Incremental and Hierarchical Hilbert Order Edge Equation Polygon Rasterization

Michael D. McCool Chris Wales Kevin Moule

Computer Graphics Lab
Department of Computer Science
University of Waterloo
http://www.cgl.uwaterloo.ca/Projects/rendering/

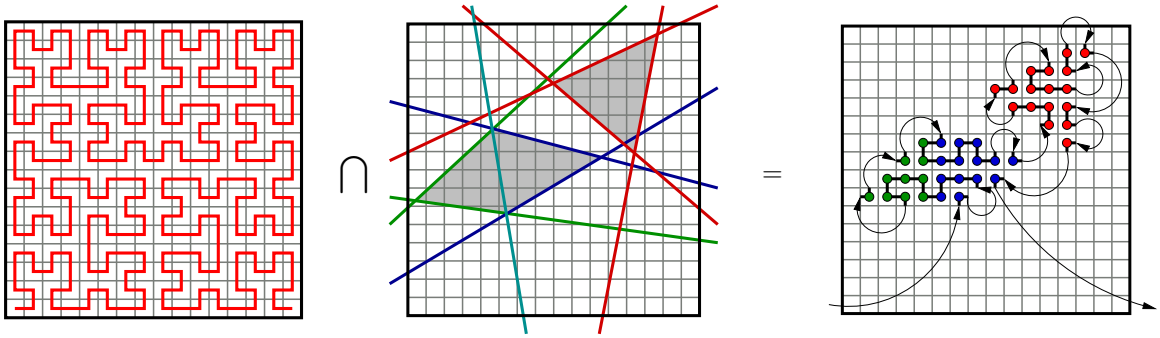

Figure 1: *The rasterization technique presented here can generate pixel fragments for a set of polygonal primitives (described by a set of homogeneous edge equations) in the same order that would result from travelling along a Hilbert curve intersecting those primitives.*

Abstract

A rasterization algorithm must efficiently generate pixel fragments from geometric descriptions of primitives. In order to accomplish per-pixel shading, shading parameters must also be interpolated across the primitive in a perspective-correct manner. If some of these parameters are to be interpreted in later stages of the pipeline directly or indirectly as texture coordinates, then translating spatial and parametric coherence into temporal coherence will improve texture cache performance. Finally, if framebuffer access is also organized around cached blocks, then organizing rasterization so fragments are generated in block-sequential order will maximize framebuffer cache performance. Hilbert-order rasterization accomplishes these goals, and also permits efficient incremental evaluation of edge and interpolation equations.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture.

Keywords: Hardware accelerated image synthesis and shading.

HWWS '01 Los Angeles, CA USA

1 Introduction

Rasterizing polygons using edge equation evaluation has been extensively investigated [4, 8, 9, 11]. In this approach, linear equations are first set up for each edge of a polygonal primitive. The fragments of the rasterized polygon are defined as the fragments for which all equations have a common sign (we will assume a positive sign defines the interior halfspace of an edge equation in this paper). Conceptually, an edge-equation based rasterizer simply tests the positions of all potential fragments against the edge equations for each polygon.

The advantages of this approach are numerous. Edge equations can be set up directly from the homogeneous coordinates of the transformed device-space vertices of polygonal primitives using a regular and parallelizable computation, and without performing any divisions. Special cases—the plane of the polygon crossing or behind the eye, backfacing or not—can be handled correctly and easily. Clipping planes and viewport edge tests can be incorporated into the same algorithm by setting up additional equations. A pure edge equation rasterizer can also easily be extended to the rasterization of an entire collection of polygons in parallel. Finally, evaluation of basis functions for the interpolation of parameters can exploit the same hardware resources used to evaluate the edge equations.

However, the advantages of the edge-equation approach are countered by the need to scan the area of primitives efficiently. Various approaches have been used, but to date most rasterization algorithms using edge equations involve performing at least one division to find a "starting" fragment for a single primitive, and unfortunately this cancels many of the above advantages. An exception is the approach taken by the PixelPlanes architecture [4] of actually

evaluating the edge equations in all pixels simultaneously using a SIMD array, but this is rather expensive. While simultaneous evaluation of equations in a "tile" may be used to improve performance, we typically do not want to test all fragments of the output buffer for all primitives.

Our approach organizes the scan for pixel fragments using a Hilbert curve. We actually move along the Hilbert curve in a hierarchical fashion, using an outcode approach to skip over blocks of potential fragments when all corners of such blocks are identified as being exterior to at least one edge of every triangle in the active set. The general idea of our approach is similar to the Warnock's recursive algorithm, or Greene's quadtree algorithm [5] except we order our visits to the children of each block in Hilbert-curve fashion. The utility of the Hilbert curve is that its strict spatial locality allows us to incrementally evaluate all edge equations. Furthermore, even though we are using a conceptually "recursive" algorithm, when using fixed-point arithmetic we can avoid the need for a large stack by exactly reversing our incremental updates when backing out of a recursive subdivision. The result is a relatively simple algorithm that requires minimal storage and implementation complexity but has excellent spatial coherence properties.

Space-filling curves [12] have been used before in computer graphics for dithering [14], for generalizing stratified sampling in multidimensional space (our work, as yet unpublished) and most relevantly, as a coherence-enhancing scan order for raytracing [13]. By using a Hilbert curve as a scan order, we gain several advantages. For the purposes of writing pixels to the framebuffer, the Hilbert curve ordering visits all pixel locations in any $2^n \times 2^n$ aligned block (for any n), before moving onto a new one. This means that if the framebuffer is organized into $2^n \times 2^n$ aligned blocks (for any integer n) and the blocks are cached, at least over the active polygon set we will make maximum use of each cache block before moving on. Secondly, the parameters interpolated by the rasterizer will also be generated in a spatially coherent order, which will improve locality in texture caches, even if nonlinear per-pixel shading computations are performed before texture lookup. The hierarchical nature of the Hilbert curve means that these benefits will accrue at all scales, so the rasterization algorithm does not need to be adjusted if the number of texels or pixels in a cache block changes (for instance, if different colour representations or precision modes pack different numbers of texels or pixels into a cache block).

We have implemented our rasterizer in support of a system that is investigating various high-performance hardware-accelerated rendering issues, including both pixel shading and geometry manipulation. We wanted a simple rasterizer that would use a small number of gates, since our prototyping system uses a field-programmable gate array (FPGA) with a relatively limited number of gates and we want to implement other subsystems on the same chip. Also, since we are investigating programmable per-pixel shading, our rasterization algorithm is designed to efficiently support the interpolation of a large number of parameters at high precision over each primitive. Rather than interpolate parameters directly, our algorithm evaluates projectively corrected basis functions first and then uses these to blend parameters stored at the vertices of each triangle.

In the following, first the background and setup computations necessary for the edge-equation based approach are reviewed in Section 2. We present the computations needed to set up equations for triangle edges, interpolation basis functions, and viewport edges. We also discuss how to effectively convert from floating-point values computed during setup to the fixed-point values used by the rasterizer, while preserving precision and range. In Section 3 we discuss the Hilbert curve and present an efficient state-machine technique for computing the Hilbert curve in hardware. Section 4 describes how the Hilbert curve and the set of potential fragments are scanned by our algorithm in an efficient hierarchical fashion. This section also describes how various quantities are updated incrementally. By using the reversibility of both the Hilbert curve generation algorithm and the incremental updates (when using exact fixed-point arithmetic), the hierarchical search can be organized with only a very small stack (only two bits per level of recursion, no more storage than needed to store the (x, y) locations of the fragments themselves). Finally, in Section 5 we describe our prototype hardware implementation.

2 Equation Setup

Polygonal primitives are specified as sequences of vertices that have been transformed by a 4×4 matrix representing a projective transformation into a device coordinate system. The edge equation approach is capable of rasterizing sets of triangles in parallel, but this section considers only individual triangles, each triangle given by three device-space homogeneous vertices $\mathbf{v}_0$, $\mathbf{v}_1$, and $\mathbf{v}_2$. Each homogeneous vertex has the form

$$\mathbf{v}_i = \begin{bmatrix} \{w_i x_i\} \\ \{w_i y_i\} \\ \{w_i z_i\} \\ w_i \end{bmatrix}, \tag{1}$$

where values surrounded by braces are scalars. The values $\{w_i x_i\}$, $\{w_i y_i\}$, and $\{w_i z_i\}$ are usually but not always generated from products of the indicated component values; however, for points at infinity these values may be non-zero while w_i must be zero.

The three homogeneous vertices defining each triangle will be converted into three edge equations, which can then (optionally) be combined with other equations defining clipping planes and viewport edges to define a visible polygonal region for that triangle. The visible region of each primitive is defined by the intersection of the positive half-spaces of all edge equations. Zero edge equation evaluations are treated with a tie-breaking rule based on the signs of the normal components of the edge equations. This avoids rasterizing fragments of adjacent primitives twice and/or leaving gaps between primitives. Basis functions for perspective-correct interpolation can be derived from the evaluation of the equations for the edges of each triangle.

2.1 Edge Equations

To convert homogeneous vertices to edge equations, first compute the adjoint of the "vertex matrix" formed from the 2D orthographic projections of all vertices:

$$\mathsf{M} = \begin{bmatrix} \{w_0 x_0\} & \{w_1 x_1\} & \{w_2 x_2\} \\ \{w_0 y_0\} & \{w_1 y_1\} & \{w_2 y_2\} \\ w_0 & w_1 & w_2 \end{bmatrix} \tag{2}$$

$$\mathsf{A} = \mathrm{adj}(\mathsf{M}) \tag{3}$$

$$= \begin{bmatrix} a_0 & b_0 & c_0 \\ a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \end{bmatrix}. \tag{4}$$

The computation of the adjoint is similar in complexity to the computation of three cross products, and requires only multiplication and subtraction in a regular and parallelizable pattern. The necessary equations are

$$a_0 = \{w_1 y_1\} w_2 - \{w_2 y_2\} w_1 \tag{5}$$

$$a_1 = \{w_2 y_2\} w_0 - \{w_0 y_0\} w_2 \tag{6}$$

$$a_2 = \{w_0 y_0\} w_1 - \{w_1 y_1\} w_0 \tag{7}$$

$$b_0 = \{w_2 x_2\} w_1 - \{w_1 x_1\} w_2 \tag{8}$$

$$b_1 = \{w_0x_0\}w_2 - \{w_2x_2\}w_0 \quad (9)$$
$$b_2 = \{w_1x_1\}w_0 - \{w_0x_0\}w_1 \quad (10)$$
$$c_0 = \{w_1x_1\}\{w_2y_2\} - \{w_2x_2\}\{w_1y_1\} \quad (11)$$
$$c_1 = \{w_2x_2\}\{w_0y_0\} - \{w_0x_0\}\{w_2y_2\} \quad (12)$$
$$c_2 = \{w_0x_0\}\{w_1y_1\} - \{w_1x_1\}\{w_0y_0\} \quad (13)$$

The adjoint is related to the inverse by a scale factor, specifically the determinant of the original matrix:

$$\mathsf{M}^{-1} = \frac{\mathrm{adj}(\mathsf{M})}{\det(\mathsf{M})} \quad (14)$$

We can ignore the magnitude of this scale factor since we are working in homogeneous coordinates. The sign of the determinant gives the orientation of the triangle: a counterclockwise order gives a positive determinant. Using only the adjoint, we will automatically cull backfacing triangles, i.e. those with a clockwise vertex ordering. If we wish to actually render back-facing triangles, we can compute the determinant and invert the signs of all elements of the adjoint if it is negative. The determinant can be computed using

$$\det(\mathsf{M}) = c_0w_0 + c_1w_1 + c_2w_2. \quad (15)$$

where the c_i are the appropriate elements of the adjoint matrix A.

From the adjoint we can find the edge equations for the triangle; they are simply the rows of the adjoint matrix:

$$\mathbf{E}_0 = [a_0, b_0, c_0], \quad (16)$$
$$\mathbf{E}_1 = [a_1, b_1, c_1], \quad (17)$$
$$\mathbf{E}_2 = [a_2, b_2, c_2], \quad (18)$$

where $\mathbf{E}_k$ represents the edge between the two vertices *opposite* $\mathbf{v}_k$. Given a fragment position (x, y), the value of edge equation $\mathbf{E}$ at (x, y) is evaluated using a linear combination:

$$\mathbf{E}(x, y) = [a, b, c][x, y, 1]^T \quad (19)$$
$$= ax + by + c. \quad (20)$$

Linear equation evaluation has the following two useful properties, which are exploited by our algorithm:

$$\mathbf{E}(0, 0) = c, \quad (21)$$
$$\mathbf{E}(x + s, y + t) = \mathbf{E}(x, y) + sa + tb, \quad (22)$$
$$= [a, b, c + sa + tb]\,[x, y, 1]^T. \quad (23)$$

The first property lets us evaluate equations at the origin with zero cost. The second lets us incrementally update an equation and its evaluation by shifting the origin. An update only modifies the c component of the equation.

The 2D vector (a, b) will be perpendicular to the edge and will point towards the interior halfspace; we will call this vector the *edge normal*. Pseudocode to test for a fragment being "inside" with respect to a given equation, including a tie-breaking rule based on (a, b), is given in Figure 2. The tie-breaking rule is based on the observation that for triangles of the same orientation sharing an edge, the shared edge is represented by equations whose coefficients are negations of one another. The above rules give semi-open triangles that are closed on their left sides, including any horizontal edge along the bottom, and open on their right sides, including any topmost horizontal side.

2.2 Parameters

Parameters will also be attached to the vertices of the triangle. Parameters may be world-space positions, texture coordinates, normal coordinates, tangent coordinates, or other user-defined values.

```
inside(E, x, y) {
  if (E(x, y) > 0) return True;
  if (E(x, y) < 0) return False;
  if (a > 0) return True;
  if (a < 0) return False;
  if (a = 0 & b < 0) return False;
  return True;
}
```

Figure 2: *Test for a particular (x, y) position being on the "inside" of a particular halfspace equation, with tie-breaking tests.*

Our target is a system with a programmable per-pixel shading unit [7] so we will be assuming a potentially large and variable number of parameters. For simplicity, we assume that all parameters and all components of multidimensional parameters will be interpolated across the primitive identically as independent scalars with perspective correction.

One approach to interpolating parameters [9] is to generate a linear equation for each per-vertex parameter triple and then substitute in the appropriate x and y values for the fragment. The coefficients $\mathbf{P}_k$ of the interpolation equation for the parameter triple $\mathbf{p}_k$ can be computed by postmultiplying each parameter's per-vertex values, placed in a row vector, by the inverse of the vertex matrix:

$$\mathbf{p}_k = [p_{k0}, p_{k1}, p_{k2}], \quad (24)$$
$$\mathbf{P}_k = \mathbf{p}_k\mathsf{M}^{-1}. \quad (25)$$

These coefficients provide linear parameter interpolation, which is only correct if all homogeneous scale factors are unity. In order for parameter interpolation to be correct under perspective, we have to divide by a common linear perspective correction factor. This factor can be computed by setting up an equation for the "reference" per-vertex parameter values $[1, 1, 1]$. We will denote this equation by $\mathbf{R}$:

$$\mathbf{R} = [1, 1, 1]\,\mathsf{M}^{-1}. \quad (26)$$

Because we will be dividing by the value given by the evaluation of $\mathbf{R}$, we can use the adjoint A in the above equations rather than M^{-1}, since common scale factors will cancel. Furthermore, when we have many parameters to interpolate, rather than computing a linear equation per parameter triple, we can compute a set of basis functions which can evaluated once and then used to create appropriate affine combinations of all parameter triples. To do this, we first set up a trio of pseudo-parameter triples, and then transform them as above:

$$\mathbf{F}_0 = [1, 0, 0]\mathsf{A} = \mathbf{E}_0, \quad (27)$$
$$\mathbf{F}_1 = [0, 1, 0]\mathsf{A} = \mathbf{E}_1, \quad (28)$$
$$\mathbf{F}_2 = [0, 0, 1]\mathsf{A} = \mathbf{E}_2. \quad (29)$$

No extra work is required to set up these equations, since the coefficients for each $\mathbf{F}_i$ are identical to the coefficients for the corresponding edge equation $\mathbf{E}_i$! Unfortunately, it turns out that we cannot represent these two equations in fixed point in exactly the same way, a point we discuss in Section 2.4, so we will use $\mathbf{F}_i$ instead of $\mathbf{E}_i$ for these equations. It is still necessary to compute the perspective correction factor. Fortunately this is also easy, since

$$\mathbf{R}' = [1, 1, 1]\mathsf{A} \quad (30)$$
$$= [1, 0, 0]\mathsf{A} + [0, 1, 0]\mathsf{A} + [0, 0, 1]\mathsf{A} \quad (31)$$
$$= \mathbf{F}_0 + \mathbf{F}_1 + \mathbf{F}_2, \quad (32)$$

and so

$$\mathbf{R}'(x,y) = \mathbf{F}_0(x,y) + \mathbf{F}_1(x,y) + \mathbf{F}_2(x,y) \quad (33)$$

The necessary rational basis functions, which sum to unity and so form an affine combination, are

$$\begin{aligned} r &= 1/(\mathbf{F}_0(x,y) + \mathbf{F}_1(x,y) + \mathbf{F}_2(x,y)), & (34)\\ f_0(x,y) &= r\mathbf{F}_0(x,y), & (35)\\ f_1(x,y) &= r\mathbf{F}_1(x,y), & (36)\\ f_2(x,y) &= r\mathbf{F}_2(x,y) & (37)\\ &= 1 - f_0(x,y) - f_1(x,y). & (38) \end{aligned}$$

which requires one reciprocation (or two divisions) to compute. After computing the basis functions, any parameter triple $\mathbf{p}_k = [p_{k0}, p_{k1}, p_{k2}]$ can be interpolated in a perspectively correct manner using

$$p_k(x,y) = p_{k0}f_0(x,y) + p_{k1}f_1(x,y) + p_{k2}f_2(x,y). \quad (39)$$

Depth values can be interpolated in the same manner as any other parameter, or alternatively, a separate linear equation can be defined and evaluated in the rasterizer. For simplicity our implementation takes the former approach.

2.3 Viewport Equations

Viewport clipping can be accomplished by considering four more equations during rasterization. The update and storage costs of these equations can be reduced slightly in an implemention because they have a simple form. A viewport with origin (x_0, y_0) and size (w, h) in device coordinates will result in the edge equations

$$\begin{aligned} \mathbf{d}_0 &= [1, 0, -x_0], & (40)\\ \mathbf{d}_1 &= [-1, 0, x_0 + w], & (41)\\ \mathbf{d}_2 &= [0, 1, -y_0], & (42)\\ \mathbf{d}_3 &= [0, -1, y_0 + h]. & (43) \end{aligned}$$

According to our tie-breaking conventions, viewports are inclusive on the bottom and left and exclusive on the top and right.

Three-dimensional user-specified clipping planes can also theoretically be supported. However, a different 2D edge equation must be generated for the intersection of each triangle with every clipping plane. To limit the number of equation evaluators in the rasterizer and reduce setup costs, we suggest implementing only full-triangle clipping during rasterization setup; fragment clipping can be performed after rasterization by evaluation of the 3D clipping equations in device coordinates.

2.4 Conversion to Fixed-Point

We perform rasterization setup operations using floating-point arithmetic. Rasterization itself uses only fixed-point arithmetic for speed and compactness, and because we must exactly reverse incremental updates.

As noted in Section 2.2, the edge equations and the interpolation equations are mathematically identical. However, in practice, they have to be converted to fixed point in slightly different ways, and so will require separate interpolators. The conversion of the edge equations must preserve *orientation precision*, but can scale each edge equation separately and clamp c values for edges outside of the visible region to accomplish this. For interpolation, clamping of the c coefficient would give the wrong answer, and likewise we cannot scale the equations independently without invalidating the perspective correction. Therefore, for interpolation equations we try to preserve their *range*, and consider them together.

The homogeneous coefficients of an edge equation can be scaled by an arbitrary amount without changing the geometric location of the edge. We also don't care about the exact position of edges that are outside the visible part of the device coordinate system, just whether they exclude or contain the visible fragments. We therefore convert edge equations to fixed-point form by finding the largest exponent of a and b and shifting the mantissas of all of a, b, and c by that amount, clamping c if it goes outside its representable range. After this normalization, both a and b can safely be represented using fixed-point values in the range $[-1, 1]$, and the length of the normalized edge normal will be bounded between 1 (if one of the components is zero) and $\sqrt{2}$ (if both components have magnitude 1). Call this length r. The magnitude of c will be the distance of the origin of the 2D device coordinate system from the edge, scaled by r. If the maximum horizontal or vertical size of the potentially visible region of rasterization is V, and the origin is at the lower left, then values of c for which the edge intersects the visible region must be in the range $[-2V, 2V]$, since no fragment location can be farther than $V\sqrt{2}$ from the origin and the maximum of r is $\sqrt{2}$. If the normalized value of c is outside this range, the edge is outside the visible region and we we need only look at its sign. If the sign of c is positive, then the entire visible region is inside the positive halfspace of the equation and the equation can actually be discarded from that triangle's equation set (if this would improve performance for a given implementation). If the sign is negative, the entire visible region is inside the negative halfspace of the edge equation and no fragments could ever be generated, so we can discard the triangle (this implements per-primitive view-frustum culling).

In contrast, for parameter interpolation equations, it is only possible to scale them all by a common factor—otherwise the perspective correction would be incorrect. To find a suitable factor, compare the exponents of the coefficients of all parameter equations and choose the largest exponent. Then, shift *all* parameter equation coefficients down to just fit the largest coefficient into its representable range. This process will be guaranteed to fit all coefficients into the available fixed dynamic range while maximizing interpolation precision, and is similar enough to the proceeding process that the same hardware resources can be used. This procedure will cause the precision of some parameters to suffer somewhat, however, if wildly different dynamic ranges are present. This should happen only when rasterizing primitives with extreme perspective distortion, however, if the precision is sufficiently high. The fact that parameter interpolation equations must be treated together means that while we can share edge equation evaluation for shared edges in a set of triangles, we *cannot* share evaluation of parameter equations between triangles.[1]

3 Hilbert Curve

The space-filling curve used in Figure 1 is generally called the Hilbert curve [12] (the class of *all* space-filling curves are called Peano curves). The Hilbert curve can be defined mathematically as the *limit* of the set of geometric rewriting rules shown in Figure 3. Normally, we only use an approximation to the true Hilbert curve generated with a finite number N of rewriting steps, in which case we say that we have a Hilbert curve *approximation* of order N. Interestingly, by repeatedly applying the Hilbert rewriting rules, only four different orientations of the base square (out of eight possible orientations) are generated. We can represent the transitions between these four possible orientations using the diagram shown in Figure 4.

[1] These complications mean that for a given implementation, it may or may not be advantageous to perform incremental evaluation of the parameter interpolation basis functions—it may be simpler to evaluate them directly after the frements are found.

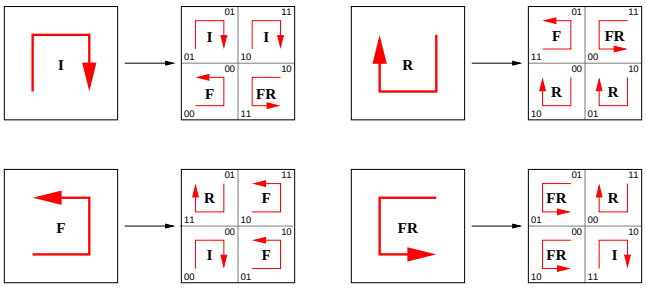

Figure 3: *The Hilbert curve can be defined as the fractal limit of a set of rewriting rules. In our application, we apply the rewriting rules only down to the level of individual pixels.*

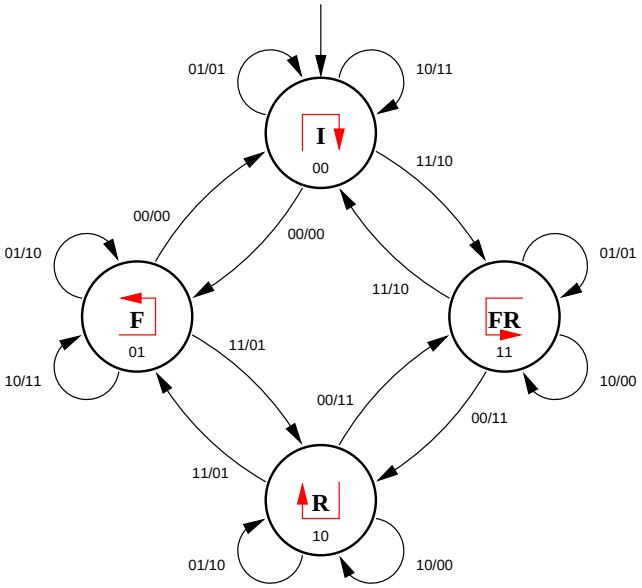

Figure 4: *The Hilbert rewriting rules can be represented as a cyclic scene graph, which can also be interpreted as a finite state machine.*

Figure 4 can be interpreted as either a cyclic scene graph [6] giving a recurrence defining the fractal Hilbert curve or as a state machine [1].

To interpret Figure 4 as a cyclic scene graph or recurrence, the arc labels $r_1 r_0 / xy$ should be read as affine transformations: xy indicates a translation in x and/or y, with an implicit scale of $1/2$, while $r_1 r_0$ indicates one of four children for each node.

To interpret the diagram as a state machine, $r_1 r_0$ should be interpreted as the high-order (input) bits of the curve arclength, and xy as the corresponding high-order output bits of the x and y position of that point along the curve. Using this state machine it is possible, in n clock cycles and using minimal amounts of hardware, to transform a $2n$-bit arclength into two n-bit positions. We have implemented this in a pipelined fashion using 9K NAND gate equivalents for a separate application (stratified sampling). For 64-bit r values we have obtained a 100MHz conversion rate, basically one conversion per clock at the maximum clock rate of our FPGA prototyping system. For comparison, a highly optimized in-register software implementation acheives a 3MHz conversion rate on a 1.4GHz Pentium 4.

The state machine can also be run backwards, translating positions into arclength. This inverse mode could be used, for instance, for translating multidimensional spatial coordinates into coherent unidimensional memory addresses. This inverse mode can also be used to back reversibly out of a Hilbert curve traversal, so recursive algorithms can be implemented without necessarily using an explicit stack.

We have labelled the states using I, R, F, and FR. These actually represent transformations from the initial orientation. The transformation I is the identity, F is a "flip" over the $x = y$ axis (which can be computed by exchanging x and y) and R is a rotation by 180° about the center of the cell (which can be computed, since x and y are single bits, by complementing both x and y). The label FR indicates a composition of F and R (which happens to be equivalent to a composition of R and F). The state assignments we have made use bit 0 for the presence of a F transformation and bit 1 for the presence of a R transformation, an encoding which lets us easily apply the required transformation of x and y using a small number of gates.

There are other bit-oriented algorithms for generating the Hilbert curve [2, 3], but we feel that the state-machine approach is the simplest for hardware implementation, at least in 2D.

4 Scanning

In this section we describe our rasterization algorithm and also provide detailed pseudocode for it. We focus on the organization of the hierarchical scanning process and its incrementalization. For simplicity, we describe the rasterization process only for a single polygon; extension of the algorithm to simultaneous rasterization of a set of polygons amounts to keeping track of which equations define which primitives, and flipping oracle results as appropriate for shared equations.

Let $\mathcal{E}$ be the set of all edge equations, and $\mathcal{F}$ the set of interpolation equations. Conceptually, we exhaustively check each potential fragment position in the framebuffer and output fragments for which $\texttt{inside}(\mathbf{E}, x, y)$ is true for all $\mathbf{E} \in \mathcal{E}$, evaluating the equations in $\mathcal{F}$ at those points. In practice, such a brute-force algorithm would be horribly inefficient. Instead, we test blocks of fragments hierarchically.

Before we evaluate the equations for fragments in some $2^n \times 2^n$ block of the framebuffer, we test whether *any* potential fragment position in that block could possibly be part of the polygon. This can be done by applying outcode tests to the corners of the block. If we can prove that a block cannot contain any fragments of the polygon, by showing that all corners are "out" against at least one equation, we can skip that block and move onto the next block at the current level of the hierarchy. Otherwise, we recursively subdivide the block into four subblocks of size $2^{n-1} \times 2^{n-1}$. Subblocks are scanned recursively until the fragment level is reached, then point tests are made to terminate the recursion. To output fragments in Hilbert order, the state machine given in Figure 4 is used to control the order in which subblocks are scanned. Finally, although the algorithm can be described recursively, the reversibilty of the incremental update (in fixed-point arithmetic) and the reversibility of the Hilbert state machine can be used to avoid the need (almost) for a stack.

It is possible to use outcodes to further "optimize" the algorithm. Specifically, once it has been proved that a block is completely interior or exterior with respect to a specific linear equation, this will also true for all subblocks of that block. Therefore, it is not really necessary to test against that particular equation again for any deeper levels of the recursion. Unfortunately, in practice, the bookkeeping required to keep track of when evaluation of such equations can be omitted and when they must be evaluated again eliminates any advantage. Normally all equations would be updated in parallel, since a relatively small amount of hardware is required to in-

crementally update each equation. Since no clock cycles can be saved anyways when using parallel evaluation, we have not used this optimization, but it might be useful in other contexts.

We incrementalize the algorithm by updating equations as we move along the Hilbert curve hierarchically. Equations are always expressed relative to the lower-left corner of the current block. After each update, the c coefficient of each equation will in fact be equal to the evaluation of the equation at the lower-left corner of the current block, and we do not need extra storage to hold the evaluated value. The interpolation equations are incrementally updated in the same way, although we do not use them when testing blocks for subdivision. When we reach a fragment, we just need to look at the current values of the c coefficient of the three interpolation equations. We can use this information to compute rational basis functions suitable for interpolation.

The top level of the rasterization algorithm is given in pseudocode in Figure 5. This algorithm depends on a few subroutines given in Figures 6, 7, 8, and 9, and finally the edge test given earlier in Figure 2. It also depends on the set $\mathcal{E}$, the set of all edge equations, and the set $\mathcal{F} = \{\mathbf{F}_0, \mathbf{F}_1, \mathbf{F}_2\}$, the set of three equations used for interpolation and perspective correction (see Section 2.2).

```
Int<⌈lg(N)⌉> n;
Int<2> s;
Int<2> r[N+1];
Int<N+1> x, y;
Bool finished;

scan(E,F) {
  n = N;
  r[N] = 0;
  x = 0;
  y = 0;
  s = F;
  finished = False;
  do {
    if (0 == n) {
      testfrag(E,F);
      update(E,F);
    } else {
      if (subdivide(E) {
        descend(E,F);
      } else {
        update(E,F);
      }
    }
    while (finished) {
        n++;
        s = prevstate[s][x[n]][y[n]];
        finished = False;
        update(E,F);
    }
  } until (N == n);
}
```

Figure 5: *Pseudocode for a hierarchical Hilbert scan of a $2^N \times 2^N$ region of the screen, using edge equation set $\mathcal{E}$. The entries of* `prevstate`*, a $(4 \times 2 \times 2) \mapsto 2$ table (32 bits), can be derived from Figure 4 by following the backward arcs. The integers x and y are treated as arrays of bits here.*

The `testfrag` subroutine tests the active equations at a specific candidate fragment position. This routine is given in Figure 6 as pseudocode. First, we test if the given fragment position is inside the polygon by checking the signs of the edge equations. We assume sampling in the lower-left corner of each pixel region (an appropriate pre-transformation of all primitives can be used to simulate center sampling if desired) so the current values of the c coefficient in the edge equations, which have been updated to be relative to the lower-left of the pixel region, can be used directly in these tests. If a fragment is to be output, then the information necessary to evaluate the interpolation basis functions at the center position is output along with the device-space position of the fragment. To compute the interpolation basis functions we just need the sample-relative coefficients of the three interpolation equations $\mathbf{F}_0$, $\mathbf{F}_1$, and $\mathbf{F}_2$, which have already been computed. Actually, derivatives, necessary for antialiasing, can also be evaluated using this information. This subroutine could also implement a parallel fragment test across a tile if greater performance were desired, or multisampling if greater quality were desired.

```
testfrag(E,F) {
  Bool interior = True;
  for (E ∈ E) {
    interior &= inside(E,x,y);
  }
  if (interior) {
    renderfrag(x,y,F);
  }
}
```

Figure 6: *Test a fragment position at the lower-left corner of pixel region and output the fragment if it belongs to a primitive's rasterization.*

The `subdivide` oracle (Figure 7) determines if there is any need to divide a block into smaller blocks. This is done by trying to prove if all corners of the block are completely contained in the negative halfspace of any equation. If this is true for any equation, we do not need to subdivide the block (when rasterizing multiple primitives simultaneously, this would have to be true for all primitives). In hardware, it is possible to classify all corners of a block against all edge equations in a single clock cycle using parallel evaluation.

If we have decided to subdivide a block, the `descend` subroutine updates the Hilbert state and all equations to be relative to the first ($r = 0$) subblock of the subdivision. The active equations need to be updated during a recursive descent if the orientation of the scan order is such that first subblock is in the upper right of the parent block, which is true for the R and FR orientations. Using the Hilbert state encoding given in Figure 4, this corresponds to the high-order bit (bit 1) of the state being set.

```
subdivide(E) {
  Bool subdiv = True;
  Bool t00, t10, t01, t11;
  for (E ∈ E) {
    t00 = (E.c < 0);
    t10 = (E.c+(E.a<<n) < 0);
    t01 = (E.c+(E.b<<n) < 0);
    t11 = (E.c+(E.a<<n)+(E.b<<n) < 0);
    subdiv &= !(t00 & t01 & t10 & t11);
  }
  return subdiv;
}
```

Figure 7: *Test all edge equations, and determine whether the current block should be subdivided. A block does not need to be subdivided if it is contained entirely in the negative halfspace of at least one equation.*

```
descend(ℰ,ℱ) {
  t = nextstate[s][r[n]];
  n--;
  if (s[1]) {
    for (E ∈ ℰ∪ℱ) {
      E.c += (E.a+E.b)<<n;
    }
  }
  r[n] = 0;
  s = t;
}
```

Figure 8: *Descend to a finer level. If the current state is* R *or* FR*, as indicated by the topmost bit of the state variable, we need to update the equations to the lower-left corner of the upper right subblock. The entries of* `nextstate`*, a* $(4 \times 4) \mapsto 2$ *table (32 bits), can be derived from Figure 4 by following the forward arcs.*

```
update (ℰ,ℱ) {
  switch (hmove[r[n]][s]) {
    case RIGHT: x += (1<<n); break;
    case LEFT:  x -= (1<<n); break;
    case UP:    y += (1<<n); break;
    case DOWN:  y -= (1<<n); break;
  }
  for (E ∈ ℰ∪ℱ) {
    switch (hmove[r[n]][s]) {
      case RIGHT: E.c += (E.a<<n); break;
      case LEFT:  E.c -= (E.a<<n); break;
      case UP:    E.c += (E.b<<n); break;
      case DOWN:  E.c -= (E.b<<n); break;
    }
  }
  r[n]++;
  finished = (0 == r[n]);
}
```

Figure 9: *Update equations when moving from one block to another at resolution level* n*. Equations are always updated four times using exact arithmetic at any given resolution level. This returns all equations to their starting point in the lower-left corner of the parent block. The table* `hmove` *(see Table 1) indicates how incremental updates in* r *translate into incremental updates in* (xy) *for each of the four possible Hilbert states.*

If no subdivision is required, or if the fragment level has been reached, or after we have returned from processing a block, we can use the `update` subroutine to incrementally update the equations along the blocks at the current level, in order by Hilbert arclength. The sequence in which blocks is visited is controlled by `hmove`, given in Table 1. The "moves", which depend on both the current r value at the current level and the current state, are designed to return the equations to their original state, relative to the lower-left corner of the parent block, four `update` calls after a `descent` call. For the I and F orientations, the last move completes a cycle to return to the lower left; for the R and FR orientations, the last move reverses the second-to-last move to return to the lower left. In both cases "extra" computation is used to return equations to a previous configuration; however, this extra computation avoids the use of a large stack which would otherwise be required to return to this previous state. Note again that all the operations in this routine can be performed in parallel, taking only a single clock cycle.

Although this is a "recursive" algorithm, only a negligibly small stack is required, namely the array for the values of r at each of the different levels of recursion, two bits per level. The size of the x and y registers also grows with resolution, but this would be required in almost any implementation, and the total number of bits in r must be only be equal to the total needed to represent these two quantities. The cost of the (virtual) elimination of the stack is some extra arithmetic to restore equations to their previous state upon return from the "recursive" processing of the subblocks of a block. However, the gate count for an implementation using a stack would be much greater.

hmove				
r	I	F	R	FR
0	UP	RIGHT	DOWN	LEFT
1	RIGHT	UP	LEFT	DOWN
2	DOWN	LEFT	UP	RIGHT
3	LEFT	DOWN	DOWN	LEFT

Table 1: *Table to control subblock progression as a function of Hilbert state and progress so far. Updates are set up so we will end up back in the lower left of the parent block after four cycles.*

5 Results

Our prototype implementation was performed on a RC1000-PP prototyping board using a Xilinx XCV1000 field-programmable gate array (FPGA). This prototype rasterizes only a single polygon and does setup in software. We are working on an implementation that rasterizes multiple triangles at once and does setup in hardware but unfortunately it was not completed in time for the publication deadline of this paper.

We implemented the algorithm given in Section 4 in Handel-C [10], a silicon compiler originally developed at Oxford and now distributed commercially by Celoxica. This system compiles directly from an extended C-like language to a hardware design targeted at a specific FPGA architecture. Handel-C is not a hardware description language like VHDL but a programming language that targets hardware. The control constructs of this language are translated into a one-hot controller structure and the arithmetic computations are translated into an appropriate datapath, using one clock cycle per statement (in other words, the C language is interpreted as a register transfer language, variables are implemented using registers, multiplexers are inferred when different statements write to the same variable, etc.). Special language constructs permit the specification of microparallelism and synchronous communication.

The RC1000-PP prototyping board runs at up to 100MHz and has four separate 2MB banks of 32-bit wide single-cycle 7ns static RAM. Data can be transferred between the host and any bank of memory using DMA while the FPGA is accessing other banks. In our system, we use one bank of memory to hold the output image and z-buffer and two others to hold the specifications of the polygons to render (letting the FPGA read from one bank while the host downloads new data into the other). We have reserved the fourth bank for storage of texture maps.[2]

The host, a 1.4GHz Pentium 4, currently performs the setup of the equations using single-precision floating-point computations. The results are then downloaded by DMA to the prototyping board.

The rasterization unit, with support for up to 16 equations[3] and

[2]Our current prototype does not use texture maps yet, unfortunately, even though one of the points of Hilbert-order traversal is better texture cache behaviour.

[3]Overkill for one triangle; we were attempting to model the gate requirements for a multiple-triangle implementation.

on-chip buffering for up to 74 parameter triples, consumes 57.9K NAND gate equivalents after compilation and optimization by the Handel-C compiler. We use 32-bit fixed-point precision for the c values of all equations and parameters and 16-bit precision for a and b, and update all equations and evaluate the subdivision oracle using multiple parallel 32-bit shifter/adder units. The rasterization unit does not output a fragment every clock, since some time is spent moving up and down the hierarchy, but does identify fragments at a rate of nearly one every two clocks for large polygons to one every $2N$ clocks for single-pixel polygons. Depending on the number of parameters to interpolate, however, more clocks may be required to actually form the interpolated parameters in the output fragment once it has been identified. Fortunately, interpolation computations can take place in parallel with a continuation of the fragment scan.

This algorithm is somewhat disadvantageous for small polygons, due to the overhead involved in hierarchical searching, and of course the coherency of the Hilbert order will not matter much if a polygon only covers one fragment. However, if several polygons are rasterized simultaneously, a relatively simple extension given the nature of the algorithm, then the overall area covered will be larger and this disadvantage will be offset. Furthermore, if the set of polygons rasterized as a group are coherent, i.e. they form a small "patch" on a surface (a consequence of triangle stripping and vertex caching geometry optimizations anyways) and use the same textures, etc. then it will be possible to exploit spatial coherence over this patch in the pixel shading unit. Note that it might be possible for two primitives in a set to cover the same fragment position. In this case, such co-located fragments should be output in the order of specification of the primitives.

Performance could also be improved by using a tiled evaluator at the lowest level, and of course by using multiple rasterizer units. Since system complexity is reduced by the elimation of the clipper, for a given gate budget more parallel rasterizers can be used. Tiled subdivision oracles could also be used at higher levels to reduce the depth of the search tree.

The storage space for the parameters in our implementation doubles as an input FIFO buffer. A streaming packet-based interface is used to communicate between modules in our system, with the maximum packet size set at $256 \times 4\text{B} = 1\text{KB}$. This is the size of the input buffer RAM in each rasterizer. When smaller packets than the maximum size are used, which of course is the usual case, the RAM is used to buffer them in FIFO fashion. The gate count could be reduced if a more direct interface were used with support for fewer parameters.

Unfortunately we have not yet implemented the setup computation in hardware. However, since the setup computations require regular computations involving only multiplication and subtraction, it should be possible to get good performance with a relatively simple implementation. Our plan is to use a shared floating-point unit for both triangle setup and fragment post-processing, and to overlap setup and interpolation computations with the search for fragments.

6 Conclusions

A rasterization algorithm has been described that can be implemented in a relatively small number of gates, generates polygon fragments in a coherent order (exact multiresolution block coherence in the case of the framebuffer locations), can rasterize multiple primitives in parallel, and can be used to interpolate a large number of parameters efficiently. This algorithm is based on edge-equation testing but uses a hierarchical search for valid pixel fragments based on the Hilbert curve. This rasterization technique works entirely in homogeneous coordinates and does not require a clipper.

Acknowledgements

This research was sponsored by research grants from the National Science and Engineering Research Council of Canada (NSERC) and the Centre for Information Technology of Ontario (CITO). Celoxica Inc. provided free access to the beta-test version of the Handel-C silicon compiler and Xilinx graciously donated an XCV1000 part. Other tools made available through the Canadian Microelectronics Corporation (CMC), specifically place-and-route software, were used to complete our hardware prototype.

References

[1] T. Bially. Space-Filling Curves: Their Generation and Their Application to Bandwidth Reduction. *IEEE Transactions on Information Theory*, 15:658–664, November 1969.

[2] Arthur R. Butz. Convergence with Hilbert's Space-Filling Curve. *Journal of Computer and System Sciences*, 3:128–146, 1969.

[3] Arthur R. Butz. Alternative Algorithm for Hilbert's Space-Filling Curve. *IEEE Transactions on Computers*, 20:424–426, April 1971.

[4] H. Fuchs, J. Goldfeather, J. Hultquist, S. Spach, J. Austin, F. Brooks, Jr., J. Eyles, and J. Poulton. Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes. In *Proc. ACM SIGGRAPH*, pages 111–120, July 1985.

[5] Ned Greene. Hierarchical polygon tiling with coverage masks. In *Proc. SIGGRAPH*, pages 65–74, 1996.

[6] John C. Hart. The object instancing paradigm for linear fractal modeling. In *Proc. Graphics Interface*, pages 224–231, May 1992.

[7] Michael D. McCool. SMASH: A Next-Generation API for Programmable Graphics Accelerators. Technical Report CS-2000-14, Department of Computer Science, University of Waterloo, April 2001. Published as part of the SIGGRAPH 2001 Course Notes.

[8] Joel McCormack and Robert McNamara. Tiled Polygon Traversal Using Half-Plane Edge Functions. In *Proc. Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 15–21, 2000.

[9] Marc Olano and Trey Greer. Triangle Scan Conversion using 2D Homogeneous Coordinates. In *Proc. Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 89–95, 1997.

[10] I. Page and R. Dettmer. Software to Silicon. *IEE Review*, 46(5):15–19, September 2000.

[11] Juan Pineda. A Parallel Algorithm for Polygon Rasterization. In *Proc. ACM SIGGRAPH*, pages 17–20, August 1988.

[12] I. J. Schoenberg. On the Peano Curve of Lebesgue. *Bull. Am. Math. Soc.*, 44:519, 1938.

[13] Douglas Voorhies. Space-Filling Curves and a Measure of Coherence. In James Arvo, editor, *Graphics Gems*, volume II, pages 26–30. Academic Press, 1991.

[14] I. H. Witten and R. M. Neal. Using peano curves for bilevel display of continuous-tone images. *IEEE Computer Graphics & Applications*, 2:47–52, May 1982.