EUROGRAPHICS 2011 / M. Chen and O. Deussen
(Guest Editors)

Volume 30 (2011), Number 2

Wavelet Rasterization

J. Manson[1] and S. Schaefer[1]

[1]Texas A&M University, USA

Abstract
We present a method for analytically calculating an anti-aliased rasterization of arbitrary polygons or fonts bounded by Bézier curves in 2D as well as oriented triangle meshes in 3D. Our algorithm rasterizes multiple resolutions simultaneously using a hierarchical wavelet representation and is robust to degenerate inputs. We show that using the simplest wavelet, the Haar basis, is equivalent to performing a box-filter to the rasterized image. Because we evaluate wavelet coefficients through line integrals in 2D, we are able to derive analytic solutions for polygons that have Bézier curve boundaries of any order, and we provide solutions for quadratic and cubic curves. In 3D, we compute the wavelet coefficients through analytic surface integrals over triangle meshes and show how to do so in a computationally efficient manner.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.3]: Picture/Image Generation - Antialiasing - Line and curve generation—[I.3.5]: Curve, surface, solid, and object representations

1. Introduction

Polygon rasterization is possibly the most fundamental operation in Computer Graphics. Nearly every modern computer includes specialized hardware that is capable of rasterizing millions of triangles per second. A rasterizer determines how to color pixels from a polygon's boundary, but pixels near the boundary are difficult to classify. Simply testing if pixels are interior to a polygon results in jagged edges, which are a form of aliasing [Cro77]. In this paper, we describe a method for analytically rasterizing an anti-aliased image.

Aliasing is a general term for any effect of sampling a signal that contains frequencies higher than half the sampling rate. In polygon rasterization, the signal is a function χ_M that is one inside the polygon and zero outside. Because χ_M changes discontinuously along polygon edges, χ_M contains infinitely high frequencies. To remove aliasing artifacts, it is necessary to filter-out high frequencies from χ_M through convolution. Different filters balance signal distortion and simplicity. From most complex to simplest, the theoretically ideal *sinc* filter removes only high frequencies, windowed *sinc* filters such as Lanczos limit filter support, piecewise-cubic filters [MN88] approximate a windowed *sinc*, while the box filter is a piecewise constant approximation thereof.

Beyond rasterizing polygons, one can rasterize curved boundaries. For example, fonts and vector images often describe regions bounded by Bézier curves. However, even methods that calculate precise anti-aliasing of polygons only approximate curves by linear pieces, which leads to aliasing artifacts for these curved shapes.

We view pixels as square regions on a display. In this interpretation, pixels that intersect a polygon's boundary can be partially covered. Rasterizing polygons into pixels that store percent occupancy rather than Boolean inside/outside information is equivalent to sampling χ_M convolved with a box filter. Although box filters are often considered to be poor approximations of the *sinc* filter, their interpretation of calculating occupancy of cells is sometimes useful, especially in 3D. Because directly calculating the convolution integral is difficult, the integral is often approximated by taking multiple samples per pixel [DW85].

Although we typically think of rasterization as a 2D problem, we can extend the idea to 3D. In 3D, the equivalent operation is to calculate the occupancies of cube cells from the triangles that enclose a volume. This volumetric rasterization (voxelization) of an object builds an implicit representation that is useful for operations, such as collision detection, Constructive Solid Geometry (CSG), and fluid simulation, that is easier to calculate over volumes than boundaries.

Collision detection, for example, determines if objects interpenetrate and can be solved by querying if points are inside (one), outside (zero), or near the boundary (fractional values) of an object. If the cell containing a test point has

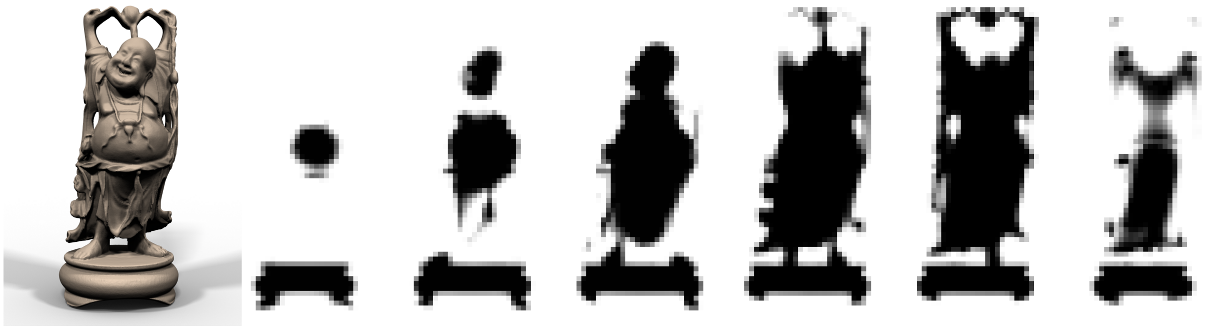

Figure 1: *Slices from a 3D rasterization of the Happy Buddha statue computed on a 64^3 grid to illustrate the anti-aliased nature of wavelet rasterization.*

a fractional occupancy, we only need to test the part of the boundary that intersects that cell.

Implicit representations also provide a natural method of calculating CSG set operations. We can approximate set operations between two objects a and b by $min(a,b)$ for intersection, $max(a,b)$ for union, and $min(a,1-b)$ for difference. We can then convert the result back to a boundary representation as shown in Figure 8. Alternately, we can accelerate exact CSG algorithms by using implicit representations of objects to quickly classify surface elements of a mesh as being completely inside or outside each object. We can then perform exact intersection tests within the remaining indeterminate voxels [CK10].

In fluid simulations, it is natural to model surface tension using a surface mesh, while pressure and advection are best computed over a volumetric grid. Coupled with surface contouring methods such as Marching Cubes [LC87], efficient methods for implicitizing objects can accelerate fluid simulations with air-water interfaces [TWGT10]. Calculating precise cell occupancies is important to preserve volume in the simulation and to maintain a smooth surface.

Contributions

We calculate the exact wavelet coefficients of rasterized polygons, fonts, and volumes. To compute the coefficients efficiently, we transform integrals over the interior of an object to integrals over the object's boundary. There are many integrals that we could use in this transform, so we choose integrals that have the smallest support and computational cost. The result is a fast algorithm for rasterizing 2D shapes and implicitizing 3D volumes. Furthermore, our algorithm is independent of surface connectivity and is robust to degenerate inputs and small gaps or overlaps in the surface.

1.1. Related Work

Perhaps the most obvious method for rasterizing an object is to sample points over a regular grid. For each point in the grid we cast a ray and count the number of intersections between the ray and the boundary [Lin90, TGR04]. If the number of intersections is odd, we classify the point as interior. Otherwise, the point is exterior. This approach is useful because it does not depend on the orientation of the boundary, but ray-intersections are difficult to handle robustly in 3D.

The classic scan-line rasterizer takes the more-efficient approach of splitting an image into separate horizontal scan-lines. In this algorithm, polygon edges split scan-lines into separate spans that we then classify as inside or outside. There are many extensions to this basic algorithm that take multiple samples within each pixel to produce anti-aliased images [DW85, Coo86, JC99, Kal07].

One of the first analytic box-filter rasterizers [Cat78] clips polygons in a scene to each pixel and then clips polygons against each other, ordered by depth. Areas of clipped polygons are then added to find the pixel's color. Duff developed a scan converter [Duf89] that calculates analytic convolutions with cubic splines. Similar to our method, Duff clips only the edges of polygons to pixels. Our main advantage over Duff's algorithm is robustness to small imperfections in input that cause errors to propagate across a line in scan-line algorithms. This property is more important in 3D than in 2D because 3D data is more likely to contain imperfections. A method of extruding box splines to filter triangles has also been developed [McC95], but requires simplicial decompositions of shapes.

Most rasterization algorithms are specific to polygons and render curves by subdividing curves into dense collections of lines [HE93]. However, there has been interest recently to directly rasterize fonts and vector graphics that are bounded by Bézier curves. A typical approach to calculating anti-aliased curves is to set the intensity of pixels based on an approximate distance to the curve. One early method [FF97] uses this approach to draw strokes, but handles nearby lines inadequately by calculating the minimum distance to all curves rather than actual overlap. Using a distance-based pre-filter to draw closed shapes only approximates true convolution.

Newer algorithms tend to be designed with GPUs in mind. One such algorithm [LB05] uses the implicit form of quadratic Bézier curves, which are bounded by the triangle formed by their control points. Loop et al. designed a pixel shader to color pixels in each triangle using the implicit form to determine if pixels are inside or outside the curve. However, this method requires the curve's interior to be triangulated and only approximates anti-aliasing. Other hardware-accelerated techniques [QMK06, QMK08, NH08] improve the quality of anti-aliasing by pre-filtering images with better approximated distance fields, but these methods are still approximate.

In addition to rasterizing polygons and curves, we consider the 3D equivalent of voxelizing surfaces. Some computationally expensive algorithms [FT97, CC95] test points inclusion. Other algorithms [FL00, HW02, DCB*04, ED06, ZCEP07, ED08] use the special-purpose hardware in a GPU to accelerate volumetric rasterization. However, these algorithms use no filtering, which leads to obvious aliasing. Binary volumes are adequate for some applications, like collision detection, but CSG operations and other methods that extract surfaces from a volumetric representation require anti-aliasing to produce attractive surfaces. Although super-sampling approximates the anti-aliased representation of these binary volumes, super-sampling volumes is extremely expensive.

Other methods [IK00] approximate the signed distance function of a surface, but often rely on finding closest points on the surface [BA05]. Some researchers [SPG03, SOM04] have used the GPU to accelerate the computation of signed distance functions, but these methods are still slow because of the complexity of the distance function.

Although seemingly unrelated to rasterization, surface reconstruction methods can also be used to rasterize an object. Surface reconstruction algorithms approximate an implicit function of an object from a set of points sampled over the object's surface. The reconstructed surface is then calculated as a level set of the function. Several methods [Kaz05, KBH06, ACSTD07, MPS08] estimate the indicator function for the object or the indicator function convolved with a small smoothing kernel.

The wavelet reconstruction work of Manson et al. [MPS08] is closely related to our method. Their method estimates wavelet coefficients of an indicator function from a set of noisy point samples. In order to smooth the reconstructed surface, the authors chose to use smooth wavelets with wide support and convolve their resulting function with a Gaussian kernel. In contrast, we compute exact integrals for Haar wavelets from lines, curves, or triangles in 3D rather than summing over point sets, which yields a more accurate and efficient rasterization.

Wavelets have also been applied to rendering ray-traced scenes, but in a way that is very different from our method. Overbeck et al. [ODR09] use ray-traced color samples in the image plane to build wavelet coefficients of the image and use the variance of the mean of wavelet coefficients to determine where more samples are required. Additionally, they use a smooth basis and reduce the contribution of coefficients with high variance so that noisy regions of the image (for example around out-of-focus objects, in soft shadows, or on semi-glossy surfaces) appear blurred. In contrast, our method uses wavelets to calculate interiors of 3D surfaces and 2D polygons.

2. Rasterizing Objects Using Wavelets

Wavelets provide a basis for representing functions through a hierarchy of localized refinements. They have a wide variety of applications, from solving differential equations, to digital image processing, signal processing, and surface reconstruction. The main advantage of wavelets over other representations of a function is that wavelets are localized in both the spatial and frequency domains. Although our explanation and derivations extend to higher dimensions, for simplicity, we initially restrict our discussion to 2D and will extend the method to 3D later.

We wish to rasterize objects by calculating the percent occupancy of voxels in a regular grid. If the area M is the set of points inside an object with boundary ∂M, represented as a set of edges, then χ_M is defined as

$$\chi_M(p) = \begin{cases} 1, & p \in M \\ 0, & otherwise. \end{cases} \tag{1}$$

This function is an implicit representation of M from which we can extract the boundary by finding the points where χ_M transitions from zero to one. In particular, χ_M defines the set of points that would be drawn if the polygon ∂M was rasterized at infinite resolution. Taken to the limit, summing super-samples [DEM96] over a pixel is equivalent to applying a box-filter or integrating χ_M over the pixel area. The value of a pixel P is therefore given by

$$\frac{\int_P \chi_M(p)\, dp}{\int_P dp}. \tag{2}$$

This equation shows that the value of a pixel is, ideally, equal to the area of the polygon that intersects the pixel divided by the area of the pixel. Our approach to rasterizing polygons is to calculate the wavelet coefficients of χ_M up to pixel resolution and then to invert the wavelet transform to complete the rasterization.

Wavelets provide an orthonormal basis that allows local refinements by adding higher-resolution basis functions. Wavelets are represented by a scaling function ϕ and a wavelet function ψ.

We use the following construction of two-dimensional wavelets. Let $\psi^0 = \phi$, $\psi^1 = \psi$, E' be the set of vertices $\{(0,0),(0,1),(1,0),(1,1)\}$, and $E = E' \backslash \{(0,0)\}$. For each $e = (e_x, e_y) \in E'$, $j \in \mathbb{N}$, and $k = (k_x, k_y) \in \mathbb{Z}^2$, we define

$$\psi^e_{j,k}(p) = 2^j \psi^{e_x}(2^j p_x - k_x)\psi^{e_y}(2^j p_y - k_y)$$

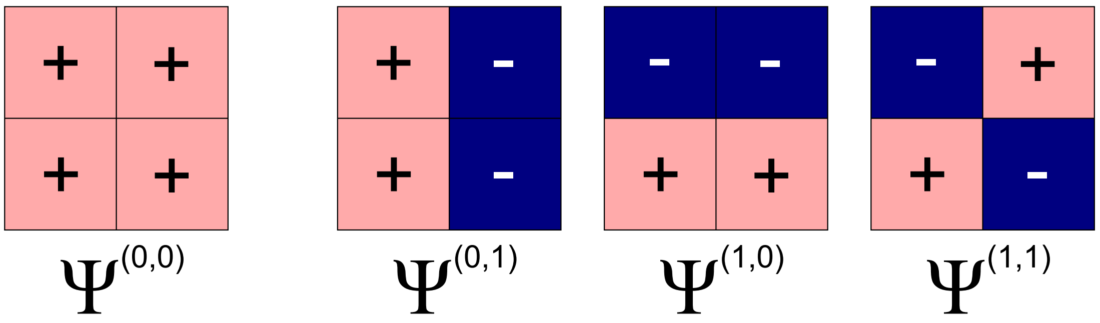

Figure 2: *The 2d Haar basis functions. Each function is shown over the domain $[0,1)^2$ and is piecewise constant ($-1/+1$) on each quadrant. Translations of $\bar{\Psi}^{(0,0)}$ give the low-resolution representation of the function, while scalings and translations of the functions $\bar{\Psi}^{(1,0)}$, $\bar{\Psi}^{(0,1)}$, and $\bar{\Psi}^{(1,1)}$ add high-resolution details.*

where $p = (p_x, p_y)$. Every function g that is locally integrable on $\mathbb{R}^2$ has the wavelet expansion

$$g(p) = \sum_{k\in\mathbb{Z}^2} c_{0,k}^{(0,0)}\psi_{0,k}^{(0,0)}(p) + \sum_{j\in\mathbb{N}}\sum_{k\in\mathbb{Z}^2}\sum_{e\in E} c_{j,k}^{e}\psi_{j,k}^{e}(p), \quad (3)$$

where each $c_{j,k}^e$ is given by

$$c_{j,k}^e = \iint\limits_{\mathbb{R}^2} g(p)\psi_{j,k}^e(p)\,dp. \quad (4)$$

Here, the index e indicates which basis functions is used and k denotes its translation at resolution j. Because the functions have supports that are power of two contractions of a square, the wavelet hierarchy in 2D is naturally represented by a quadtree. Note that j controls the resolution of the wavelet expansion. In practice, we truncate j to stop at pixel resolution.

If we consider the wavelet coefficients of χ_M and use the definition of χ_M in Equation 1, Equation 4 reduces to

$$c_{j,k}^e = \iint\limits_{M} \psi_{j,k}^e(p)\,dp. \quad (5)$$

Integrating over the domain of M is difficult, so we use the divergence theorem to relate the integral over M to an integral over its boundary ∂M. The divergence theorem states that

$$\iint\limits_{M} \nabla\cdot F_{j,k}^e(p)\,dp = \oint_{p\in\partial M} F_{j,k}^e(p)\cdot n(p)\,d\sigma \quad (6)$$

where $F = (f_x, f_y)$ is a vector-valued function on $\mathbb{R}^2$, $n(p)$ is the outward unit normal to the curve ∂M at point p, and $d\sigma$ is the differential length of ∂M. By finding functions $F_{j,k}^e$ that satisfy $\nabla\cdot F_{j,k}^e = \psi_{j,k}^e$, we can calculate the wavelet coefficients of χ_M using only the polygon boundary in the line integrals

$$c_{j,k}^e = \sum_i \int_0^1 F_{j,k}^e(P_i(t))\cdot n(P_i(t))\|P_i'(t)\|dt. \quad (7)$$

where P_i represents the i^{th} polynomial segment of the boundary (line segment for polygons).

This construction follows that of Manson et al. [MPS08] except that we show how to compute the integral exactly rather than to approximate the integral by summing over points. We can use any orthogonal basis, each of which offers a trade-off among support, smoothness, symmetry, and ease of computation. Unlike more complex wavelets, Haar wavelets [Haa10] have small support and analytic functions. Specifically, the scaling function

$$\phi(t) = \begin{cases} 1, & 0 \le t < 1 \\ 0, & otherwise. \end{cases}$$

generates the Haar basis, and ψ is given by

$$\psi(t) = \phi(2t) - \phi(2t-1).$$

The 2D Haar wavelets, shown in Figure 2, exactly represent piecewise-constant functions made of squares and model a box-filtered sampling of an image given by Equation 2. Because χ_M is constant over both the interior and exterior and wavelets have constant precision, wavelet coefficients are only non-zero where the boundary intersects the basis functions. This property yields an adaptive quadtree that is refined only along the boundary of the polygon.

We make a few simplifying assumptions to facilitate analysis. First we calculate basis functions over the $[0,1)^2$ domain (the support of the 2D Haar basis) by translating the input edges by $-k$ and scaling by 2^j. Furthermore, we clip edges to this domain because the support of the wavelet functions is only $[0,1)^2$. Note that we only need to clip the boundary edges rather than the polygons themselves because we calculate a boundary integral. This simplification allows us to drop the j,k subscripts so that

$$\psi^e(p) = \psi^{e_x}(p_x)\psi^{e_y}(p_y).$$

There are many possible function F^e in 2D that satisfy

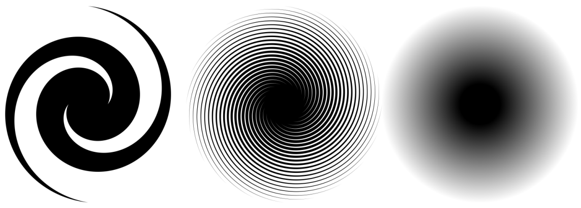

Figure 3: *We show spirals with 2, 20, and 10000 arms. The images were all rendered at* 512^2 *pixels on a side. When many arms pass through each pixel there is no aliasing from numerical inaccuracies.*

$\nabla \cdot F^e(p) = \psi^e(p)$. In particular, if $\alpha+\beta=1$, then

$$F^e(p) = \left(\alpha\bar{\Psi}^{e_x}(p_x)\psi^{e_y}(p_y), \beta\psi^{e_x}(p_x)\bar{\Psi}^{e_y}(p_y)\right)$$

where

$$\bar{\Psi}^{\ell}(t) = \int_0^t \psi^{\ell}(s)\, ds$$

and $\ell \in \{0,1\}$. Not all choices of α, β yield practical solutions or efficient calculations. In the following sections we show how we choose α, β such that Equation 7 yields a calculation that has both small support and low computational cost.

2.1. 2D Polygons

The $c^{(0,0)}$ coefficient is special because it exists only for the root node of the quadtree and gives the value that is refined by all other wavelet coefficients. The $c^{(0,0)}$ coefficient also has the clear geometric interpretation of being the area of the polygon. Letting $\alpha = \beta = \frac{1}{2}$ in Equation 7 yields

$$\int_0^1 F^{(0,0)}(P(t)) \cdot n(P(t)) \|P'(t)\| dt = \frac{1}{2} det\left(v_0, v_1\right)$$

where v_0 and v_1 are the end-points of the edge defined by P. Adding determinants of edges is geometrically equivalent to adding the areas of the triangles formed between edges and the origin, which is a simple method for using signed areas to compute the area of a polygon.

The three coefficients other than $c^{(0,0)}$ calculate the difference between a cell and its sub-cells at the next higher resolution. These three refinement coefficients, in addition to the known scale coefficient, uniquely determine values of all four sub-cells.

Now consider the $c^{(1,0)}$ coefficient ($c^{(0,1)}$ follows in a similar manner). Again, we could choose $\alpha = \beta = \frac{1}{2}$ to give

$$F^{(1,0)}(p) = \tfrac{1}{2}(\bar{\Psi}(p_x)\phi(p_y), \psi(p_x)\bar{\Phi}(p_y)),$$

where $\bar{\Psi} = \bar{\Psi}^1$ and $\bar{\Phi} = \bar{\Psi}^0$. Although this function satisfies the divergence theorem, $F^{(1,0)}$ has infinite support because $\bar{\Phi}$ has support of $[0,\infty)$. We only want to use detail functions with finite support such that we limit the number of edges that influence a coefficient. Notice, however, that the support of $\bar{\Psi}$ is finite and is $[0,1)$. By choosing $\alpha = 1$, $\beta = 0$ for $F^{(1,0)}$ and $\alpha = 0$, $\beta = 1$ for $F^{(0,1)}$, we obtain the compactly supported functions

$$\begin{aligned} F^{(1,0)}(p) &= (\bar{\Psi}(p_x), 0) \\ F^{(0,1)}(p) &= (0, \bar{\Psi}(p_y)). \end{aligned}$$

We must also calculate the coefficients of the wavelets that add differences across the diagonal. Manson et. al. [MPS08] chose the most symmetric set of functions F that have compact support and thus used the diagonal detail function

$$F^{(1,1)}(p) = \frac{1}{2}(\bar{\Psi}(p_x)\psi(p_y), \psi(p_x)\bar{\Psi}(p_y)).$$

This symmetric solution is ideal for noisy data, but our boundary is a perfect polygon, and noise is not a concern. Therefore, we simplify this computation even more by choosing $\alpha = 1$, $\beta = 0$ to give

$$F^{(1,1)}(p) = (\bar{\Psi}(p_x)\psi(p_y), 0).$$

The functions F^e are piecewise-linear because $\bar{\Psi}$ is piecewise-linear. However, each quadrant is linear, so we evaluate these integrals by splitting each polygon edge P_i into the quadrants and transforming the domains of the quadrants to be $[0,1)^2$. We then add the contribution of each split edge to the coefficients. Quadrants that do not contain an edge contribute nothing. Appendix A gives concise formulas for computing the contribution to each wavelet coefficient for each node of the quadtree.

2.2. 2D Bézier Curves

It is common to specify closed regions by quadratic and cubic Bézier curves in font rendering or vector graphics.

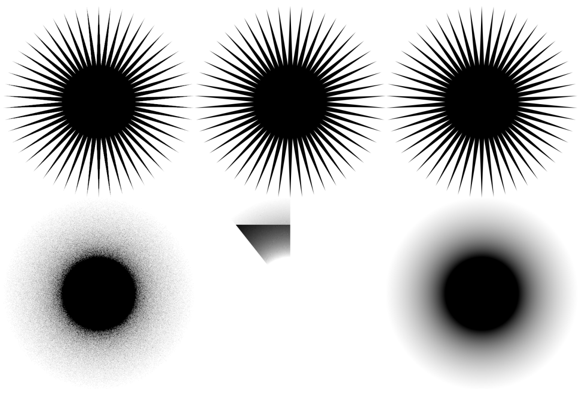

Figure 4: *We compare the quality of rasterizing a 50-point (top) and 100,000-point (bottom) star with the GPU using 16xQ supersampling, AGG, and our wavelet rasterizer on a* 512^2 *grid.*

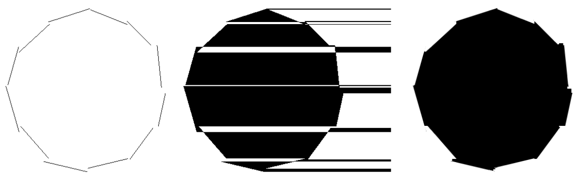

Figure 5: *We rasterize a polygon made of disconnected edges (left) using a standard scanline rasterizer (center) and our wavelet rasterizer (right). Wavelets localize errors because of their local support.*

Because Equation 7 is very general, we can handle curved boundaries in addition to straight edges. We represent the boundary of a shape as a set of general Bézier curves parameterized by $t \in [0,1)$. Using the fact that $n(P(t)) = P^{\perp}(t)/\|P'(t)\|$, where $P^{\perp}(t) = (-P_y'(t), P_x'(t))$, Equation 7 becomes

$$c^e = \sum_i \int_0^1 F^e(P_i(t)) \cdot P_i^{\perp}(t)\, dt.$$

Notice that, because each segment $P_i(t)$ is polynomial, $P_i^{\perp}(t)$ and $F^e(P_i(t))$ are also polynomial. The definite integral is then easy to evaluate and results in an expression that is quadratic in the Bézier control points. Like in Section 2.1, F^e is piecewise-polynomial, and we can compute the coefficients c^e using the same formula in Equation 8 of Appendix A, except that we replace the functions with those in Appendix B for quadratic and cubic Bézier curves. The only remaining difficulty is to recursively clip Bézier curves to cells of the quadtree, for which we use a recursive subdivision scheme [SP86].

2.3. 3D Triangle Surfaces

In 3D, we assume the boundary ∂M of our volume M is composed of triangles. Like in 2D, we define our wavelet basis as the tensor product of Haar scale and wavelet functions so that there are eight types of functions in total, indexed by e, that we contract and translate by j,k.

$$\psi^e_{j,k}(p) = 2^{3j/2}\psi^{e_x}(2^j p_x - k_x)\psi^{e_y}(2^j p_y - k_y)\psi^{e_z}(2^j p_z - k_z)$$

Also like in 2D, we translate and scale input triangles rather than modify the basis functions and clip transformed triangles to the unit cube. We therefore restrict our discussion to the normalized basis

$$\psi^e(p) = \psi^{e_x}(p_x)\psi^{e_y}(p_y)\psi^{e_z}(p_z).$$

To satisfy Equation 6, we find functions F^e whose divergences are equal to ψ^e. These functions have the general form $\alpha+\beta+\gamma = 1$,

$$F^e(p) = \begin{pmatrix} \alpha\bar{\Psi}^{e_x}(p_x)\psi^{e_y}(p_y)\psi^{e_z}(p_z) \\ \beta\psi^{e_x}(p_x)\bar{\Psi}^{e_y}(p_y)\psi^{e_z}(p_z) \\ \gamma\psi^{e_x}(p_x)\psi^{e_y}(p_y)\bar{\Psi}^{e_z}(p_z) \end{pmatrix}.$$

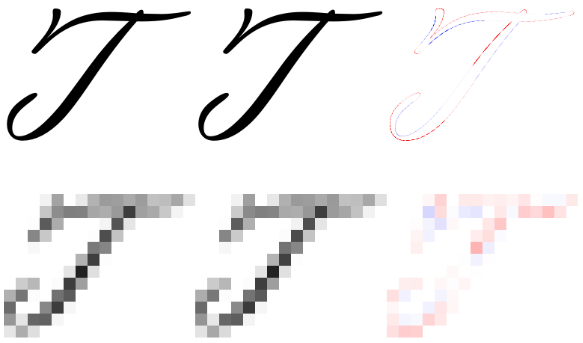

Figure 6: *Comparison of font rendering between FreeType (left) and our Wavelet algorithm (center). In the difference image (right), red values indicate that our rendering has a higher cell coverage while blue indicates that we have a lower cell coverage. Differences are multiplied by a factor of 10 for visibility.*

We choose functions that have as small a support as possible and that are as efficient to compute as possible, yielding

$$\begin{aligned}
F^{(0,0,0)}(p) &= \tfrac{1}{3}(\bar{\Phi}(p_x), \bar{\Phi}(p_y), \bar{\Phi}(p_z)) \\
F^{(1,0,0)}(p) &= (\bar{\Psi}(p_x), 0, 0) \\
F^{(0,1,0)}(p) &= (0, \bar{\Psi}(p_y), 0) \\
F^{(0,0,1)}(p) &= (0, 0, \bar{\Psi}(p_z)) \\
F^{(1,1,0)}(p) &= (\bar{\Psi}(p_x)\psi(p_y), 0, 0) \\
F^{(1,0,1)}(p) &= (\psi(p_x)\bar{\Psi}(p_z), 0, 0) \\
F^{(0,1,1)}(p) &= (0, \bar{\Psi}(p_y)\psi(p_z), 0) \\
F^{(1,1,1)}(p) &= (\bar{\Psi}(p_x)\psi(p_y)\psi(p_z), 0, 0).
\end{aligned}$$

Again, the wavelet functions (functions 2-8) have finite support, although the scale function ($F^{(0,0,0)}$) does not, but there is only one top-level scale function corresponding to the root node of the octree. Also like in 2D, the symmetric solution $\alpha = \beta = \gamma = 1/3$ for the $c^{(0,0,0)}$ coefficient gives the determinant of the triangle. This is equivalent to the signed volume of the tetrahedron formed between the triangle and the origin.

$$c^{(0,0,0)} = \int_{p\in T} F^{(0,0,0)}(p)\cdot n d\sigma = \frac{1}{6} det\left(v_0, v_1, v_2\right)$$

We compute the remaining coefficients just like in 2D by splitting triangles with vertices (v_0, v_1, v_2) into octants that we label $Q_{i,j,k}$. We give the closed-form solution to the piecewise integral in Appendix C.

3. Results

We compare the 2D rasterization performance of our algorithm against other freely available, high-quality scanline rasterizers. Specifically, we compare our polygon rasterization on a Core i7 960 against Anti-Grain Geometry (AGG),

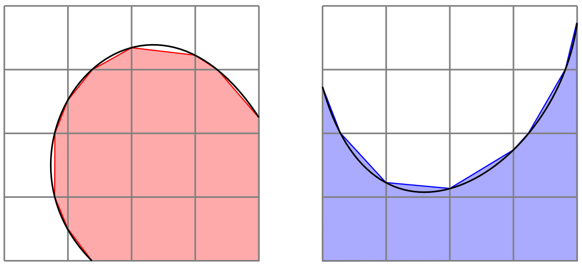

Figure 7: *Approximation of Bézier curves by line segments introduces error, even when a line segment is used for each pixel the curve intersects. This approximation underestimates coverage in convex regions (left) and overestimates coverage in concave regions (right).*

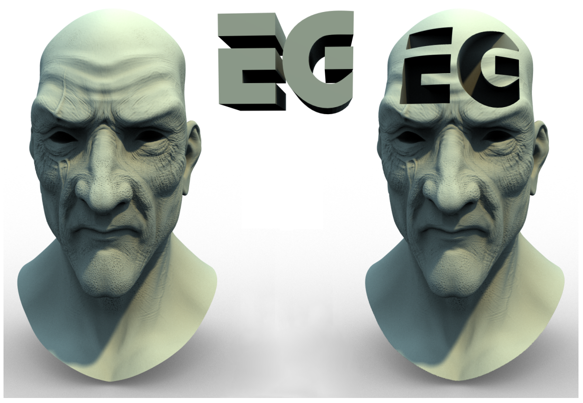

Figure 8: *We calculate the CSG set difference between the head and the Eurographics logo using an anti-aliased rasterization of each model on a* 1024^3 *grid with our method.*

which is an open-source, highly-optimized software rasterizer, and native GPU rasterization on an Nvidia 8800GT. For font rasterization, we compare against a high-quality open-source font rasterizer called FreeType. Even though our algorithm is relatively efficient in terms of computation, we cannot compete with the speed of native hardware or even highly optimized software implementations with assembly tuning. For complex shapes, the speed of our algorithm is about a factor of three slower than these optimized implementations. We rasterized a circle with a million vertices at 1024^2 resolution in 50.2ms on the GPU with 16xQ anti-aliasing, 36.8ms with AGG, and 107ms with our method.

Many rasterizers, like the GPU, only operate on triangles, which means that the shape must be triangulated before rendering. However, some shapes, like the one in Figure 3, can be difficult or time-consuming to triangulate. The figure shows three polygons of increasing complexity. Each arm of the spiral is composed of 10,000 line segments, and the arms combine to form a complex concave shape. Our algorithm still accurately rasterizes the shape even when the number of spirals (10,000 arms in a 512^2 image) is far greater than the image resolution. We encourage the reader to zoom into the figure to see the rasterization without artificial aliasing caused by deficiencies in the sampling algorithm for PDFs.

Although Figure 3 contains shapes that are difficult to triangulate, Figure 4 shows an example of a shape that is easy to triangulate and can be compared with other rasterization implementations. The figure shows an example of a 50-point star and a 100,000-point star rendered by the GPU, AGG, and our method. The 50-point stars look very similar with the different rasterizers, but some aliasing is present in the GPU rendering since 16 samples yields a total of 16 gray levels for the image. The high-frequency shape on the bottom illustrates the artifacts of super-sampling on the GPU, even with the highest anti-aliasing setting, as well as AGG's failure in this case. In contrast, our result smoothly transitions from black to white.

Quality and robustness are strong points in favor of wavelet rasterization. Wavelets build a low-resolution image that is subsequently refined in localized areas, which means that the overall picture is retained even in the presence of degeneracies and holes. Small errors in the polygon can have a large effect, as shown in the example of the non-closed polygon in Figure 5, because a scanline rasterizer with an even-odd fill-rule propagates information only from the current line to the right. However, the wavelet rasterizer uses information in both the x and y directions to refine a coarse image locally, but requires oriented edges. Although it is difficult to say what the correct rasterization of a non-closed polygon is, wavelet rasterization localizes rasterization errors and produces a plausible image.

For polynomial boundaries such as those found in fonts and vector graphics, we calculate the occupancy of pixels analytically rather than segment the curve into dense collections of lines. Figure 6 shows a comparison between our output and the output of FreeType. We rasterized an upper case "T" with both methods at $256pt$ and $16pt$ sizes. Blue pixels indicate that our image had lower occupancy, whereas red pixels indicate we had higher occupancy (multiplied by ten for visibility) in the difference images. Notice that FreeType overestimates occupancies in regions of negative curvature and underestimates occupancies in regions of positive curvature. This artifact is primarily an effect of the bias introduced by linear approximation, as shown in Figure 7.

Wavelet rasterization is particularly useful for rasterizing volumes of triangle meshes. Table 1 shows the times taken to rasterize triangle meshes of increasing complexity. The highest resolution mesh we use is a reconstruction of Michelangelo's David statue, which contains 7.2 million triangles that we rasterized at a resolution of 4096^3. Storing one byte per voxel consumes 64GB of space at this resolution, but our adaptive octree stores the entire function in memory because

		256^3		4096^3	
	polys	coeff	synth	coeff	synth
Armadilloman	30.0k	.113	.022	7.31	3.99
Head	477k	.393	.023	12.0	4.74
Buddha	1.09M	.557	.021	10.7	3.34
David 2mm	7.23M	2.25	.019	14.8	1.79

Table 1: *Time taken (in seconds) to rasterize volumes of increasing complexity at 256^3 and 4096^3. We show the time taken for coefficient calculation and synthesis separately.*

the tree is only refined around the boundary of the surface. We spend the majority of our time computing the wavelet coefficients, and this computation is proportional to the surface area of the object times tree depth. However, computing the function values from these coefficients (i.e. synthesis) over a uniform grid is proportional to the object's volume and grows quickly as the resolution increases.

It is difficult to demonstrate anti-aliasing in a volumetric image, but we have endeavored to do so in Figure 1. This figure shows slices through the volume of the Happy Buddha statue. Notice that the silhouette of each slice is anti-aliased. Moreover, voxels have partial occupancies at the front and back of the statue because anti-aliasing occurs in the z-dimension as well as the x,y-dimensions. This effect is most easily visible on Buddha's back.

Anti-aliasing is important for many algorithms that process rasterized volumes. For example, CSG operations can be performed by rasterizing the volumes of two meshes, performing a pairwise CSG operation on the two volumes, and then extracting a surface as a level set using an algorithm such as Marching Cubes (MC) [LC87]. Figure 8 shows such an operation using our wavelet rasterization on a 1024^3 grid. The quality of the surface generated by MC depends on the rasterization algorithm. Figure 9 shows the result of using a binary rasterization, which is typical of other methods [CC95, FT97, FL00, HW02, DCB∗04, ED06, ZCEP07, ED08], and our anti-aliased rasterization over a 256^3 grid. Note that MC smoothes the surface from the binary voxelization because vertices lie at the midpoints of grid edges and are connected using the MC table. Even so, MC generates only a small set of orientations for the polygons from a binary voxelization, which produces a poor-quality surface.

4. Conclusions and Future Work

We believe that 2D and 3D rasterization is a fundamental problem in Computer Graphics, and our algorithm offers a method for analytically computing anti-aliased, box-filtered rasterizations. The method we present is efficient and general in that we can rasterize arbitrary 2D polygons, shapes bounded by Bézier curves, and 3D triangle surfaces.

In wavelet rasterization, wavelet synthesis and analysis correspond to pre- and post-filtering. Direct extension to

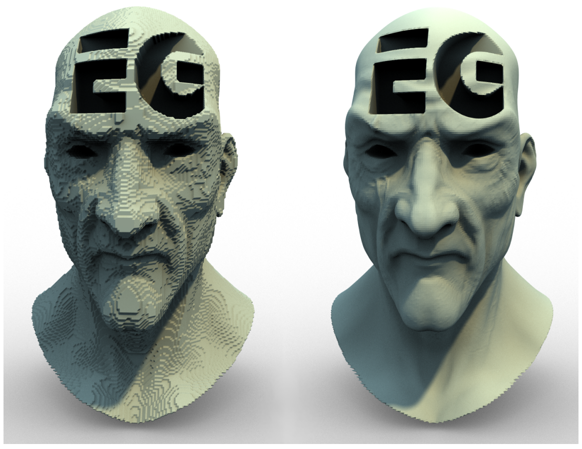

Figure 9: *The CSG operation from Figure 8 computed on a 256^3 grid contoured with Marching Cubes. The left shows the result of using binary rasterization and the right shows the result from our anti-aliased rasterization.*

higher order filters is trivial for filters that form orthogonal bases, though most common filters do not satisfy this property. Interestingly, *sinc* forms an orthogonal wavelet basis but has infinite support. We believe that extending our method to non-orthogonal filters is non-trivial but would like to explore this possibility in the future.

Wavelet rasterization also provides anti-aliased images at multiple resolutions that we can compute by truncating the summation of the detail coefficients in Equation 3 before pixel resolution. This progressive refinement of rasterized images suggests the interesting possibility of generating a fixed-framerate rasterizer that continuously adds detail until a time limit for the frame is reached. Extremely detailed geometry would simply result in a more pixelated image rather than dropped frames. However, such a change would require a relatively large overhaul of the current rasterization pipeline used by modern GPUs.

Another interesting aspect of wavelet rasterization is that it is extremely easy to parallelize. If addition is atomic, the contribution of every line segment, curve, or triangle can be processed independently. Each depth can also be processed independently, although it is probably more efficient to reuse the clipping operations from parent cells. Conversion of coefficients to function values is even more easily parallelized, because memory accesses are disjoint.

Acknowledgements

Funding is from NSF grant CCF-07024099 and DARPA grant HR0011-09-1-0042. The head model is by Valdic. Rouguan Huang helped render the 3D shapes.

References

[ACSTD07] ALLIEZ P., COHEN-STEINER D., TONG Y., DESBRUN M.: Voronoi-based variational reconstruction of unoriented point sets. In *Symposium on Geometry Processing* (2007), pp. 39–48. 3

[BA05] BÆRENTZEN J. A., AANÆS H.: Signed distance computation using the angle weighted pseudonormal. *IEEE Transactions on Visualization and Computer Graphics 11* (2005), 243–253. 3

[Cat78] CATMULL E.: A hidden-surface algorithm with antialiasing. *SIGGRAPH Comput. Graph. 12* (1978), 6–11. 2

[CC95] CARVALHO P. C. P., CAVALCANTI P. R.: *Graphics Gems V*. AP Professional, Chestnut Hill, MA, 1995, ch. Point in Polyhedron Testing Using Spherical Polygons, pp. 42–49. 3, 8

[CK10] CAMPEN M., KOBBELT L.: Exact and robust (self-) intersections for polygonal meshes. *Computer Graphics Forum (Proceedings of Eurographics) 29*, 2 (2010), 397–406. 2

[Coo86] COOK R. L.: Stochastic sampling in computer graphics. *Transactions on Graphics 5*, 1 (1986), 51–72. 2

[Cro77] CROW F. C.: The aliasing problem in computer-generated shaded images. *Commun. ACM 20* (1977), 799–805. 1

[DCB*04] DONG Z., CHEN W., BAO H., ZHANG H., PENG Q.: Real-time voxelization for complex polygonal models. In *Pacific Graphics* (2004), pp. 43–50. 3, 8

[DEM96] DOBKIN D. P., EPPSTEIN D., MITCHELL D. P.: Computing the discrepancy with applications to supersampling patterns. *Transactions on Graphics 15*, 4 (1996), 354–376. 3

[Duf89] DUFF T.: Polygon scan conversion by exact convolution. In *International Conference On Raster Imaging and Digital Typography* (1989), pp. 154–168. 2

[DW85] DIPPÉ M. A. Z., WOLD E. H.: Antialiasing through stochastic sampling. In *SIGGRAPH* (1985), pp. 69–78. 1, 2

[ED06] EISEMANN E., DÉCORET X.: Fast scene voxelization and applications. In *Symposium on Interactive 3D Graphics and Games* (2006), pp. 71–78. 3, 8

[ED08] EISEMANN E., DÉCORET X.: Single-pass GPU solid voxelization for real-time applications. In *Graphics Interface* (2008), pp. 73–80. 3, 8

[FF97] FABRIS A. E., FORREST A. R.: Antialiasing of curves by discrete pre-filtering. In *SIGGRAPH* (1997), pp. 317–326. 2

[FL00] FANG S., LIAO D.: Fast CSG voxelization by frame buffer pixel mapping. In *Symposium on Volume Visualization* (2000), pp. 43–48. 3, 8

[FT97] FEITO F. R., TORRES J. C.: Inclusion test for general polyhedra. *Computers and Graphics 21*, 1 (1997), 23–30. 3, 8

[Haa10] HAAR A.: Zur Theorie der orthogonalen Funktionensysteme. *Mathematische Annalen 69* (1910), 331–371. 4

[HE93] HERSCH R. D., (ED.): Font rasterization, the state of art. In *Visual and Technical Aspects of Type*. Cambridge University Press, 1993, pp. 78–109. 2

[HW02] HAUMONT D., WARZÉE N.: Complete polygonal scene voxelization. *Journal of Graphics, GPU, and Game Tools 7*, 3 (2002), 27–41. 3, 8

[IK00] IX F. D., KAUFMAN A.: Incremental triangle voxelization. In *Graphics Interface* (2000), pp. 205–212. 3

[JC99] JOUPPI N. P., CHANG C.-F.: Z3: An economical hardware technique for high-quality antialiasing and transparency. In *SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (1999), pp. 85–93. 2

[Kal07] KALLIO K.: Scanline edge-flag algorithm for antialiasing. In *Theory and Practice of Computer Graphics* (2007), pp. 81–88. 2

[Kaz05] KAZHDAN M.: Reconstruction of solid models from oriented point sets. In *Symposium on Geometry Processing* (2005), pp. 73–82. 3

[KBH06] KAZHDAN M., BOLITHO M., HOPPE H.: Poisson surface reconstruction. In *Symposium on Geometry Processing* (2006), pp. 61–70. 3

[LB05] LOOP C., BLINN J.: Resolution independent curve rendering using programmable graphics hardware. *Transactions on Graphics 24*, 3 (2005), 1000–1009. 3

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph. 21*, 4 (1987), 163–169. 2, 8

[Lin90] LINHART J.: A quick point-in-polyhedron test. *Computers and Graphics 14*, 3-4 (1990), 445–447. 2

[McC95] MCCOOL M. D.: Analytic antialiasing with prism splines. In *SIGGRAPH* (1995), pp. 429–436. 2

[MN88] MITCHELL D. P., NETRAVALI A. N.: Reconstruction filters in computer-graphics. *SIGGRAPH Comput. Graph. 22* (1988), 221–228. 1

[MPS08] MANSON J., PETROVA G., SCHAEFER S.: Streaming surface reconstruction using wavelets. *Computer Graphics Forum (Proceedings of SGP) 27*, 5 (2008), 1411–1420. 3, 4, 5

[NH08] NEHAB D., HOPPE H.: Random-access rendering of general vector graphics. *SIGGRAPH Asia 27*, 5 (2008), 135:1–10. 3

[ODR09] OVERBECK R. S., DONNER C., RAMAMOORTHI R.: Adaptive wavelet rendering. *SIGGRAPH Asia 28*, 5 (2009), 140:1–12. 3

[QMK06] QIN Z., MCCOOL M. D., KAPLAN C. S.: Real-time texturemapped vector glyphs. In *Symposium on Interactive 3D Graphics and Games* (2006), pp. 125–132. 3

[QMK08] QIN Z., MCCOOL M. D., KAPLAN C.: Precise vector textures for real-time 3d rendering. In *Symposium on Interactive 3D Graphics and Games* (2008), pp. 199–206. 3

[SOM04] SUD A., OTADUY M. A., MANOCHA D.: Difi: Fast 3d distance field computation using graphics hardware. *Computer Graphics Forum 23*, 3 (2004), 557–566. 3

[SP86] SEDERBERG T., PARRY S.: Comparison of three curve intersection algorithms. *Computer Aided Design 18*, 1 (1986), 58–63. 6

[SPG03] SIGG C., PEIKERT R., GROSS M.: Signed distance transform using graphics hardware. In *IEEE Visualization* (2003), pp. 83–90. 3

[TGR04] THON S., GESQUIÈRE G., RAFFIN R.: A low cost antialiased space filled voxelization of polygonal objects. In *GraphiCon 2004* (2004), pp. 71–78. 2

[TWGT10] THÜREY N., WOJTAN C., GROSS M., TURK G.: A multiscale approach to mesh-based surface tension flows. *SIGGRAPH 29*, 3 (2010), 48:1–10. 2

[ZCEP07] ZHANG L., CHEN W., EBERT D., PENG Q.: Conservative voxelization. *Visual Computer 23*, 9 (2007), 783–792. 3, 8

Appendix A: Polygon Coefficient Calculation

Before scaling edges, calculate the $c^{(0,0)}$ coefficient as

$$c^{(0,0)} \mathrel{+}= \frac{1}{2} det(v_0, v_1).$$

Note that we only compute the $c^{(0,0)}$ coefficient for the root node of the quadtree. We compute the other three coefficients for all nodes (interior and leaf, although the quadtree is only refined where the boundary intersects the nodes). Let the edge intersecting the quadrant $Q_{i,j}$ have vertices (v_0, v_1). We define the constant terms K_x, K_y and the linear terms L_x, L_y in each direction as

$$\begin{aligned}
K_x(Q) &= \tfrac{1}{4}(v_{0,y} - v_{1,y}) \\
K_y(Q) &= \tfrac{1}{4}(v_{1,x} - v_{0,x}) \\
L_x(Q) &= \tfrac{1}{8}(v_{0,y} - v_{1,y})(v_{0,x} + v_{1,x}) \\
L_y(Q) &= \tfrac{1}{8}(v_{1,x} - v_{0,x})(v_{0,y} + v_{1,y}).
\end{aligned}$$

The coefficients for the basis functions are then given by summing over all edges intersecting the support of the basis function using the following expressions:

$$\begin{aligned}
c^{(1,0)} \mathrel{+}= {} & L_x(Q_{0,0}) + L_x(Q_{0,1}) + K_x(Q_{1,0}) \\
& - L_x(Q_{1,0}) + K_x(Q_{1,1}) - L_x(Q_{1,1}) \\
c^{(0,1)} \mathrel{+}= {} & L_y(Q_{0,0}) + L_y(Q_{1,0}) + K_y(Q_{0,1}) \\
& - L_y(Q_{0,1}) + K_y(Q_{1,1}) - L_y(Q_{1,1}) \\
c^{(1,1)} \mathrel{+}= {} & L_x(Q_{0,0}) - L_x(Q_{0,1}) + K_x(Q_{1,0}) \\
& - L_x(Q_{1,0}) - K_x(Q_{1,1}) + L_x(Q_{1,1}).
\end{aligned} \tag{8}$$

Appendix B: Bézier Coefficient Calculation

We use the same computation in Equation 8 except that we replace the function calls for L_x, L_y, K_x, K_y to compute the coefficients for a Bézier curve of any degree. We assume that each quadratic Bézier curve is split into quadrants $Q_{i,j}$ and that each piece of the curve within a quadrant has control points (v_0, v_1, v_2). Then $c^{(0,0)}$ can still be written in terms of determinants

$$c^{(0,0)} \mathrel{+}= \frac{1}{3}det(v_0, v_1) + \frac{1}{3}det(v_1, v_2) + \frac{1}{6}det(v_0, v_2)$$

and

$$\begin{aligned}
K_x(Q) &= \tfrac{1}{4}(v_{0,y} - v_{2,y}) \\
K_y(Q) &= \tfrac{1}{4}(v_{2,x} - v_{0,x}) \\
L_x(Q) &= \tfrac{1}{24}(3v_{0,x}v_{0,y} + 2v_{0,y}v_{1,x} - 2v_{0,x}v_{1,y} + v_{0,y}v_{2,x} \\
&\quad + 2v_{1,y}v_{2,x} - (v_{0,x} + 2v_{1,x} + 3v_{2,x})v_{2,y}) \\
L_y(Q) &= \tfrac{1}{24}(2v_{1,y}v_{2,x} + v_{0,y}(2v_{1,x} + v_{2,x}) - 2v_{1,x}v_{2,y} \\
&\quad + 3v_{2,x}v_{2,y} - v_{0,x}(3v_{0,y} + 2v_{1,y} + v_{2,y})).
\end{aligned}$$

Similarly, a cubic Bézier curve with control points (v_0, v_1, v_2, v_3) has a scale coefficient

$$\begin{aligned}
c^{(0,0)} \mathrel{+}= {} & \tfrac{3}{10}det(v_0, v_1) + \tfrac{3}{20}det(v_1, v_2) + \tfrac{3}{10}det(v_2, v_3) \\
& + \tfrac{3}{20}det(v_0, v_2) + \tfrac{3}{20}det(v_1, v_3) + \tfrac{1}{20}det(v_0, v_3)
\end{aligned}$$

and the constant and linear terms in each quadrant are given by

$$\begin{aligned}
K_x(Q) &= \tfrac{1}{4}(v_{0,y} - v_{3,y}) \\
K_y(Q) &= \tfrac{1}{4}(v_{3,x} - v_{0,x}) \\
L_x(Q) &= \tfrac{1}{80}(6v_{2,y}v_{3,x} + 3v_{1,y}(v_{2,x} + v_{3,x}) \\
&\quad + v_{0,y}(6v_{1,x} + 3v_{2,x} + v_{3,x}) \\
&\quad + v_{0,x}(10v_{0,y} - 6v_{1,y} - 3v_{2,y} - v_{3,y}) \\
&\quad - 6v_{2,x}v_{3,y} - 10v_{3,x}v_{3,y} - 3v_{1,x}(v_{2,y} + v_{3,y})) \\
L_y(Q) &= \tfrac{1}{80}(6v_{2,y}v_{3,x} + 3v_{1,y}(v_{2,x} + v_{3,x}) \\
&\quad + v_{0,y}(6v_{1,x} + 3v_{2,x} + v_{3,x}) \\
&\quad - 6v_{2,x}v_{3,y} + 10v_{3,x}v_{3,y} - 3v_{1,x}(v_{2,y} + v_{3,y}) \\
&\quad - v_{0,x}(10v_{0,y} + 6v_{1,y} + 3v_{2,y} + v_{3,y})).
\end{aligned}$$

Appendix C: Triangle Coefficient Calculation

Coefficient calculation for triangles follows the same pattern as for polygons in 2D. Before scaling triangles, calculate the $c^{(0,0,0)}$ coefficient as

$$c^{(0,0,0)} \mathrel{+}= \frac{1}{6}det(v_0, v_1, v_2).$$

We again define functions K_x, K_y, K_z and L_x, L_y, L_z corresponding to the constant and linear portions of the computation in each quadrant as

$$\begin{aligned}
K_x(Q) &= \tfrac{1}{16}(-v_{1,z}v_{2,y} + v_{0,z}(-v_{1,y} + v_{2,y}) \\
&\quad + v_{0,y}(v_{1,z} - v_{2,z}) + v_{1,y}v_{2,z}) \\
K_y(Q) &= \tfrac{1}{16}(v_{0,z}(v_{1,x} - v_{2,x}) + v_{1,z}v_{2,x} \\
&\quad - v_{1,x}v_{2,z} + v_{0,x}(-v_{1,z} + v_{2,z})) \\
K_z(Q) &= \tfrac{1}{16}(-v_{1,y}v_{2,x} + v_{0,y}(-v_{1,x} + v_{2,x}) \\
&\quad + v_{0,x}(v_{1,y} - v_{2,y}) + v_{1,x}v_{2,y}) \\
L_x(Q) &= \tfrac{-1}{48}(v_{0,x} + v_{1,x} + v_{2,x})(v_{0,z}(v_{1,y} - v_{2,y}) \\
&\quad + v_{1,z}v_{2,y} - v_{1,y}v_{2,z} + v_{0,y}(-v_{1,z} + v_{2,z})) \\
L_y(Q) &= \tfrac{1}{48}(v_{0,y} + v_{1,y} + v_{2,y})(v_{0,z}(v_{1,x} - v_{2,x}) \\
&\quad + v_{1,z}v_{2,x} - v_{1,x}v_{2,z} + v_{0,x}(-v_{1,z} + v_{2,z})) \\
L_z(Q) &= \tfrac{-1}{48}(v_{0,y}(v_{1,x} - v_{2,x}) + v_{1,y}v_{2,x} \\
&\quad - v_{1,x}v_{2,y} + v_{0,x}(-v_{1,y} + v_{2,y}))(v_{0,z} + v_{1,z} + v_{2,z}).
\end{aligned}$$

Assuming that $Q_{i,j,k}$ represents the portion of the polygon (triangulated) in the ijk octant with vertices v_0, v_1, v_2, we can then compute the wavelet coefficients using

$$\begin{aligned}
c^{(1,0,0)} \mathrel{+}= {} & L_x(Q_{0,0,0}) + L_x(Q_{0,0,1}) + L_x(Q_{0,1,0}) + L_x(Q_{0,1,1}) \\
& + K_x(Q_{1,0,0}) - L_x(Q_{1,0,0}) + K_x(Q_{1,0,1}) - L_x(Q_{1,0,1}) \\
& + K_x(Q_{1,1,0}) - L_x(Q_{1,1,0}) + K_x(Q_{1,1,1}) - L_x(Q_{1,1,1}) \\
c^{(0,1,0)} \mathrel{+}= {} & L_x(Q_{0,0,0}) + L_x(Q_{0,0,1}) + L_x(Q_{1,0,0}) + L_x(Q_{1,0,1}) \\
& + K_x(Q_{0,1,0}) - L_x(Q_{0,1,0}) + K_x(Q_{0,1,1}) - L_x(Q_{0,1,1}) \\
& + K_x(Q_{1,1,0}) - L_x(Q_{1,1,0}) + K_x(Q_{1,1,1}) - L_x(Q_{1,1,1}) \\
c^{(0,0,1)} \mathrel{+}= {} & L_x(Q_{0,0,0}) + L_x(Q_{1,0,0}) + L_x(Q_{0,1,0}) + L_x(Q_{1,1,0}) \\
& + K_x(Q_{0,0,1}) - L_x(Q_{0,0,1}) + K_x(Q_{1,0,1}) - L_x(Q_{1,0,1}) \\
& + K_x(Q_{0,1,1}) - L_x(Q_{0,1,1}) + K_x(Q_{1,1,1}) - L_x(Q_{1,1,1}) \\
c^{(1,1,0)} \mathrel{+}= {} & L_x(Q_{0,0,0}) + L_x(Q_{0,0,1}) - L_x(Q_{0,1,0}) - L_x(Q_{0,1,1}) \\
& + K_x(Q_{1,0,0}) - L_x(Q_{1,0,0}) + K_x(Q_{1,0,1}) - L_x(Q_{1,0,1}) \\
& - K_x(Q_{1,1,0}) + L_x(Q_{1,1,0}) - K_x(Q_{1,1,1}) + L_x(Q_{1,1,1}) \\
c^{(1,0,1)} \mathrel{+}= {} & L_x(Q_{0,0,0}) - L_x(Q_{0,0,1}) + L_x(Q_{0,1,0}) - L_x(Q_{0,1,1}) \\
& + K_x(Q_{1,0,0}) - L_x(Q_{1,0,0}) - K_x(Q_{1,0,1}) + L_x(Q_{1,0,1}) \\
& + K_x(Q_{1,1,0}) - L_x(Q_{1,1,0}) - K_x(Q_{1,1,1}) + L_x(Q_{1,1,1}) \\
c^{(0,1,1)} \mathrel{+}= {} & L_x(Q_{0,0,0}) - L_x(Q_{0,0,1}) + L_x(Q_{1,0,0}) - L_x(Q_{1,0,1}) \\
& + K_x(Q_{0,1,0}) - L_x(Q_{0,1,0}) - K_x(Q_{0,1,1}) + L_x(Q_{0,1,1}) \\
& + K_x(Q_{1,1,0}) - L_x(Q_{1,1,0}) - K_x(Q_{1,1,1}) + L_x(Q_{1,1,1}) \\
c^{(1,1,1)} \mathrel{+}= {} & L_x(Q_{0,0,0}) - L_x(Q_{0,0,1}) - L_x(Q_{0,1,0}) + L_x(Q_{0,1,1}) \\
& + K_x(Q_{1,0,0}) - L_x(Q_{1,0,0}) - K_x(Q_{1,0,1}) + L_x(Q_{1,0,1}) \\
& - K_x(Q_{1,1,0}) + L_x(Q_{1,1,0}) + K_x(Q_{1,1,1}) - L_x(Q_{1,1,1}).
\end{aligned}$$