calculating the triangle edge slopes and increments necessary for the scan-conversion
3. the rate at which the fixed point iterators generate pixel values
4. the achieved bandwidth into the frame buffer and texture memory

1) and 2) are accomplished through the design of highly efficient floating point units, arranged as a SIMD array [1]. 3) is achieved through hyperpipelining several iteration units, allowing each unit to sustain the pixel generation requirements of multiple pixel memory buses. The level of pipelining of each iteration unit is limited by the integration grain of the technology. Hyperpipelining adds pipeline stages to the iterator until the desired rate of pixel generation is reached. Inserting pipeline stages in the iterator requires significantly less gates than replicating or parallelizing the iterators. Multiple memory buses or the use of logic embedded memories [2, 3] can provide the necessary bandwidth for the frame buffer to satisfy the requirements of 4). The iteration pipeline must support a pixel generation rate of N times the worst case bandwidth of a frame buffer DRAM in page mode, where N is the number of memory banks.

Many existing rasterizers do not include the initialisation step, but leave it to a dedicated floating-point coprocessor or to the same processor which performs the geometry processing. When the host CPU performs the initialisation, overall performance of a

⁽¹⁾Universität Tübingen

Wilhelm-Schickard-Institut für Informatik
Graphisch-Interaktive Systeme

Auf der Morgenstelle 10

D-72076 Tübingen - Germany

email: kugler@gris.uni-tuebingen.de

http://www.gris.uni-tuebingen.de/~kugler

graphics subsystem drops, because of the latency introduced by the setup in the graphics rendering pipeline and not enough bandwidth between the geometry processor and the rasterizer. To circumvent this bottleneck, some graphics systems include special-purpose coprocessors or DSP (Digital Signal Processing) chips which perform the setup for the rasterizer. The drawback of these solutions is that most such processors require a considerable number of logical interface chips which increase the size and cost of a graphics accelerator board. Second, DSPs or floating-point coprocessors are generally not optimized for 3D graphics rendering tasks. If a dedicated floating point unit is used, the setup parameters still have to be transferred to the rasterizer over a bus with a fixed bandwidth which does not increase the bandwidth over the situation where the geometry engine computes the setup values. Division usually is the slowest operation, and even if FPU's are able to serialize divisions, they do not support parallel divisions.

Moving the setup to the rasterizer reduces the number of data which is transferred from the geometry unit to the rasterizer for each drawn primitive. Furthermore, it minimizes the software overhead of raster algorithms. In this paper, we give a theoretical model from which future solutions can be derived. To our knowledge, only very few rasterizers include a setup engine on the same chip.

2 The graphics rendering pipeline

The generic pipeline for 3D graphics is shown in figure 1 [4]. Individual systems differ in the partitioning of this graphics rendering pipeline. Two areas in rendering have been subject to separate optimization: the floating-point intensive initial stages and the numerous primitive setup operations; the drawing-intensive part which scan-converts the pixels of a primitive and z-buffers them into the frame buffer. Rasterizing is computationally intensive since it must handle the

interpolation for color, texture and transparency, before it sends the final image for display to the frame buffer.

Initially, a triangle is defined by the coordinates (x, y, z) of its three vertices and the color values $RGB\alpha$ associated to each vertex. Before scan conversion, a series of increments used to walk along the edges of the triangle and for the span interpolation must be computed. In the framework of a hardware implemented rasterizer, this step initializes the rasterizer with the initial values defining the triangle and the increments necessary for interpolation. The initialisation is part of the graphics pipeline and must be done for every triangle being rendered.

The throughput of the rasterization pipeline depends on the granularity of the primitives being rendered. When processing very small or degenerated triangles, the span interpolation can not operate at its full speed because it is slowed down by the edge interpolation. By moving the setup on the rasterizer, we aim at increasing the throughput of the standard graphics pipeline at the cost of additional chip surface.

3 Scan conversion pipeline

To render triangles we will use a variation of Bresenham's incremental line drawing algorithm [5]. The chosen algorithm ensures that triangles which share an edge do not share any pixels and do not produce any dropouts or overlaps between adjacent polygons. In broad lines, the modified algorithm uses the edge's slope $\Delta x/\Delta y$ and initial error ϵ_0 (the horizontal distance between the edge and the pixel center) and updates an error term ϵ for each scanline.

During the edge walk phase, triangles are decomposed into horizontal spans and lines into pixels. Two iterators are used to compute the beginning and end x-locations of a span and six other iterators to compute the $RGB\alpha ZY$ starting values for the first pixel on a

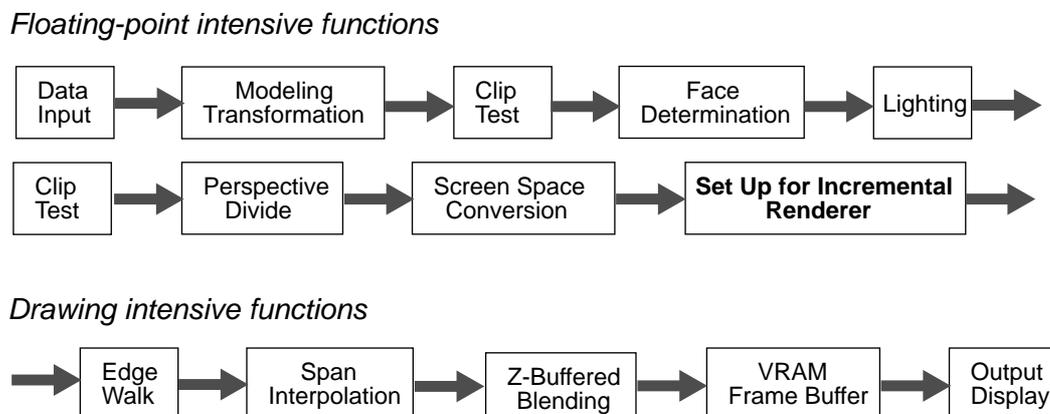


Figure 1. The standard 3D graphics pipeline.

span. The edge extending from the vertex with the maximum y-coordinate value to the vertex with the minimum y will be denoted as the leading edge. The edge running from the vertex with minimum y to the vertex with the middle y value is called the first trailing edge. The edge between the vertex with the middle y and the vertex with the maximum y value is called the second trailing edge. Edge processing starts by iterating down in parallel the leading and trailing edge. When the vertex with middle y is crossed, first and second trailing edges get swapped and the edge processing continues down the edge with the maximum y coordinate.

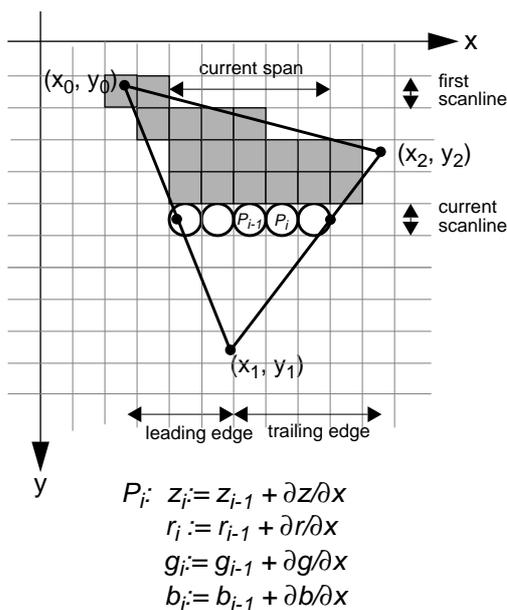


Figure 2. Triangle decomposition into spans and pixels.

In figure 4 we model a typical raster pipeline. The measure of granularity of pipelining is chosen to be bounded by the latency of a 32-bit full adder and the clock frequency is set to 80 MHz. Any subsequent data flow decomposition is scaled to this measure.

An edge processor decomposes triangles into horizontal spans and lines into pixels. Spans are further decomposed into pixels by a span processor. Two successive iterators are needed for computing the beginning and end x-locations of the span, and six iterators to compute the RGBαZY for the first pixel on the span. The edge walk is able to generate a new pixel every clock cycle. For processing the spans, six iterators scan the pixels on a span and generate the RGBαXZ values of each pixel. Assuming Gouraud shading and z-buffering, the span processor will generate one pixel per cycle in the x-direction. Schematically a span iterator consists of 3 stages of logic: a two-input adder, two multiplexers (equivalent to three 2-to-1 multiplexers) and two registers. Thus the span

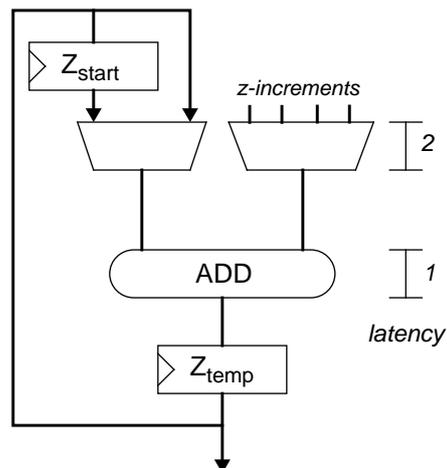


Figure 3. A span iterator.

processor has a pipeline latency of 3 cycles.

Before pixel data can be written to the color buffer, the newly generated colors may be blended with an already existing color in the color buffer. Blending requires readback from the color buffer. Since blending requires a read-modify-write to the frame buffer, it only is possible at half the fill performance of direct color write performance. We assume three cycles for the read and write operations to the color buffer, four cycles for multiplying the 8-bit RGB components in parallel and three cycles for blending the components with the illumination coefficients from the Phong shading engine or a previous color taken from the FIFO. If the frame buffer is driven by a raster pipeline clocked at 80 MHz, it must be five times interleaved, so that color filling along a span happens contentionless.

The z-buffer operation consists of reading back the old z-value, comparing it with the new z-value, and if the comparison succeeds, the new z-value and color value are written to their respective buffers. Since the z-buffer requires two accesses (read and write) for every write to the color buffer, the memory for the z-buffer shall be twice interleaved as the color buffer to accommodate the z-buffer update at the color Gouraud shaded fill rate.

Synchronous DRAMs with a bandwidth five times wider than the bandwidth of single ported DRAMs are right becoming available. Still, operations like color blending or the z-buffer compare do require a read-modify-write to the memory, resulting in a sequential two-way data exchange between the rasterizer and memory, which is one limiting factor to real-time performance. Converting the read-modify-write of the z-value and the RGBα color blend into a single write operation is the solution proposed with the FBRAM [2] and offers a ten times higher bandwidth than standard 60ns VRAM.

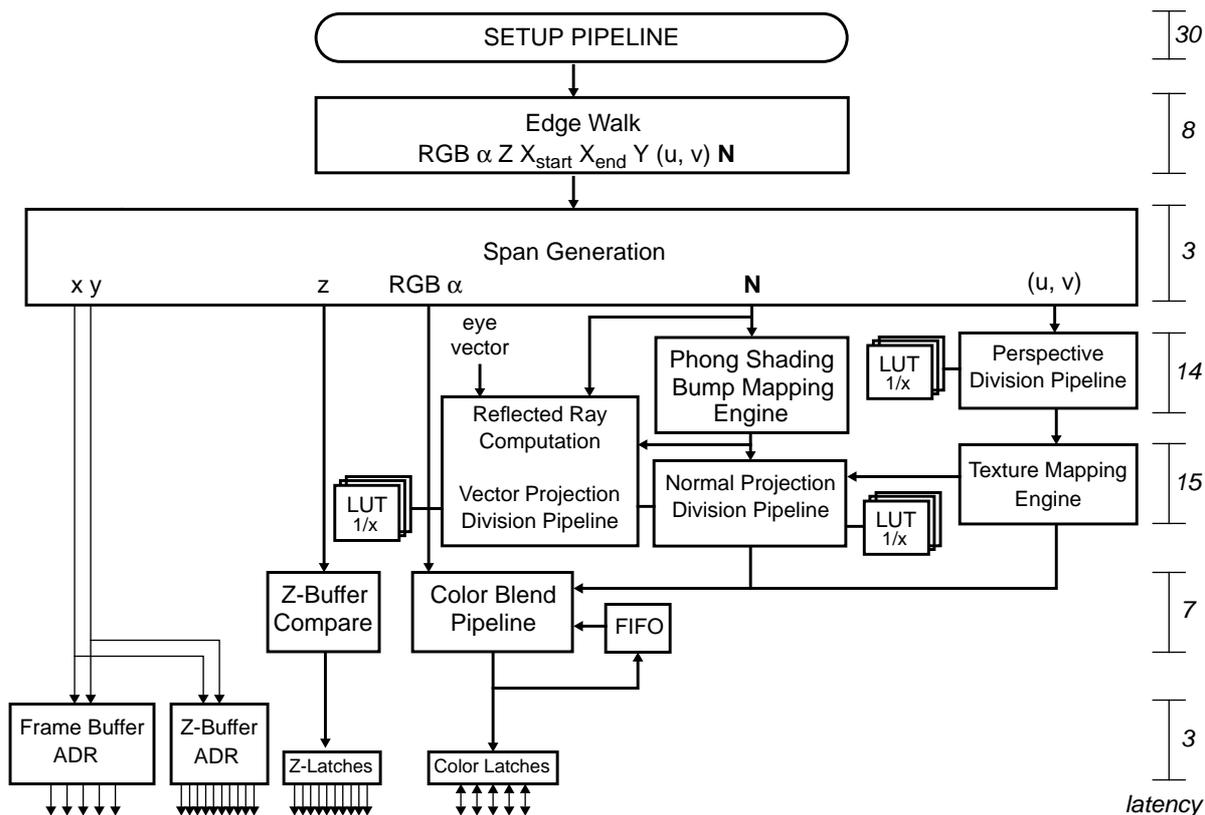


Figure 4. The raster pipeline.

Phong shading and texture mapping requires the triangle vertex normals N and (u, v) texture coordinates to be interpolated. Because the reflection vector is based on the changing eye vector, we expect it to be computed by the Phong shading engine. The reflection vector can be calculated from unnormalized surface normal and eye vectors [6]. Normal and reflection vectors are characterized by their horizontal and vertical angles which are used to index specular and diffuse maps [6, 7, 8]. Computing these angles involves division of the vector's minor axis values by the major axis values. To eliminate the distortions due to perspective projection of textures a division is necessary per rendered pixel. Accurate perspective division at pixel rate requires costly division operations for each pixel, and therefore approximation techniques will be employed [9]. The texture mapping pipeline has a latency of 15 cycles: 5 cycles for the address generation, 3 cycles for the texture memory controller and 7 cycles to trilinearly interpolate the textured pixels. Computing the reflected ray vector has a latency of 10 pipeline stages, followed by 14 stages for projecting the vector and 5 stages for specular and diffuse shading coefficients lookup.

4 Triangle Meshes

Triangles can be generalized to triangle meshes. A

mesh given by N vertices is rendered by sending the vertices down the graphics pipeline to the setup stage, where they are pushed on a stack to be popped later when no longer needed.

OpenGL provides the functionalities for mesh rendering [10]. A triangle mesh is rendered using the GL graphics library by sending a sequence of vertices through the graphics pipeline. This sequence is usually encapsulated in a display-list, recognized by the rendering hardware, and a triangle is automatically drawn between every three consecutive vertices of the sequence. A sequence of N vertices specifies $N-2$ triangles. Because at least one vertex must be supplied to render each triangle, ideally only the data for one vertex is sent through the graphics pipeline.

The time cost of the rendering stage is proportional to the number of vertices sent down the graphics pipeline and bounded by the latency of the setup computations for each pair of adjacent vertices. Triangles may either come as simple *strip* like meshes, or as *star* like meshes, in which cases N vertices specify $N-2$ triangles. In these two cases, triangles can be scanned by a non-intersecting path, or Hamiltonian path. In the third case of triangle meshes, a mesh is specified as a chain of K vertex sequences defining separate isolated paths. Here the time cost for N triangles is proportional to $2K+N$. It can be shown [11] that any N -vertex triangle mesh can be rendered by sending each

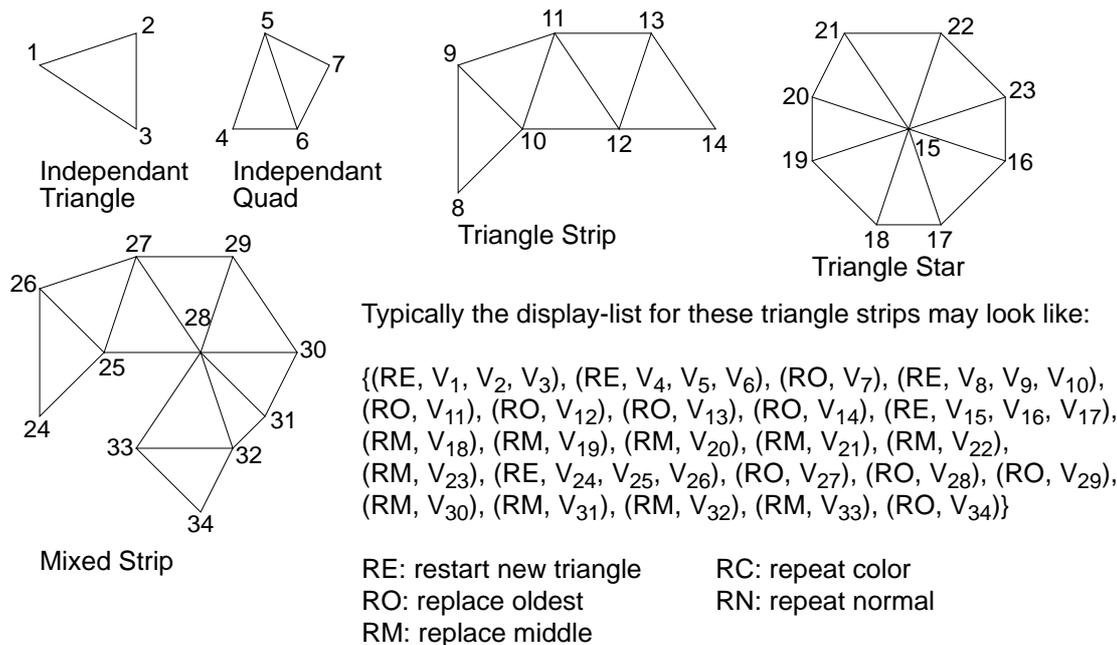


Figure 5. Triangle strips.

vertex only once down the graphics pipeline in minimum time (time cost proportional to N) by buffering the vertices in a stack of size $\Theta(\sqrt{N})$. Some triangle meshes may have a topology for which the stack size is considerably less.

To be able to handle triangle strips, information on the connexion and order between vertices is associated with each vertex. Bits (*replace oldest, replace middle, restart new triangle, repeat color, repeat normal*) in the vertex header within a strip specify how the incoming vertex is combined with the previous three vertices to form the next triangle. Before scan conversion starts, the incoming vertices are sorted with the vertices of the previous triangle to determine the leading (left) edge and trailing (right) edge which enclose the spans of the current triangle.

The ability of a generalized triangle strip to effectively change from a *strip* to a *star* mode in the middle of a strip as shown in figure 5 allows the geometry to be represented compactly and requires less input data bandwidth.

5 Geometry Compression

Geometry compression [12] is a good alternative to reduce the triangle vertex input data width, if the data is loaded from a media in compressed form. Hardware decompression of the vertex data is possible at high rate and increases the input bandwidth to the setup stage.

Normal compression takes advantage of the symmetries in the unit sphere. Normals are encoded as following: the first three bits specify the octant, the

next three bits the sextant and finally two 6-bit fields specify two spherical coordinates [12]. Successful compression of vertex normals from 96 down to 18 bits can be done by simple table look-up in software, so normals can be transferred compressed to the setup stage, where on-the-fly decompression is done by a dedicated decompression circuitry. Decompressed normals have 16-bit components, including one sign and one guard bit. Another benefit of this representation is in the design of a Phong shading and environment mapping engine with datapaths for compressed vectors. We are currently investigating this.

6 Initialisation Data

Like in any digital interpolator, the number of fractional bits of precision is chosen such that the accumulated error over interpolation is invisible in most cases (Table 1). The needed resolution is determined by the resolution of the delta terms (Δx , Δy) which require one more bit than the number of bits in the X or Y range and by the number of bits wished for sub-pixel positioning. For a display with a 1280x1024 pixel resolution 12 bits of fraction are enough (16 bits when using 4 bits for sub-pixel positioning). The number of bits of fraction for the color and depth increments is chosen such that the error accumulated by interpolation along the longest line in a 1280x1024 raster is unnoticeable. If the software for the geometry processing computes these increments, it will use the IEEE floating point format to send the increments to the rasterizer.

Since ϵ is a measure of the distance between the

TRIANGLE DATA	32-bit words	TOTAL
<i>Vertices:</i>		
$x_0, x_1, x_2, y_0, y_1, y_2, z_0, z_1, z_2$	9 words	
<i>Colors:</i>		
$r_0, g_0, b_0, r_1, g_1, b_1, r_2, g_2, b_2, \alpha_0, \alpha_1, \alpha_2$	3 words	
<i>Normals and texture coordinates</i>		
$\mathbf{N}_0, \mathbf{N}_1, \mathbf{N}_2, (u_0, v_0), (u_1, v_1), (u_2, v_2)$	6 words	18 words
SETUP PARAMETERS		
<i>information on vertex connexion and replacement:</i>	1 word	
<i>edge interpolation increments:</i>		
$\Delta x/\Delta y$ (edge 1, 2, 3)	3 words	
<i>color, depth, texture, normal interpolation increments:</i>		
$\partial r/\partial x, \partial g/\partial x, \partial b/\partial x, \partial r/\partial y, \partial g/\partial y, \partial b/\partial y, \partial \alpha/\partial x, \partial \alpha/\partial y$	8 words	
$\partial z/\partial x, \partial z/\partial y, \partial u/\partial x, \partial v/\partial y, \nabla \mathbf{N}(x), \nabla \mathbf{N}(y)$	10 words	22 words
TOTAL		40 words

Table 1: Triangle Setup Data.

current pixel center and the triangle edge, the value of ϵ in conjunction with the edge slope $m = \Delta x/\Delta y$ can be used to calculate subpixel information and look up a coverage mask for optional antialiasing edges. Antialiasing only makes sense when using appropriate complementary subpixel masks for the pixels on edges of adjacent triangles [13].

7 Stereoscopic Rasterization

To properly perceive depth, our eyes perceive the world from a slightly different perspective. In a stereoscopic system [14], the scene is separately rendered from two different viewpoints, one for each eye and the work for the geometry transformations is doubled, as each vertex must be transformed twice. In previous work [15], the secondary view is obtained by transforming the pixels from the primary view, and the objects visible to both the left and the right eye no longer need to be rendered twice. This approach will undeniably produce holes in the secondary image after the transformation due to either image expansion or object visibility change. Holes appear at pixel locations invisible to the left eye, but visible to the right eye. There is no color information for the invisible pixels of one view, so the other view is not properly shaded. Holes also appear when zooming in a polygon: the visible area of the polygon is expanded in one view, but incorrectly expanded in the other view.

An efficient algorithm for filling the holes that may arise in the secondary view after transformation of

each pixel in the primary view was described in [16] and fills holes maintained in a linked list by interpolating the boundary pixels around the hole. This method can be efficiently implemented in software, but is not amenable to a special purpose and adequate hardware solution. The architecture proposed by [15] is approximate. Though it certainly is a cost-effective solution, because the raster pipeline is not duplicated for the secondary view, it may produce erroneous results.

Instead, we propose to generate the stereo view after the perspective projection, by applying a simple shear operation in screen coordinates on the triangle vertices of the perspective view [17]. Such a shear operation can be done by one common setup stage shared by two rendering pipelines that run in parallel. To support alternative rendering modes, an additional x-coordinate interpolator is necessary in the raster pipeline that generates the secondary view.

Projected vertices of polygons share y-coordinates but have different corresponding x-coordinates. The width of polygon pairs differs in each view, and therefore interior pixel information must be interpolated twice, once for each view.

Antialiasing can also be applied to stereoscopic pairs and requires the pixel coverage value to be calculated individually for each view, since the polygons of the primary and secondary views have different pixel coverage and geometrical characteristics.

Given the coordinate (x_p, y_p, z_p) of a point P in space, $(x_{proj}, y_{proj}, z_{proj})$ its perspective transformation and $(x_{sl}, y_{sl}), (x_{sr}, y_{sr})$ the coordinates of the left and right stereoscopic view.

$$x_{proj} = \frac{x_p}{\frac{z_p}{d} + 1} \quad y_{proj} = \frac{y_p}{\frac{z_p}{d} + 1} \quad z_{proj} = \frac{z_p}{\frac{z_p}{d} + 1}$$

$$x_{sl} = \frac{x_p - z_p \frac{e}{2d}}{1 + \frac{z_p}{d}} = x_{proj} - z_{proj} \frac{e}{2d}$$

$$x_{sr} = \frac{x_p - z_p \frac{e}{2d}}{1 + \frac{z_p}{d}} = x_{proj} + z_{proj} \frac{e}{2d}$$

$$y_s = \frac{y_p}{1 + \frac{z_p}{d}}$$

The coordinates of the stereoscopic views can be obtained by applying a shear operation on the projective coordinates.

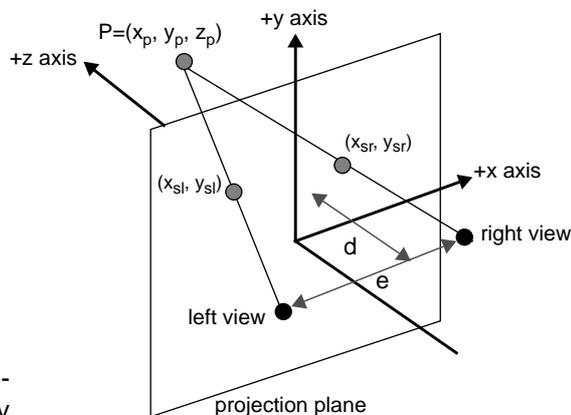


Figure 6. Shear operation applied to projective vertex coordinates.

8 Latency Intensive Computations

Multiplication is an operation which can be easily parallelized by breaking the multiplication of two values in two, or four individual multiplications and by summing the partial results. Multiplying two 16-bit numbers together has a pipeline latency of 4 cycles, if several adds are performed in parallel at each stage.

Division presents a coarser problem. The most commonly used division algorithms in modern FPUs are the subtractive and multiplicative methods. Among the subtractive algorithms, digit recurrence algorithms use subtraction as the iterative operator. This class of algorithms can be further separated into restoring and non restoring division. Restoring division is similar to the traditional paper and pencil method. The division of two N-bit numbers converges linearly and requires up to $2N+1$ adds. Non restoring division eliminates the restoring cycles and example algorithms can be found in [18]. It progresses by trial and error using the following relation:

$$P_{j+1} = r.P_j - q_{j+1} \cdot D$$

To calculate the next partial remainder P_{j+1} , the divisor D is multiplied by the next quotient digit, and the result is subtracted from the product of the last partial remainder, or dividend for the first iteration, and a radix r. Pipelining divisions in a long pipeline comes at the cost of replicating an N-bit adder in each stage. In each pipeline stage, the quotient is expanded with one bit of precision and N bits of significance will require N pipeline stages.

To reduce the high latency of linearly convergent

division, other iterative schemes were developed [19, 20]. These are based on series expansion of the reciprocal. For example, Newton-Raphson implementations can achieve very low latency, as Newton-Raphson division converges to a result quadratically. This performance comes at the price of additional hardware, accuracy and complexity for storing the lookup table containing the initial estimates. Comparing the latency bounds of these algorithms when applied to the IEEE double precision 64-bit data type, restoring division requires 428 gates, non-restoring 212, and Newton-Raphson 109 gates [19].

The Newton-Raphson iterates in the following way: it finds an approximation to the reciprocal $1/b$ and multiplies this to calculate the quotient. Each iteration involves two multiplications that cannot be performed simultaneously and one subtraction:

$$x_{i+1} = x_i \cdot (2 - b \cdot x_i)$$

One iteration can be split into one table look-up (1 cycle), two multiplies (4 pipeline stages - we assume the presence of a pipelined multiplier and adder) and one subtraction (1 pipeline stage), so evaluating $q = 1/b$ in one iteration has a total latency of 10 pipeline stages.

The performance of iterative algorithms generally depends on the initial approximation for the reciprocal, taken from a ROM look-up table (LUT). Such a LUT is generally designed for normalized arguments $1 \leq x < 2$ and truncated to k bits to the right of the radix point, $trunc(x) = 1.x_1x_2...x_k$. These k bits are used to index a table providing m output bits which form the m bits after the leading bit in the m+1-bit

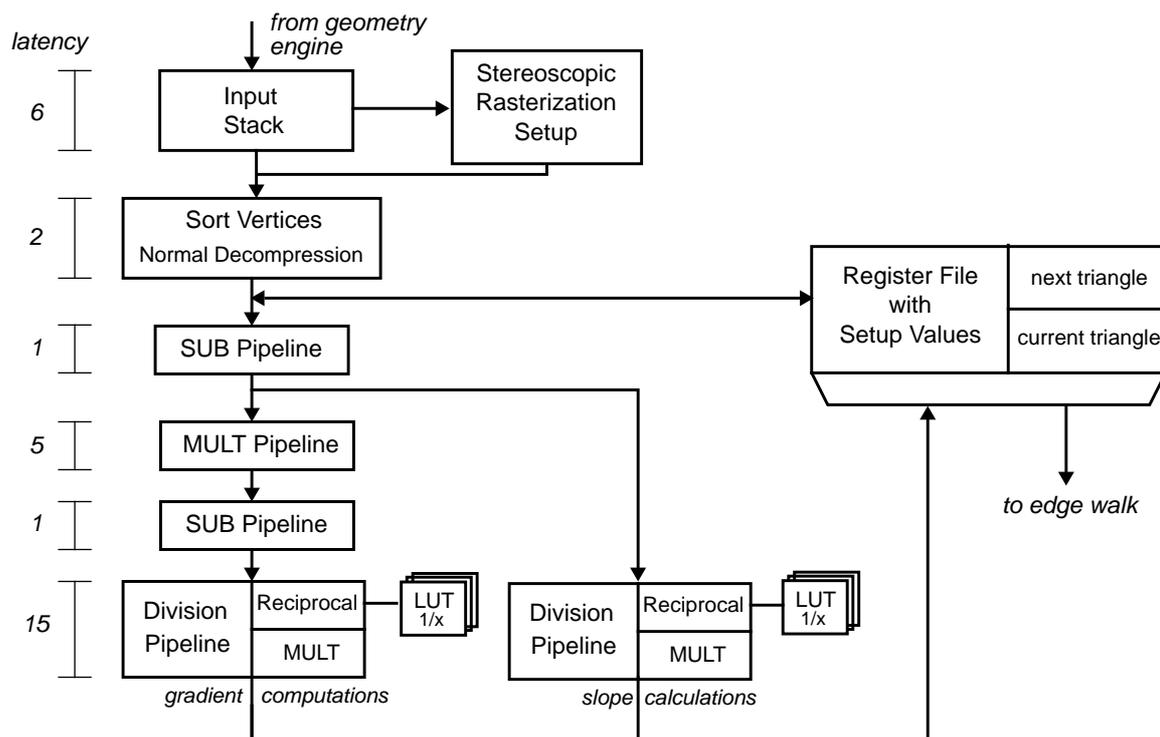


Figure 7. The Setup Pipeline.

fraction reciprocal approximation of $x: 0.1q_1q_2...q_m$. Since the leading bit of x is known to be 1 after normalization, it is not actually useful to use that bit as part of the index. When increasing the precision by one bit, the size of the approximation LUT is immediately doubled. Therefore, a better form of reciprocal table [21, 22] is constructed with k -bits-in \times ($k+g$ -bits-out), where g is the number of guard digits in the input. The size of this table is $2^k \times (k+g)$ instead of $2^k \times m$ bits. Adding guard bits does provide sufficient additional accuracy in place of the more costly step of increasing k to $k+1$, which results in more than doubling the LUT size. If the initial approximation is to be refined by one Newton-Raphson iteration as part of a division process, the resulting doubling of precision in each iteration then provides that a three guard bit initial enhancement implicitly contributes approximately two more bits of precision to the final reciprocal evaluation.

In order to compute 12 bits of reciprocal precision in one Newton-Raphson iteration (or 10 cycles), the reciprocal ROM lookup table must provide at least $2^5 \times (5+3)$ or 256 bits. For 16-bit reciprocal precision, a $2^8 \times 8$ or 2048 bits LUT would normally be used, but a smaller $2^7 \times (7+3)$ or 1280 bits LUT does give enough accuracy, when making one Newton-Raphson iteration on the initial approximation.

Five independent reciprocal lookup tables are necessary to fully pipeline the computations of the slopes, gradient calculations, perspective texture coordinate divisions, Phong illumination coefficients and bump

mapping parameters. The use of reciprocal tables with guard bits [21] reveals to be a practical means of lowering the latency of complex computations and reducing the size of a rasterizer implementation. Such lookup tables occupy about 30% less space than the starting reciprocal approximation tables usually employed for Newton-Raphson iterations.

9 Setup Architecture and Bandwidth

To synchronize the input stage of the rasterizer with the geometry unit, the setup data is buffered in a stack. Special control words describing the connectivity between vertices separate different series of vertices and are decoded on the fly. Current and future vertices are dropped in a stack. Vertex data is represented compactly as explained in section 4 to limit data redundancy and reduce data transfer bandwidth. An internal state machine transfers the values from the stack to the registers of the rasterizer. The registers are part of a pipeline in which the setup parameters are calculated. Figure 7 shows the data flow in the setup pipeline.

The vertex and color data for the current triangle goes down the setup pipeline, where edge, color, depth, texture coordinate and normal vector increments and starting values for every span are computed, before being stored in the registers of the rasterizer. A double-sized register file is used for the vertex and setup values. One half of the registers is used for rendering the current triangle, while the val-

ues for the next triangle can already be prepared.

Color, depth and texture coordinate increments are all computed from the same reciprocal evaluation:

$$\frac{\partial z}{\partial x} = \frac{(z_2 - z_1) \cdot (y_3 - y_1) - (z_3 - z_1) \cdot (y_2 - y_1)}{(x_2 - x_1) \cdot (y_3 - y_1) - (x_3 - x_1) \cdot (y_2 - y_1)}$$

$$\frac{\partial z}{\partial y} = \frac{(z_3 - z_1) \cdot (x_2 - x_1) - (z_2 - z_1) \cdot (x_3 - x_1)}{(x_2 - x_1) \cdot (y_3 - y_1) - (x_3 - x_1) \cdot (y_2 - y_1)}$$

The Δ -terms (vertex coordinate differences $x_i - x_j$, $y_i - y_j$) are operands shared between the gradients and therefore gradients should be calculated in parallel. Gradients are computed in individual *SUB-MULT-SUB* pipelines running in parallel and sharing the registers containing the Δ -terms. The last step involves multiplication of each gradient by the same reciprocal. Any new triangle specified by one or three new vertices requires recomputation of the gradients.

The slopes $\Delta x / \Delta y$ are computed by a secondary division pipeline running in parallel with the gradient calculation pipeline.

Since a shear operation applied to a triangle of the primary view in a stereoscopic display is invariant on the area of the triangles projected on the zx - and zy -planes, the same numerators ($\det[Z, X]$, $\det[Z, Y]$) can be taken for calculating the depth, color and texture coordinate increments in the setup for stereoscopic rasterization. Only the reciprocal ($1/\det[X, Y]$) must be computed individually for each view.

To model the throughput of our setup pipeline, we will make following considerations: the setup and raster pipeline are clocked at 80 MHz, producing one rendered pixel per cycle. Standard VRAM and DRAM are used for the frame buffer and z-buffer. To achieve maximum performance in page mode access with a memory controller operating at 80 MHz, the frame buffer must be five times interleaved. As already mentioned, the z-buffer has a 10-way interleaving.

For isolated triangles, the geometry processor will have to pass 18 words of 32-bit data to the setup. Tri-

angle meshes only require one new vertex for each new triangle or 6 words to be transferred to the setup only. In a configuration without integrated setup calculation, the geometry processor must supply 40 words of setup data to the rasterizer for any new triangle. Table 2 shows the amount of data transferred to the rasterizer and the necessary input bandwidth for different configurations. If we add up the latencies of the individual processing stages, we get a latency of 50 stages for the raster pipeline and a latency of 30 stages for the setup pipeline, giving a total latency of 80 pipeline stages. If the rasterizer and setup are able to handle one million triangles per second (typical triangles are between 50 and 100 pixels large), then the geometry processor must transfer 16 Mbytes/s of data to the rasterizer. This is well within the capabilities of current PCI chipsets (The Intel Triton offers 80 MBytes/s of PCI bandwidth) and relaxes well the PCI bandwidth between the geometry processor and rasterizer.

Triangles with a size $S > 30$ pixels are likely to get rendered in S cycles. The raster pipeline will start rendering the last pixel S cycles after the first pixel of the triangle. During this period, the setup data values for the next triangle are computed and the old contents of the register file are swapped with new values after S cycles, in order to have a continuous flow of pixels at the output. For $S > 30$, the geometry engine can transfer triangle vertices to the setup at the same rate as the triangle output rate of the rasterizer. Small triangles with $S < 30$ are processed at a lower rate, in the sense that the setup does not offer sufficient bandwidth to handle triangle vertices at the same rate as the output rate of the rendered triangles.

The texture mapping stage between the span generator and the color blend stage is a performance bottleneck. A texture mapping engine operating at the same speed as the rasterizer can hardly deliver one trilinear interpolated textured pixel per cycle, when texture data is retrieved from standard DRAM [23, 24].

Division performance can be increased by using a reciprocal cache [25] that contains results from previous reciprocal evaluations for later reuse, instead of

SETUP DONE BY	Gouraud Shaded: 1M triangles/s		Texture Mapped and Blended: 500k triangles/s	
	Data Amount	Input Bandwidth	Data Amount	Input Bandwidth
GEOMETRY ENGINE <i>isolated triangles</i>	34 words	136 MBytes/s	40 words	80 MBytes/s
RASTERIZER <i>triangle meshes</i>	4 words	16 MBytes/s	6 words	12 MBytes/s
RASTERIZER <i>isolated triangles</i>	12 words	48 MBytes/s	18 words	36 MBytes/s

Table 2: Necessary Input Bandwidth to the Rasterizer.

increasing the size of the initial approximation reciprocal table.

Performance at interactive rendering rates for texture mapped triangles comes at the cost of keeping the latencies of the division stage and texture mapping engine as small as possible. If the output bandwidth to frame buffer and z-buffer is sufficiently high and no bandwidth degradation is on the input to the setup and at the output from the texture mapping engine, performance virtually becomes limited by the video output rate of the frame buffer.

10 Conclusion

Performance of a graphics processor is affected by its architecture and the available bandwidth on the interfaces to the geometry processor and the external memories. In the design of a high-speed graphics rendering pipeline, we are faced with several bottlenecks. One dominant time cost in rendering triangles is within the geometry processor, when computing the slope, color, depth and texture increments for every triangle and sending them to the rasterizer.

In this paper, we have described an architecture for a complete raster pipeline including a setup stage operating at the same rate as the rasterizer. Moving the setup from the geometry processor to the rasterizer greatly improves the available input bandwidth to the rasterizer and virtually permits to send triangles to the setup at the same rate as they can be written into the frame buffer by the rasterizer. To limit data redundancy, the setup is able to handle generalized triangle strips and stores triangle vertex values in a stack for later reuse.

We showed how to combine the setup engine and rasterizer into an evenly balanced pipeline, where high latency computations such as multiplication and division are efficiently implemented along the raster pipeline through hyperpipelining and the use of small reciprocal tables.

11 Acknowledgements

This work was done under the supervision of Professor Straßer, within the ESPRIT project MONO-GRAPH funded by the CEC. I would like to thank Andreas Schilling from the Computer Graphics Lab at the University of Tübingen for his input and comments on many issues related to triangle rasterization.

12 References

- [1] M. Deering, S. Nelson, *Leo: A System for Cost Effective 3D Shaded Graphics*, ACM SIGGRAPH Proceedings, 101-108, 1993.
- [2] M. Deering, S. Schlapp, M. Lavelle, *FBRAM: A new Form of Memory Optimized for 3D Graphics*, ACM SIGGRAPH Proceedings, 167-174, 1994.

- [3] G. Knittel, A. Schilling, W. Straßer, *GRAMMY: High Performance Graphics Using Graphics Memories*, In: High Performance Computing for Computer Graphics and Visualisation, Springer, London, 1996.
- [4] J. Foley, A. van Dam, S. Feiner, J. Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley, 1990.
- [5] O. Lathrop, D. Kirk, D. Voorhies, *Accurate Rendering by Subpixel Addressing*, IEEE Computer Graphics and Applications, 45-52, September 1990.
- [6] D. Voorhies, J. Foran, *Reflection Vector Shading Hardware*, ACM SIGGRAPH Proceedings, 163-166, 1994.
- [7] D. Jackël, H. Rüsseler, *A Real Time Rendering System with Normal Vector Shading*, Proceedings of the 9th Eurographics Workshop on Graphics Hardware, 48-57, 1994.
- [8] I. Ernst, D. Jackël, H. Rüsseler, O. Wittig, *Hardware Supported Bump Mapping: A Step towards Higher Quality Real-Time Rendering*, Proceedings of the 10th Eurographics Workshop on Graphics Hardware, 57-52, 1995.
- [9] M. Flynn, *On Division by Functional Iteration*, IEEE Transactions on Computers, Vol. C-19, No. 8, 702-706, 1970.
- [10] J. Neider, T. Davis, M. Woo, *OpenGL Programming Guide*, Addison-Wesley, 1993.
- [11] R. Bar-Yehuda, C. Gotsman, *Time/Space Tradeoffs for Polygon Mesh Rendering*, ACM Transactions on Graphics, Vol. 15, No. 2, 141-152, 1996.
- [12] M. Deering, *Geometry Compression*, ACM SIGGRAPH Proceedings, 1995.
- [13] A. Schilling, *A New Simple and Efficient Antialiasing with Subpixel Masks*, Computer Graphics, Vol. 25, No. 4, 133-141, 1991.
- [14] L. Hodges, *Tutorial: Time-Multiplexed Stereoscopic Computer Graphics*, IEEE Computer Graphics and Applications, 20-30, March 1992.
- [15] S. McCann, G. Dunnett, S. Pearce, M. White, M. Waller, P. Lister, *An Architecture for Rapid Stereoscopic Image Generation*, Proceedings of the 10th Eurographics Workshop on Graphics Hardware, 57-52, 1995.
- [16] S. Fu, H. Bao, Q. Peng, *An Accelerated Rendering Algorithm for Stereoscopic Display*, Computers & Graphics, Vol. 20, No. 2, 155-160, 1996.
- [17] A. Schilling, *Rasterizer Setup for Stereo Rendering*, Technical Report WSI-96-23, ISSN 0946-3852, August 1996.
- [18] M. Ercegovac, T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations*, Kluwer Academic Publishers, 1994.
- [19] S. Oberman, M. Flynn, *Design Issues in Floating-Point Division*, Stanford Technical Report CSL-TR-94-647, December 1994.
- [20] Ch. Narayanaswami, W. Luken, B.-O. Schneider, *Efficient Evaluation of $1/x$ for Texture Mapping*, IBM Technical Note, May 1995.
- [21] D. DasSarma, D. Matula, *Measuring the Accuracy of ROM Reciprocal Tables*, IEEE Transactions on Computers, Vol. 43, No. 8, 932-940, 1994.
- [22] D. Wong, M. Flynn, *Fast division using accurate quotient approximations to reduce the number of iterations*, IEEE Transactions on Computers, Vol. 41, No. 8, 981-995, 1992.
- [23] A. Schilling, G. Knittel, W. Straßer, *TEXRAM - A Smart Memory for Texturing*, IEEE Computer Graphics and Applications, Vol. 16, No. 3, 32-41, 1996.
- [24] G. Knittel, A. Schilling, A. Kugler, W. Straßer, *Hardware for Superior Texture Performance*, Proceedings of the 10th Eurographics Workshop on Graphics Hardware, 33-39, 1995.
- [25] S. Oberman, M. Flynn, *Reducing Division Latency with Reciprocal Caches*, Journal of Reliable Computing, Vol. 2, No. 2, 1996.